**Python**

**Functions** parentheses indicate that *print(‘hello world’)* is a function

**New line** \n

**Formatting**

In Python programming, code formatting is crucial. Things like indentation, spaces, tabulation will make a huge difference - and you will soon realise (even the hard way). So, if you're not using an editor of the one listed above, please make sure that your editor can properly setup spacing settings, removing any tab and use 4 spaces as indentation. This is not mandatory for the Python interpreter, but it is the general convention used.

In general, adding blank lines to appropriate places is a good idea. If you are asked to "fix code", feel free to add missing blank lines.

Capitalize names.capitalize() → input will begin from capital letter

Begin with capital letter names.title() → every word will begin from a capital letter

**Importing**

Math import math

Statistics import statistics

**Variables**

Line of code. Like *a = 2* is a variable.

Variables can only contain letters, numbers, and underscores (symbol \_). Names can start only with a latter or an underscore, not with a number. Spaces are not allowed. Names should be descriptive without being too long: *mc\_wheels* is better than *wheels* or *number\_of\_wheels\_on\_a\_motorcycle*. Name should be the same. E.g. *name* = *name* but *name* != *Name*.

Python keywords that cannot be used as names:



Keywords don’t need parentheses like print().

Be careful about letter l (lowercase) and O (uppercase). They could be confused with 1 and 0.

**Type of data**

**Integer, int** numbers like 2, 3, 40

**Floating-point, float, число с плавающей запятой** numbers like 2.3, 4.32, 54.34

**String (строка букв), str** text like ‘Hello world’ or ‘2’ or ‘2.3’ (numbers in quotation marks)

Strings are contained either like this “double-quoted” or like this ‘single-quoted’.

Quotation inside a string:

Graphical user interface, text, application, email

Description automatically generated

Use triple-quote ‘’’ or “”” for preserving formatting between lines:

Graphical user interface, text, application, email, Teams

Description automatically generated

**Check type of data** type(‘hello world’) → str type(17) → int

Another type of data is None. It is Python’s ‘nothing’ value. It behaves just like any other value, and it's often used as a default value for different kinds of things.

Graphical user interface, text, application

Description automatically generated

**Booleans**

There are two Boolean values: True and False.

The *equal* symbol = is assigning a = 1 means *a* is set to *1*

Two *equal* symbols == are comparing a == 1 means *check if* a *is equal to* 1 / *does* a *equal* 1

Alternative to *a == 1* is *(a == 1) == True*. However, because of readability simply use *a == 1*.

Empty or blank values are seen as False by Boolean and non-empty values are seen as True:

Graphical user interface, text, application, email

Description automatically generated

Background pattern

Description automatically generated with low confidence

**Examples of data type:**

**LIST = [‘A’, ‘B’]**

A **list (square brackets)** is a collection of items that is stored in a variable (lists are collections of objects). The items should be related in some way, but there are no restrictions on what can be stored in a list.

Naming convention: plural name like *cars*, *dogs*. Each item is then a *car*, a *dog*. This gives you a straightforward way to refer to the entire list (dogs), and to a single item in the list (dog).

Example:

Graphical user interface

Description automatically generated with medium confidence

**Example of a list with two coupled variables like Name/Surname** people\_list = [("Alex", "De Jong"), ("John", "van Maarten"), ("Louis", "Verbeek")]

Retrieve a value: people[0] → ‘Alice’

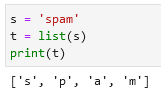
Retrieve the last value: people[-1] → ‘Charlie’

Length of a list: len(people) → 3

Update list: people[1] = “John” → people → [“Alice”, “John”, “Charlie”]

Create an empty list: list = []

Convert a string (text) to a list:

 This method breaks a string into individual letters.

If you want to break a string into words use the split method:

A picture containing graphical user interface

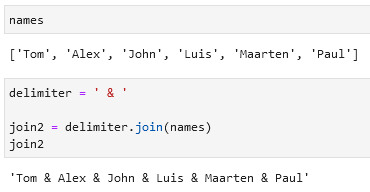
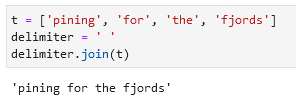
Description automatically generated

To split a string by a delimiter add an optional argument:

Graphical user interface, text, application

Description automatically generated

You can also join values in a list using a .join() and a delimiter. Two examples:



The + operator concatenates lists:

Scatter chart

Description automatically generated with low confidence

The \* operator repeats the list a given number of times:

