**מערכי שיעור פייתון**

שיעור 1.

Getting\_started

This is the most basic action in programming. The '=' sign assigns the value on the right to a variable, whose name is on the left.

A variable is like a box. You can put something in the box (thats called Assignment) and use the contents later for further operation. **Variable's name should be meaningful**.

Python variables do not need explicit declaration (nor type declarations) to reserve memory space. The declaration happens automatically when you assign a value to a variable.

a = 3

b = a + 1

c = b \* 5

my\_number = 12.345

my\_name = 'Liran'

is\_spam = True

result = max(12, 32, 17, c)

print(a)

**Note:** Don't use Python terms for naming your variables and files. E.g. calling your variable list or your file numpy may cause Python to not work properly.

Case sensitivity

Python is a case-sensitive language, which means that whether you use lower-case or upper-case letters makes a lot of difference.

a = 1

A = 2

print (123 + a)

print (123 + A)

124

125

## Comments

If we want to add a comment to our code, we add a **#** sign, and everything after it within the same line will not be ecaluated.

# This line is a comment

# The next line is not a comment

my\_age = 33

## Printing

Very often we would like to print something to the "console", the textual interface. This is acheived by the print function.

print("This will be printed.")

This will be printed.

print() can print several elements, and they should be separated with commas. It automatically adds a white space between them.

my\_name = 'Balu'

my\_age = 10

print("My name is", my\_name, 'and my age is', my\_age)

My name is Balu and my age is 10

The function print() automatically starts a new line (after the printing). This can be modified by the end argument.

for name in ['Arik', 'Bentz']:

    print("My name is", name)

My name is Arik

My name is Bentz

if we wish print not to start a new line, we can change the end argument (the default is a new line)

for name in ['Arik', 'Bentz']:

    print("My name is", name, end='#')

My name is Arik#My name is Bentz#

## Indentation

While most programming languages use various syntax methods to mark a block of code (e.g. { ... }, BEGIN ... END, etc.), Python's uses indentation. This feature gives the language a very "clean" look.

A code block which represents the body of a function or a loop begins with the indentation and ends with the first unindented line.

standard indention is 4 spaces.

my\_name = 'Arik'

other\_name = 'Arik'

if other\_name == my\_name:

    print("You are awesome!")

else:

    print("You are NOT awesome!")

File "<ipython-input-26-81beccff3099>", line 4 else: ^ SyntaxError: invalid syntax

The following variations illustrate the importance of correct indentation.

for x in [1, 2, 3, 4, 5, 6]:

    if x % 2 == 0:

        print(x, "is even.")

    else:

        print(x, "is odd.")

1 is odd.

2 is even.

3 is odd.

4 is even.

5 is odd.

6 is even.

# Self-explanatory examples

**Note:** The following examples are for illustration purposes. Every detail in them will be explained in depth throughout the course.

## Example 1 - Interactive Guess-My-Number game

The following script implements a game, in which the computer randomly chooses a number between 1 and 100, and you have to guess it guided by the computer's hints.

# import necessary module

import random

# The computer "chooses" a number

my\_number = random.randint(1, 100)

# The program asks you to play

print ("I have chosen a number between 1 and 100.")

print ("Please try to guess it...")

# The program reads your guess and stores it

user\_guess = int(input())

# The game is on...

while my\_number != user\_guess:

    if user\_guess > my\_number:

        print ("Wrong! You are too high, please try again...")

        user\_guess = int(input())

    else:

        print ("Wrong! You are too low, please try again...")

        user\_guess = int(input())

print ("Finally :-)\nGAME OVER")

I have chosen a number between 1 and 100.