A screenshot of a computer

Description automatically generated with low confidence

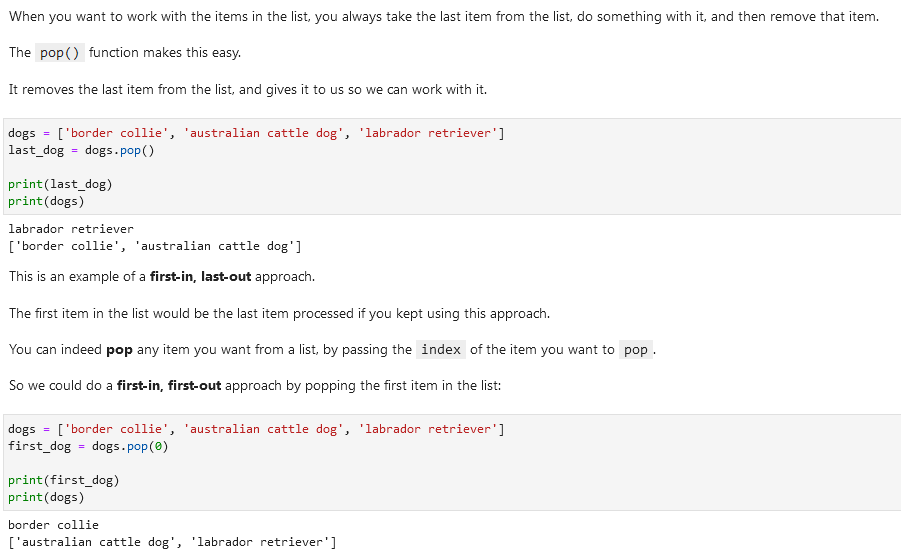
Add 1 value to a list using .append():



Delete the last 4 values in the list using del:



Remove one last item / remove one first item:



**TUPLE = (‘A’, ‘B’)**

Graphical user interface, text, application

Description automatically generated

Tuples are basically lists that can never be changed.

Lists are quite dynamic; they can grow as you append and insert items, and they can shrink as you remove items. You can modify any element you want to in a list. Sometimes we like this behavior, but other times we may want to ensure that no user or no part of a program can change a list. That's what tuples are for.

Technically, lists are mutable objects and tuples are immutable objects. Mutable objects can change (think of mutations), and immutable objects can not change.

If you try to change the tuple using e.g. append() you will get an error. The same kind of thing happens when you try to remove something from a tuple, or modify one of its elements. In sum: once you define a tuple, you can be confident that its values will not change.

Tuple unpacking means:

A picture containing calendar

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, Word

Description automatically generated

**SET = {‘A’, ‘B’, ‘C’}**

Graphical user interface, chat or text message

Description automatically generated with medium confidence

A set object is an unordered collection of distinct hashable objects. Common uses include membership testing, removing duplicates from a sequence, and computing mathematical operations such as intersection, union, difference, and symmetric difference.

Operations in sets:

Example 1

Graphical user interface, application, Word

Description automatically generated

Example 2

Text

Description automatically generated

Example 3

Graphical user interface

Description automatically generated with medium confidence

**Difference between lists, tuples, and sets:**

Lists:

* Ordered collection of elements (i.e. elements have a defined order and can be accessed by their index)
* Mutable (i.e. elements can be added, removed, and modified)
* Can contain duplicate elements
* Defined using square brackets [] or using the list() constructor function

Tuples:

* Ordered collection of elements
* Immutable (i.e. elements cannot be added, removed, or modified once the tuple is created)
* Can contain duplicate elements
* Defined using parentheses () or using the tuple() constructor function

Sets:

* Unordered collection of unique elements (i.e. no duplicates allowed)
* Mutable (i.e. elements can be added and removed)
* Elements are not accessed by index, but rather by value (i.e. you can check if an element is in a set, but you can't access it directly by index)
* Defined using curly braces {} or using the set() constructor function

Some similarities between lists, tuples, and sets:

* Can contain elements of any data type (strings, integers, floats, etc.)
* Can be iterated over using loops
* Can be used in combination with built-in Python functions like len() and sorted()

You would use a list when you need to store an ordered sequence of items that can be changed. For example, a list is useful when you want to keep track of a to-do list or a shopping list, where you may need to add or remove items.

You would use a tuple when you need to store an ordered sequence of items that cannot be changed. For example, a tuple is useful when you want to store the coordinates of a point in two-dimensional space, where the x and y values are fixed.

You would use a set when you need to store a collection of items that are unique and unordered. For example, a set is useful when you want to keep track of a list of unique words in a document, where the order of the words doesn't matter.

Here are some specific examples:

* List: todo\_list = ['shower', 'brush teeth', 'mow lawn', 'eat brains']
* Tuple: point = (3, 5)
* Set: unique\_words = {'apple', 'banana', 'orange', 'pear'}

Of course, these are just a few examples and there are many other use cases for lists, tuples, and sets depending on the specific problem you are trying to solve.

[] () {}

Table

Description automatically generated

[] () {}

Table

Description automatically generated

**Hashable / non-hashable objects**

In Python, a hashable object is an object that can be hashed, meaning it can be used as a key in a dictionary or an element in a set.

Hashable objects must have a hash value that remains the same throughout their lifetime (immutable). This means that if two hashable objects are equal, their hash values must be equal as well.