Please try to guess it...

twenty seven

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

ValueError Traceback (most recent call last)

<ipython-input-29-769d138ba002> in <module>()

**10**

**11** # The program reads your guess and stores it

---> 12 user\_guess = int(input())

**13**

**14** # The game is on...

ValueError: invalid literal for int() with base 10: 'twenty seven'

## Example 2 - Data exploration

**Note:** In this example we will use the Diamonds dataset, which can be found in the *Datasets* folder of the course ([link](https://drive.google.com/drive/folders/15J2Y2y81Uc9Wy8-ceBF7ZQj4C-uQr1dT?usp=sharing)).

**Note:** Google Colab allows you to upload your data to a local destination (under the /content folder). You can do it using the *Upload* button at the top of the *Files* tab on the window on the left of the Google Colab GUI.

The following script reads a file from a given path and make some manipulations to its data.

**Note:** We are making use of the pandas and seaborn packages, which will be an important tool throughout our course.

import pandas as pd

import seaborn as sns

file\_path = 'diamonds.csv'

df\_diamonds = pd.read\_csv(file\_path)

We can now see the actual raw data

df\_diamonds.head()

We can query the data for anything we like, e.g. how many diamonds weigh more than 2.5 carats and cost less than 10000.

len(df\_diamonds[(df\_diamonds.carat>2.5) & (df\_diamonds.price<10000)])

We can explore it with many manipulations, e.g. sort the diamonds Cut categories by their average price.

df\_diamonds.groupby('cut')['price'].mean().sort\_values(ascending=False)

cut

Premium 4584.257704

Fair 4358.757764

Very Good 3981.759891

Good 3928.864452

Ideal 3457.541970

Name: price, dtype: float64

We can make any plot we desire, e.g. the boxplots of the prices of each Clarity category.

sns.boxplot(x='clarity', y='price', data=df\_diamonds)

# Working environment

We will get to know many components of our work environment, but for start we need to be familiar with the three main interfaces:

* **Python** - the engine behind the scenes
* **Jupyter** - the IDE we will use to write our Notebooks during the course
* **Google Colaboratory** - A free cloud service to run your Jupyter notebboks. It supports nearly all the relevant modules for our course (and many more). See [here](https://colab.research.google.com/notebooks/welcome.ipynb) for full details.

**Discussion:** How does it really run with Google Colab? Discuss the responsibility of the data scientist regarding the availibility of necessary infrastructures.

The closer you get to "production" the less convenient it is to work with Jupyter Notebbooks. For simplicity we will stick with Google Colaboratory for te entire course. Nevertheless, two additional environments should definitely be mentioned:

* **Anaconda** - A scientific distribution of Python, which includes hundreds of packages (including most of the packages we will meet in our course).
* **PyCharm** - one of the most common IDEs for python developers

שיעור 2

Data types

# General

The program must know the data type of each variable. It has a significant influence on many techanical aspects:

* Every data type "supports" different operations.
* The same "function" may have different meaning for different data types.
* Every data type is stored differently in the machine memory.

num1 = 1.2

num2 = 2.3

str1 = 'micro'

str2 = 'phone'

print(num1 + num2)

print(num1 \* num2)

print(str1 + str2)

# print (str1 \* str2)  # ERROR!!!

# print (num1 + str1)  # ERROR!!!

3.5

2.76

Microphone

The type of the variable can be retrieved by the type() function.

**Note:** Python is a dynamic language, which means you don't have to specify the type of a variable.

print(type(num1))

print(type(str1))

<class 'float'>

<class 'str'>

At this stage we will discuss four basic data types, but later in the course we will see many more data types and even create some new ones by ourselves.