**Examples of hashable objects** include **integers, floats, strings, and tuples** (as long as their elements are also hashable).

On the other hand, non-hashable objects are objects that cannot be hashed, meaning they cannot be used as keys in a dictionary or elements in a set.

**Examples of non-hashable objects** include **lists, dictionaries, and other mutable objects**, which can change their contents or structure over time.

**DICTIONARY = {‘A’: 1, ‘B’: 2, ‘C’: 3}**

A dictionary is like a list, but more general.

In a list, the indices have to be integers *(numbers)*; in a dictionary they can be (almost) any type.



You can think of a dictionary as a **mapping** *(links)* between a set of indices (which are called keys) and a set of **values**. Each key maps to a value *(links to a value).*



The association of a key and a value is called a **key-value pair** or sometimes an **item**.

A dictionary connects two pieces of information. Those two pieces of information can be any kind of data structure in Python.

Create an empty dictionary: ruseng = {} or ruseng = dict()

What can be stored in a dictionary:

Text

Description automatically generated

**Add new key-value pair**

Dictionary\_name[‘new key name’] = new\_value

Graphical user interface, text, email

Description automatically generated

Add a new key-value pair where the value is equal to a different key:

Text

Description automatically generated

**Modify a value**

Table

Description automatically generated with medium confidence

**Remove key-value pair using .pop()**

If you give a name to the .pop line you can select the removed value

Text

Description automatically generated

Remove key-value pair using .popitem(). It will remove the last inserted key-value pair & also return it.

Graphical user interface, text, application, email

Description automatically generated

**Lists in a dictionary**

A picture containing calendar

Description automatically generated

**Traversing through dictionaries**

Text, letter

Description automatically generated

**Loop through all key-value pairs** using .items(). This code pulls all key-value pairs from a dictionary into a list of tuples.

Text

Description automatically generated

**Looping through all keys** in a dictionary using .keys() or without it:

A picture containing table

Description automatically generated Text

Description automatically generated

Both get the same result. This is because when you iterate over a dictionary using a for loop, it only iterates over the keys by default. In other words, in the line for key, value in my\_dict:, the variable key will be assigned to the key of each key-value pair in my\_dict, while the variable value will be left unassigned. To fix this error, you need to use the .items() method to iterate over both keys and values of the dictionary.

**Get the value of a specific key:**

Graphical user interface, text, application, email

Description automatically generated

Alternative example:

A picture containing diagram

Description automatically generated

Text

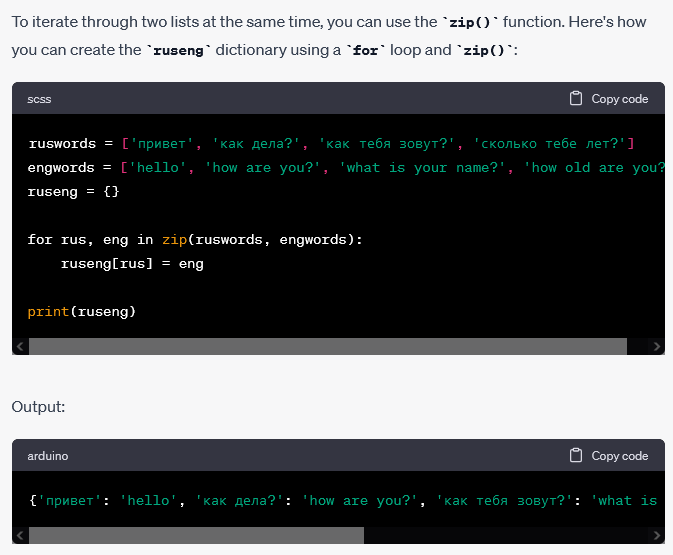
Description automatically generated

**Looping through all values** in a dictionary using .values():

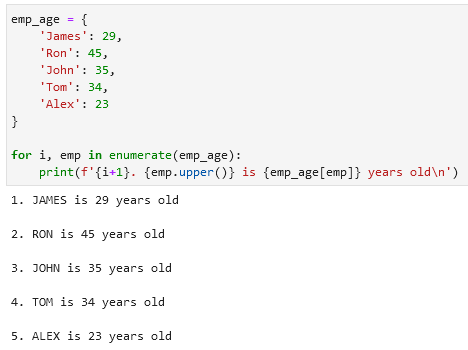
Graphical user interface, text, application

Description automatically generated

**Traverse through two lists / iterate through two lists** at the same time using zip():



Enumerate through a dictionary / print a dictionary:



Enumerate through a dictionary with multiple values per key:

Graphical user interface, text, application, email

Description automatically generated

**IMPORT statistics**

**Calculate average of a dictionary’s values** using .mean():

Text

Description automatically generated

**Dictionary within a dictionary**

Print-out a dictionary within a dictionary:

Text

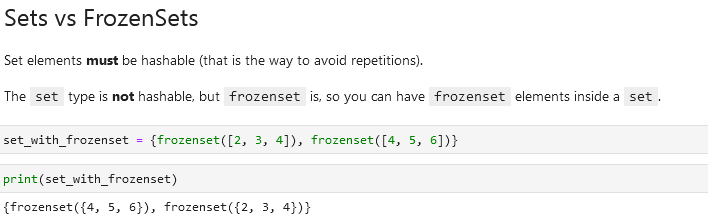
Description automatically generated

**Note on nesting**

While one level of nesting is really useful, nesting much deeper than that gets really complicated, really quickly. There are other structures such as classes which can be even more useful for modeling information. In addition to this, we can use Python to store information in a database, which is the proper tool for storing deeply nested information.

Often times when you are storing information in a database you will pull a small set of that information out and put it into a dictionary, or a slightly nested structure, and then work with it. But you will rarely, if ever, work with Python data structures nested more than one level deep

**FrozenSets**



**Named Tuples**

**COLLECTIONS** First you need to *import collections* 

Example 1:

Graphical user interface, text, application, email

Description automatically generated

Example 2:

Text

Description automatically generated

My example:



**Deques**

See also *Remove one last item / remove one first item:*

If you want to **add/remove a value** to a list *deques* can be an alternative.

**COLLECTIONS** First you need to *import collections* 

Then you need to transform your list into a *deque*:

Graphical user interface, text, application

Description automatically generated

Use *appendleft* or *append* to add values at the beginning/end:

Graphical user interface, text, application

Description automatically generated

If you want to select the first/last value use *popleft()* and *pop():*

Graphical user interface, application

Description automatically generated

These values will be removed from the *deque*. Use *list()* to transform the deque to a list.

**Operators and Operands**

Operators are special symbols that represent computations like addition and multiplication. The values the operator is applied to are called operands.

Operators:

+ addition

- subtraction

\* multiplication

/ division

\*\* exponentiation

% [modulo operator](https://www.freecodecamp.org/news/the-python-modulo-operator-what-does-the-symbol-mean-in-python-solved/) used to check if a number is чётный/нечётный (even/uneven number)

Text

Description automatically generated

\*\* or *exponentiation* называется по-русски *в степени*. То есть 5 \*\* 2 это 5^2 или 52 или 5 \* 5.

В некоторых других языках символ ^ используется для *степени*, но в питоне это [bitwise operator](https://wiki.python.org/moin/BitwiseOperators) called XOR.

**Comparing operators**

Graphical user interface, text, application

Description automatically generated

There can also be multiple comparisons.

Graphical user interface, text

Description automatically generated

Examples:

Graphical user interface, text, application

Description automatically generated

**Order of operations**

A picture containing text

Description automatically generated

**String operations**

There are mostly two: *+* and *\**

Graphical user interface, text, application, email

Description automatically generated

More elaborate:

Graphical user interface, text, application, email

Description automatically generated

**Floating-point operations**

Could have unexpected results like here:

Graphical user interface, text, application

Description automatically generated

Don’t worry, this is the way computers represent numbers internally.

More elaborate:

Graphical user interface, text, application, email

Description automatically generated

**Commenting and comments**

Use hashtag symbol *#* to leave a comment.

Everything from the # to the end of the line is ignored.

Comments are most useful when they document non-obvious features of the code. It is reasonable to assume that the reader can figure out what the code does; it is much more useful to explain why.

Example:





Another way to leave a comment is using “””:

Text

Description automatically generated

**Functions**

An example of a function is *print()*. If you just execute *print* you will get then <function print> as output.

Functions do something when they are *called* (executed in my words) by typing their names and parentheses. Inside the parentheses, we can pass some arguments too. In *print(‘hello’)* the function is *print* and we give it one argument *‘hello’*.

Sometimes people think that doing func = print('hello') means that Python is going to print hello every time we type func. But this is not correct! print('hello') runs print right away, and if we type func later, it's not going to run print('hello') again.



In the context of programming, a function is a named sequence of statements that performs a computation.

When you define a function, you specify the name and the sequence of statements. Later, you can “call” (sometime "invoke") the function by name.

The left side of a function has always to be a name like a = 2 +2, not 2 + 2 = a.

**Module**

Python has a *math* module that provides most of the familiar mathematical functions.

A *module* is a Python file that contains a collection of related functions.

To access a function in a module you need to specify the name of the module then dot (also known as a period) then the name of the function. Like *math.log10()*. This format is called dot notation (запись через точку).