# Numbers

We will work with two types of numbers:

* integers - round numbers, which are very useful for enumeration.
* float numbers - numbers with decimal point, which are used for accuracy.

print(type(3))

print(type(3.1))

<class 'int'>

<class 'float'>

Python assumes that if a number is given as a round number then it is an integer. If a float is needed, then there are two ways to "force" it:

* add a "fictive" decimal point
* use the conversion function *float()*

print(type(3.0))

print(type(float(3)))

<class 'float'>

<class 'float'>

Obviously, numbers support all the mathematical functions we expect them them to support. Note, however, that the basic Python includes only the most basic arithmetical operations, so things like trigonometric functions, exponential functions, etc. are not readily available.

a = 3

b = 17

print("a =", a, ", b =", b)

print("a + b =", a + b)

print("a - b =", a - b)

print("a \* b = ", a \* b)

print("b / a =", b / a)

print("b mod a =", b % a)

print("b^a =", b\*\*a)

print("max(a, b, 12) =", max(a, b, 12))

a = 3 , b = 17

a + b = 20

a - b = -14

a \* b = 51

b / a = 5.666666666666667

b mod a = 2

b^a = 4913

max(a, b, 12) = 17.

**Your turn:** Evaluate 34−43/25−52

# Strings

String is a **sequence** of **characters**, and they are defined by one of two methods:

* wrapping them with either single-quotes or double-quotes (it doesn't matter)
* Applying the *str()* function on an object.

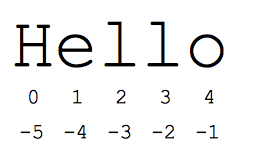
str1 = 'This is a string'

str2 = "This is also a string"

str3 = str(12.34)

print(type(str1), type(str2), type(str3))

Being a sequence, the characters of a string have **index**, that gives direct access to any part of the string. The indexing system is not intuitive at first, but I you'll get it very fast. The picture below explains the indexation.



Two things to note:

* The concept of **negative indexing** is very useful when you want to deal with the end of the string, but you don't know its length. This is the case, for example, when you want to know the type of a file bsed on the file name.
* The indexing system also supports **slicing** using the '**:**' sign. The slicing excludes the last specified index.
* str1 = 'This is a string'
* str2 = "This is also a string"
* str3 = str(12.34)
* print(str1[0])
* print(str3[2])
* print(str1[6:11])
* print(str2[-4:-1])   # Take all BUT the lase character
* print(str3[-4:])     # Take all INCLUDING the lase character
* print(str1[:6])
* print(str1[5:2])     # Backwards indexing returns an empty string
* T
* .
* s a s
* rin
* 2.34
* This i

## Common operations

Like any data type, strings support several operations. Being sequences as well, strings support also sequence operations.

* Sequence types:
  + the len() function - returns the length of the sequence
  + the in operator - test whether an element is within the sequence
  + the + operator - concatenation of two sequences of the same type
  + the index() method - returns the index of an element within the sequence
  + the count() method - counts the number of occurrences of an element within the sequence
* Strings
  + the method replace(old, new) - returns a new string in which all the occurrences of old are replaced with new
  + the method find(sub) - returns the index of sub within a given string. Returns -1 if sub is not in the calling string.
  + the methods upper() and lower() - return a new string with the same letters as the original, but with all the letters in uppercase or lowercase

**Note:** Later in the course, in the [chapter about object-oriented programming](https://drive.google.com/drive/folders/1tvtiiU0hd5kgkCzIHx0xFK78g7wzJDys?usp=sharing), we will understand the exact differene between operators, functions and methods. In the meantime we just need to pay attention to the "dot" syntax, where obj.app(arg) means that the object obj applies the method app() on the argument arg.

### Illustration

str1 = 'Made in Israel'

print(len(str1))

14

print('a' in str1)

print('in' in str1)

print('In' in str1)

True

True

False

str2 = str1[0:8] + "China"

print(str2)

Made in China

print(str1.count('a'))

print(str1.index('in'))

print(str1.find('In'))

2

5

-1

print(str1.upper())

MADE IN ISRAEL

## Immutability

A string cannot be changed after its creation, and the only way to produce new strings objects from it is by ceating new ones. This is a very important feature of the string data type, and we say that strings are **immutable**. During the course we will see the importance of this characteristic and compare it to other data types.