**Python functions**

Print() Like print(‘hello’) → hello

Input() Like input(‘Enter something:’) → Enter something \_\_\_\_ input line. If you assign input to something like abc = input(‘enter:’) and then write a text in the input line like texttext, when you run abc you will get ‘texttext’

Graphical user interface, application

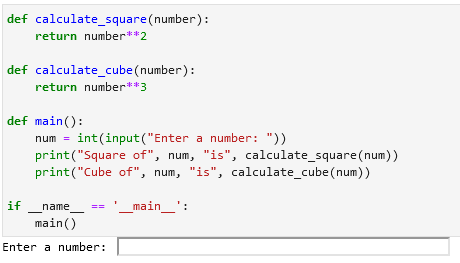
Description automatically generated

**\_\_main\_\_ function**

Graphical user interface, text, application, email

Description automatically generated

Своими словами: ты написал 2 функции и ещё последнюю функцию назвал main, которая сводит первые две функции в одну. Если ты хочешь чтобы твоя третья функция main сразу же была активизирована, то ты пишешь в конце if \_\_name\_ и так далее. Пример из Chart GPT:



Альтернативой может быть просто main(), но тогда:

Text, letter

Description automatically generated

**Math module functions**

Math.log() compute logarithms

Math.sin() get the sinus, also .cos, .tan etc.

Math.pi gets the π value / pi value

Math.sqrt() get the square root of a number like math.sqrt(16) = 4.0

Math.exp ?

**Converting data / data conversion / datatype conversion / change data type**

Change data to integer (to int) int(‘32’) → 32

won’t work on *int(‘dadw’)*

*int(2.99) → 2* it will drop the part after the point (here .99)

Convert data to float float(3) → 3.0 float(‘3.2342’) → 3.2342

Convert data to string (to str) str(32) → ‘32’

**If-statements**

There two are the same:

Graphical user interface

Description automatically generated with medium confidenceA picture containing diagram

Description automatically generated

However the second is better since by default python considers blank values as false. So we can say that if message is not blank then print what was entered, otherwise print that you did not enter anything. Check the Boolean section for more info.

Same code (last is the best):

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text

Description automatically generatedGraphical user interface, text, application

Description automatically generated

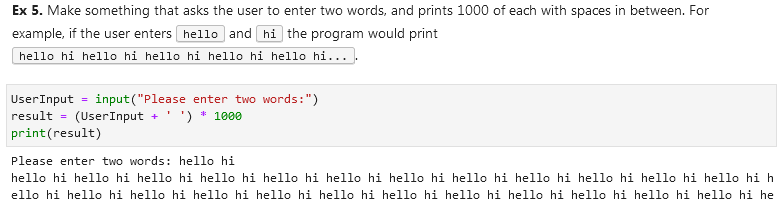
Another example:

Graphical user interface, text, application

Description automatically generatedText

Description automatically generated

Make a space between two values:



A statement should be equal to null (should be blank):

Graphical user interface, text, application, email

Description automatically generated

**Function definition / define a function**

Text

Description automatically generated

Graphical user interface, text, application, email

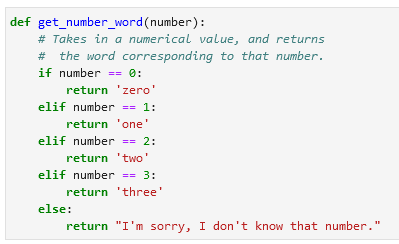
Description automatically generated

Function example:

Graphical user interface, text, application, email

Description automatically generated

Function example:



Function example:

Graphical user interface, text, application

Description automatically generated

**Setting a default value in a function.** In this case the default value is *‘everyone’:*

Graphical user interface, text, application, email

Description automatically generated

If you have multiple arguments like def thank\_you(name, surname) you don’t need to input your arguments like thank\_you(‘John’, ‘Smit’). This will also work: thank\_you(surname=’Smit’, name=’John’). By keywording arguments you can place them in arbitrary order.



Text

Description automatically generated with low confidence

**Assigning two or more names to a tuple / extracting two or more variables from a tuple:**

Graphical user interface, application

Description automatically generated

1. If we have a list of elements the first and last variables will always select first and last elements. E.g. if we have 10 elements and 2 variables we will only select first and last element.

2. If I add extra variables between these (first and last) variables I will only select 2nd, 3rd etc. elements, but I other elements (e.g. from 4 to 9) will not be selected.

3. If I want to assign all other elements e.g. from 2 to 9 to a variable I need to add an asterisk before that variable.

4. Asterisk should be before that variable without any spaces.

Example:

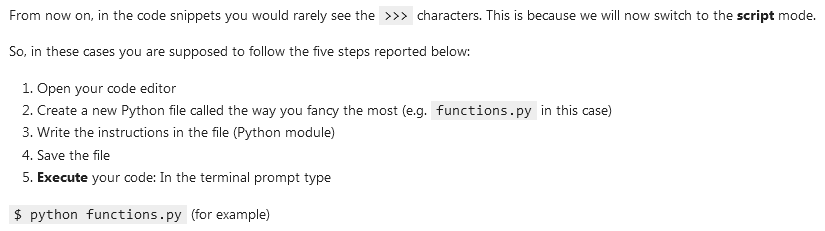


a = 1

c = [2, 3, 4]

b = 5

**VSCode executing code**

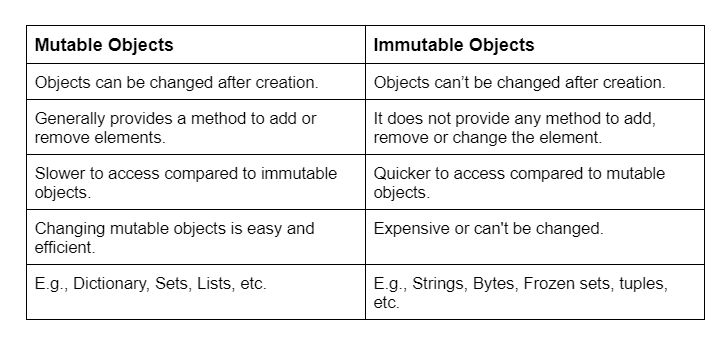


**Sequences**

* **Container sequences** can hold items of different types. They hold reference to the objects they contain, which may be of any type. Examples:
  + list
  + tuple
  + collections.deque
* **Flat sequences** hold items of one type. They physically store the value of each item within its own memory space, and not as distinct objects. They are more compact, but are limited to holding primitive values like characters, bytes, and numbers. Examples:
  + str
  + bytes
  + bytearray
  + memoryview
  + array.array

Another way of grouping sequence types is **mutability**:

* **Mutable** sequences:
  + list
  + bytearray
  + array.array
  + collections.deque
  + memoryview
* **Immutable** sequences:
  + tuple
  + str
  + bytes



**for loop / for iteration loop / iterations / traversing**

This most widely used iteration mechanism in Python. Every sequence can be iterated (element by element). Also *while* loops are permitted, but the *for* loop is the one you’ll see and use most of the time. Strings and lists are iterable, but integers and floats are not

Example:

A picture containing text

Description automatically generated

Printing out names from the list called *names:*





Example of a *for* loop that prints out people’s names using a function *greeter:*

A picture containing diagram

Description automatically generated

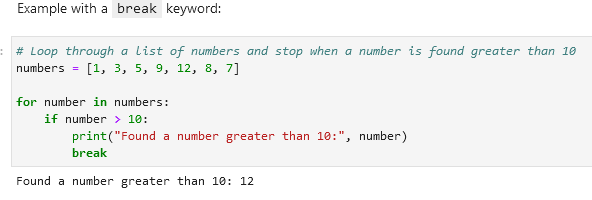
*for* \*arbitrary name of the variables in the list\* *in* \*name of the list\*:

\*function name\*(\*arbitrary name of the variables in the list\*)

Example of a for loop with two or more variables:



**Break** and **continue** keywords in a *for* loop:



Graphical user interface, text

Description automatically generated Graphical user interface, text, application, email

Description automatically generated

**Enumerate a loop:**

Instead of *for dog in dogs* you add *for index, dog* and you add *enumerate(dogs):*

Graphical user interface, text, application

Description automatically generated

The value in the variable index (i) is always an integer (number). If you want to print it in a string turn the integer into a string like here:



Another example for **enumerating in a pair**:

Text

Description automatically generated with low confidence

**Manually iterating**

Use .\_\_next\_\_() to iterate manually:

Graphical user interface, text, application, chat or text message, email

Description automatically generated

Alternative function that does the same thing is next():

Graphical user interface, text, application

Description automatically generated

**Iter() iteration**

There's a built-in function called iter() that converts anything iterable to an iterator.



Iterating using iter(): Alternative:

Text

Description automatically generated Graphical user interface, text, application

Description automatically generated

**Retrieve a specific part of e.g. a list:**

Make use of an additional row captures only the needed rows. In this case it is the *first\_batch = usernames[0:3]* row that selects only the first 3 items in the list:

Graphical user interface, text

Description automatically generated

**Examples of iterators**

enumerate(iterable)

range(start, end, step)

dictionary.keys()

dictionary.values()

dictionary.items()

Example of a .range():  
Graphical user interface, application

Description automatically generated

**Functions**

range() range(start, end, step) Generates a list of numbers between *start* and *end* with a given *step:* **range(0, 20, 2)**

The last value will not be included.