What happens when we execute the *replace()* method on the string *str1*?

str1.replace('Israel', 'China')

print(str1)

Made in Israel

**Nothing!** Since strings are immutable, str1 cannot be modified, and therefore the result of the replace method actually creates a new string, which can be assigned to a new variable, e.g. str2.

str2 = str1.replace('Israel', 'China')

print(str1)

print(str2)

Made in Israel

Made in China

The immutability of strings can be also demonstrated by the error raised when we try to use the indexation for assignment.

str1[8:] = 'China'

**Reference:** More about the mutability concept and other related functionalities can be found [in Ventsislav Yordanov's Medium article](https://towardsdatascience.com/https-towardsdatascience-com-python-basics-mutable-vs-immutable-objects-829a0cb1530a).

## String literals

String literals are auxiliary strings that allow to use characters which are either invisible or raise technical issues when used. To symbolize these characters, you havee to use the backslash **escape character** (\). The full list of string literals can be found [here](https://docs.python.org/3/reference/lexical_analysis.html#literals), and the following are the most common:

* \n - New line
* \t - Tab
* \' - Single quote
* \" - Double quote
* \\ - Backslash

**Note:** The escape character is part of the string.

### Illustrations

print('First line.\nSecond line.')

print('Israel\t:\tJerusalem\nSpain\t:\t?\n')

print('(a) Paris\t(b) Rome\t(c) Madrid\t(d) Berlin')

print('He is John')

print('He\'s John')

The existance of two string wrappers enables the following trick...

print('And god said: There was light')

print('And god said: \"There was light\"')

#print ("And god said: "There was light"")  # This raises an error

print('And god said: "There was light"')

**Your turn:** Take the last string ('And god said: "There was light"') and extract god's saying.

String literals are characters like any other.

len('a\nb')

**Your turn:** Print the following text in a more readable fashion with proper newlines and tabs.

s = 'Sunday: visitors: 20, unique: 13, Monday: visitors: 25, unique: 14, Tuesday: visitors: 31, unique: 19'

print(s)

Sunday: visitors: 20, unique: 13, Monday: visitors: 25, unique: 14, Tuesday: visitors: 31, unique: 19

## multi-lines string

String can also be wrapped with three characters, using either single-quote or double-quote, allowing to write text with quotes and new lines as you normally do:

text = '''A String warpped with triple quotes can extend over

multiple lines like this one, and can contain 'single'

and "double" quotes without using string literals.'''

print(text)

## Conversions

Strings can be concatenated, but this may get a little bit tricky when combined with other data types.

num1, num2 = 1234, 4321

the\_sum = num1 + num2

print('The sum of', num1, 'and', num2, 'is', the\_sum)

**Note:** The print() function does not concatenate strings, but rather print them one by one, adding a white space between them.

# sentence = 'The sum of ' + num1 + ' and ' + num2 + ' is ' + the\_sum

sentence = 'The sum of ' + str(num1) + ' and ' + str(num2) + ' is ' + str(the\_sum)

print(sentence)

# Booleans

Boolean variables can have only True and False values, and they are also called "binary" and "logical" variables. Boolean variables can be created directly by assigning True or False to them, however it is much more common to create them with comparisons and tests.

is\_connected = False

type(is\_connected)

bool

num1, num2 = 12, 27

print(num1 == num2)

print(num1 != num2)

print(num1 < num2)

print(num1 >= num2)

False

True

True

False

str1 = 'Python'

print('p' in str1)

print(str1[3:5] == 'ho')

## Common operations

Booleans support less known operations, which are:

* *not* - the negative
* *and* / *&* - return *True* if and only if both of its sides are *True*
* *or* / *|* - return *False* if and only if both of its sides are *False*
* *^* (xor) - return *True* if and only if exactly one of its sides is *True*

x, y = True, False

print(x)        # True

print(not x)    # False

print(x & y)    # False

print(x and y)  # False

print(x | y)    # True

print(x or y)   # True

print(x ^ y)    # True

## Illustration

avg\_height, avg\_weight = 1.79, 75

my\_height, my\_weight = 1.77, 112

am\_i\_tall = my\_height >= avg\_height

print(am\_i\_tall)