Graphical user interface, text, application

Description automatically generated

**Check if data exists in your list/tuple/set:**

A picture containing calendar

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

**IMPORT statistics**

Calculate average of a dictionary’s values using .mean():

Text

Description automatically generated

|  |  |
| --- | --- |
| **Easter egg** | Import this |
| **Import pandas** | import pandas as pd |
| **Import NumPy** | import numpy as np |
| **Series (a column)** | pd.Series([21, 22, 23, 24], name = ‘age’)  *age = column name* |
| **Select an element (row; from 0 to …)** | table\_name[3]  *Select the 4th row* |
| **Make a custom axis to a Series** |  |
|  |  |
| **Index** is also called an **axis**; each element is called axis label.  Data in columns is called as **values**.  A series ideally should have the same datatype throughout its values (same format for the whole column). |  |
| **Dataframe** | A series is a column and a dataframe has multiple columns |
| **Create a dataframe** |  |
| **Columns and rows in a dataframe (table)** | df.shape  (3, 7)  3 rows and 7 columns |
| **Axis 0**  **Axis 1** | Vertical axis (rows)  Horizontal axis (column names) |
| **Select a column** | df[‘ADDRESS’]  Returns a series |
| **Select several columns** | df[[‘ADDRESS’, ‘CITY’]] |
| **Retrieve axis 1 / column names / information** | df.columns  Index([‘first\_name’, ‘last\_name’, ‘email’], dtype=‘object’) |
| **Retrieve axis 0 / vertical axis information** | df.index  RangeIndex(start=0, stop=3, step=1) or min 0, max 3 (3 rows) |
| **Retrieve a row** |  |
| **Make a column as an index** | E.g. you want to filter by last name. Instead of the default 0, 1, 2 etc. you can make the index as Nield, Scala Morrison.  *df.set\_index(‘column\_name’, inplace = True*) inplace is true so it edits the existing df instead of creating a new one  *df.loc[‘Scala’]* you can now use the last name to search for this row  **df.reset\_index(inplace = True)** reset the axis to default |
| **Copy a dataframe** | df2 = df.copy() |
| **New line \n** | Print(‘line1 \nline2’) |
| **Tab \t** | Print(‘line1 \tline2’) |
| **Raw string** | If you don’t want Python to see *\n* as code but part of the text (part of the string) then add an *r* add the beginning of the code.  Print(r’line1 \nline2’)  Output will not be  line 1  line 2  But line1 \nline2 |

**Importing Data**





[**Import CSV / read csv / load csv**](https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html)

Read csv as text:



Read csv into as a pandas dataframe:



**Read csv alternative:**

df = pd.read\_csv('https://raw.githubusercontent.com/thomasnield/machine-learning-demo-data/master/regression/winequality-red.csv')

df

**With *header* and *names*:**



**Alternative:**

Import pandas as pd

url = 'https://raw.githubusercontent.com/thomasnield/machine-learning-demo-data/master/timeseries/datetime\_formatting.csv'

df = pd.read\_csv(url)

df

**Format dates columns as dates while importing csv:**



**SQL** [**pd.read\_sql**](https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html)

Import an SQL database:



Alternative:



If you don’t *parse\_dates*:

With parsing and without:



**JSON**

Read a JSON file:



Read a JSON file as a pandas dataframe:



**Selecting Rows and Columns**

**loc and iloc**

* **loc** works on **labels** assigned to the axis, **iloc** works on **numbers**.
* E.g. if axis is surname then **loc** will work only on the surname like ‘**vanli’** while **iloc** will only work on **numbers**.

**Select** (first two) **rows** df.iloc[0:2] or df.iloc[:2]

**Exclude first row** df.iloc[1:]

**Select all** df.iloc[:]

**All rows and columns 2-3** df.iloc[:, 1:3] you are selecting columns 2 and 3 with indexes 1 and 2

**Select last two columns** df.iloc[:, -2:] count from 0 to -1, -2 etc. from right to left. Select the column you want to have and with : you will select everything to the right

**Select the last row** df.iloc[-1]

**Select all rows and last column** df.iloc[:, -1]

**Select all rows and all columns except for the last column** df.iloc[:, :-1]

If you want to get the first 2 rows you need to select the third index, in this case 2:



**Select specific columns and rows**. In this case you will select column index 0 and column index 2 (first\_name and email). In the second query you will select second row and third column



**Select rows and columns using loc** df.loc[["samiam","thomasnield"], "email"]

**Reset index** df.reset\_index(inplace = True)

**Select values that start with a specific letter** condition = df["username"].str.startswith("s") username = column, s = letter start

df[condition]

**Multiple conditions. AND & OR are & and |** condition = df["username"].str.startswith("s") & df["email"].str.contains("gmail")

df[condition]

**Get gata in a column that equals to a value** df[df[‘column’].eq(‘value’)] *(filter data to a value)*

[**at and iat**](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iat.html)

Similar to loc and iloc there is also an *at* and *iat*. These return a single value at a specific row and column index using numeric or labelled indices respectively.

**Drop columns and rows**





**Remove columns:**



**Remove columns by selecting specific columns like column 1 and 4:**



**Adding rows and columns (appending) / Joining concatting**

**Add a column at the end of the dataframe:**



**Add a column at a specific place:**



**Add a row:**



**Using concat (merge two datasets):**



**Merge two datasets using .concat()** In the example below the first *df* is the first dataset and *df.loc[0:2,:]* is the second.