False

am\_i\_fat = my\_weight >= avg\_weight

print(am\_i\_fat)

True

am\_i\_giant = am\_i\_tall and am\_i\_fat

print(am\_i\_giant)

False

am\_i\_chubby = am\_i\_fat and not am\_i\_tall

print(am\_i\_chubby)

True

am\_i\_abnormal\_1 = (am\_i\_fat and not am\_i\_tall) or (am\_i\_tall and not am\_i\_fat)

am\_i\_abnormal\_2 = am\_i\_fat ^ am\_i\_tall

print(am\_i\_abnormal\_1, am\_i\_abnormal\_2)

True True

**Note:** In addition to simple Booleans, Python allows [Truth-value testing](https://docs.python.org/3/library/stdtypes.html#truth-value-testing) with non-Boolean variables

# Exercises

## Exercise 1

1. Assign your height in meters to a variable called height and your weight in kilograms to a variable called weight.
2. Calculate your BMI and assign the value to a variable called BMI. (BMI=wh2)
3. Print the sentence "My height is \_\_\_ meters, my weight is \_\_\_ Kgs, and my BMI is \_\_\_.”.
4. Print the sentence, so that each part of it will be in a new line.

### Solution

Weight, Height = 104, 1.77

BMI = Weight / Height\*\*2

print("My height is " + str(Height) + " meters, my weight is " + str(Weight) + " Kgs, and my BMI is " + str(BMI) + ".")

print("My height is " + str(Height) + " meters,\nmy Weight is " + str(Weight) + " Kgs,\nand my BMI is " + str(BMI) + ".")

## Exercise 2

1. Assign your first name and your last name to the variables first\_name and last\_name, respectively.
2. Create your full name by concatenating first\_name and last\_name with a space in the middle, and assign the result to a variable called full\_name.
3. Use the string method replace() to change all the vowels in your full name from lower case to upper case (Vowels are the letters 'a', 'e', 'i', 'o' & 'u').
   * Note that you are not allowed to use the for syntax at the moment.

first\_name, last\_name = 'Amit', 'Rappel'

print(len(first\_name) + len(last\_name))

full\_name = first\_name + ' ' + last\_name

upper\_aeiou\_name = full\_name.replace('a','A').replace('e', 'E').replace('i', 'I').replace('o', 'O').replace('u', 'U')

print(upper\_aeiou\_name)

## Exercise 3

Define proper variables with the following data for yourself and for one of your imaginary friends: gender (string), age (int), city (string) and marital status (Boolean). Then create the following Boolean variables:

1. same\_city – do both of you live in the same city?
2. is\_older – is your friend older than you?
3. diff\_gender – are you from different genders?
4. both\_married – are you both married?

### Solution

my\_gender, other\_gender = 'M', 'F'

my\_age, other\_age = 33, 29

my\_city, other\_city = 'Kfar-saba', 'Kfar-saba'

my\_marriage, other\_marriage = True, False

same\_city = my\_city == other\_city

is\_older = other\_age > my\_age

diff\_gender1 = (my\_gender == 'M' and other\_gender == 'F') or \

                   (my\_gender == 'F' and other\_gender == 'M')

diff\_gender2 = my\_gender != other\_gender

diff\_gender3 = not(my\_gender == other\_gender)

both\_married = my\_marriage & other\_marriage

שיעור 3

Data structures

# Introduction

In many cases the data we work with has to be organized. This is obvious when we talk about tables and databases, but it is not less crucial when we deal even with the simplest data objects. Every program language provides its programmers with many built-in data structrues to allow this "organization". It is up to us, though, to use the most appropriate structure for each task.

Similar to the data types, each data structure supports specific operations, which are optimized to its characteristics. Python provides several dozens of built-in data structures, and in this chapter we will learn about the most notable four:

* list
* tuple
* dictionary
* set

As noted earlier, one of the most important characteristics of a data structure is its (im)mutability, therefore it is important to keep in mind which data structure is mutable and which is not.

# Lists

List is a sequence of elements. That's it.

**Reminder:** *sequence* is Python's generic term for an ordered set of elements. Strings are also sequences.

To create a list we usually use the [] constructor, either with or without elements, separated by commas if present. Alternatively we can use the list() function, which is useful in some specific scenarios. The elements don't have to be of the same type, and they can even be of type *list* themselves.

list0 = []

list1 = [4.2, 'House', 3, False, ['a', False, 1, 'book'], [], 'dog']

Lists are sequences

Lists are sequences, so they support all the sequences operations we already met when we learned about strings. These include indexing and slicing, the *len()* function, the *in* operator, concatenation with the '+' sign, the *index()* method, etc.

### Indexing and slicing

Specific index returns the element itself (which may be a list itself)

print(list1[3])

print(list1[-3])

False

['a', False, 1, 'book']

Slicing returns a list

print(list1[3:5])

print(list1[2:-1])

print(list1[2:])

[False, ['a', False, 1, 'book']]

[3, False, ['a', False, 1, 'book'], []]

[3, False, ['a', False, 1, 'book'], [], 'dog']

print(list1[2])

print(list1[2:3])

3

[3]

It should be noted that when we refer to a list as an element of a list, then we should use **chained indexing** and not think of it as a two-dimensional object.

print(list1[4][3][-1])

book

**Your turn:** For the same list1, calulcate the following expressions by yourself, and then check with Python (note also the data type of each answer):

* list1[1][-2]
* str(list1[0])[1]
* len(list1[-1])
* list1[len(list1[1])-1][0][0]

### Other illustrations

details = ['John', 'Doe', 'm', 32, 'Tel Aviv', False, 1.71]

print(len(details))

7

print('John' in details)

print('Jon' in details)

print(['John'] in details)

True

False

False

print(details[:3] + details[4:5])

['John', 'Doe', 'm', 'Tel Aviv']

**Note:** Why will details[:3] + details[4] not work?

print(details.index(32))

3

## Common operations

**Reference:** The full list of methods can be found in the [official docs](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types).

### Lists are mutable

Unlike strings, lists are **mutable** objects, which means their value(s) can change during runtime.

print(details)

details[3] = 33

print(details)

['John', 'Doe', 'm', 32, 'Tel Aviv', False, 1.71]

['John', 'Doe', 'm', 33, 'Tel Aviv', False, 1.71]

As we just saw, this can be done by simply reassigning the values of the elements in the list. However, as we will see below, it is much more convenient to apply the various methods which allow to modify their content. Such methods modify the list object **in-place** and usually do **not** return the modified list itself. What they do return is None.

### Adding elements - append(), insert() & extend()

* append(x) - adds the element x at the end of the list
* insert(i, x) insert the element x at the i-th place
* extend(iter) - concatenate the elements of iter` to the list
* list1 = ['Aa', 'Bb']
* print(list1)
* list1.append('Dd')
* print(list1)
* list1.insert(2, 'Cc')
* print(list1)
* list1.extend(['Ee', 'Ff'])
* print(list1)
* list1.append(['Gg', 'Hh'])
* print(list1)
* list1.extend('Ii')
* print(list1)

['Aa', 'Bb']

['Aa', 'Bb', 'Dd']

['Aa', 'Bb', 'Cc', 'Dd']

['Aa', 'Bb', 'Cc', 'Dd', 'Ee', 'Ff']

['Aa', 'Bb', 'Cc', 'Dd', 'Ee', 'Ff', ['Gg', 'Hh']]

['Aa', 'Bb', 'Cc', 'Dd', 'Ee', 'Ff', ['Gg', 'Hh'], 'I', 'i']