Graphical user interface, text, application, email

Description automatically generated

**Updating data**

**Making a column in caps lock (upper):**



**Updating on a condition:**



**Update row on condition:**



**Unpivoting data (melting):**



Id\_vars = untouched columns

Value\_vars = columns that will be unpivoted

Var\_name = column names will be moved to this column

Value\_name = column values will be moved to this column



**Sorting, Casting, and Categories**

**Datatypes:**



**Timestamp example (date)** pd.Timestamp(‘20230130’) → 2023-01-30

**Difference between days** pd.Timestamp('20230130') - pd.Timestamp('20230127') → 3 days

**View datatypes** df.dtypes



You will get column name on the left (float, int, datatime etc.) and datatype on the right.

**Change a column to a different datatype** df[‘columnname’] = df[‘columnaname’].astype(‘bool’)

**Sort by 2 columns (first lightning, then rain inches)** df.sort\_values(by=["lightning","rain\_inches"])

df.sort\_values(by=["lightning","rain\_inches"],ascending=[False,True])

df.sort\_values(axis=0, ascending = True) *ascending if one value*

When using the sort methods, remember to **add the inplace=True parameter if you want to replace the existing dataframe** with the sorted one.

Sort by an row index or columns index df.sort\_index(axis = 0) for rows

df.sort\_index(axis = 1) for columns

**Replace a column with a *category* datatype:**

cat\_type = pd.CategoricalDtype(categories=["CLEAR", "MINOR", "MAJOR", "SEVERE"], ordered=True)

df["severity"] = df["severity"].astype(cat\_type) severity = column you want to replace

df.sort\_values(by=["severity"])

If you apply a categorization on a column that has values not mapping to any category, then those will become NA values.

**Python if/elif/else category function:**



**Apply this function to the wind\_speed\_mph column:**



**Add a new column:**



Categorize the last column and sort the data by that column in DESC:



**Removing Duplicative and Sparse Data**

**df used:**



**Get duplicated rows using .duplicated() function:**  df.duplicated()



It will mark rows that are duplicates.

If you want to see original rows and their duplicates (like in Excel) then add (keep=False):



**Look for duplicates in a specific column using *subset*:**



Or if you want to use multiple columns:



**Delete duplicates** df.drop\_duplicates(inplace=True)

**Delete duplicates based on a column** df.drop\_duplicates(subset=['record\_id'], inplace=True)

Number of unique values in a column df.nunique()



**Identify columns with single-values (e.g. value Shop in the whole column):**



**Drop these columns:** df.drop(delete\_cols, axis=1, inplace=True)

**Read csv file (open csv file) (using a link):**

wine\_df = pd.read\_csv('https://raw.githubusercontent.com/thomasnield/machine-learning-demo-data/master/regression/winequality-red.csv')

wine\_df

**Get number of rows and columns from .shape function:**



**Alternatively. Get number of rows ([0]) and columns ([1]) from the .shape function (where X = df):**



**Count the number of unique values per column:**



**Remove columns with 5% or less unique values:**



**Alternative using scikit-learn, VarianceThreshold and fit\_transform():**



Return from ndarray to get the columns using get\_support:



**Remove columns with duplicates and 3 or less unique values:**



**f-string / f’’ / f”” / printing with f / print(f:**



Example



**Before f’’:**

You can use f’’ like here:

Graphical user interface, text, application

Description automatically generated

**Or you can use %s / %d / %f / %r:**

Graphical user interface, text, application

Description automatically generated

%s is used as a placeholder for the value of name.title(), which is a string. When the print() statement is executed, Python replaces the %s placeholder with the value of name.title(), resulting in a string like "Eric's favorite numbers are:" or "Willie's favorite numbers are:".

The %s is just one of many string formatting operators available in Python, and is used specifically to insert **string** values into a string. Other operators include %d for inserting **integer** values, %f for inserting **floating**-point values, and %r for inserting **any Python** object as a string representation. In Python 3.6 and later versions, a new string formatting method was introduced using f-strings, which offer a more concise and readable way to format strings.

**Handling missing data**

**Looking for missing values**

**Looking for missing values per row** df.isna() *not efficient in my opinion, alternatives:* df.notna(), df.isnull(), df.notnull()



**Check columns for missing values** df.isna().any() *or you can specify axis*  any(axis=0)



**Check rows for missing values** df.isna().any(axis = 1)



**Select (show) columns with NaN (null) values** df.loc[:, df.isna().any()]



**Select (show) rows with NaN (null) values** df.loc[df.isna().any(axis=1), :] *or* df.loc[df.isna().any(axis=1)]



**Look for missing values in specific columns** df[df['TEMPERATURE'].isna() | df['RAIN'].isna()]



**Removing rows with missing values**

Note: many ML and statistical models do not tolerate NA, NaN or other missing null values.

**Remove rows with missing values / drop rows with missing values** df.dropna(axis=0, inplace=True)

**Remove rows with missing values in specific column** df.dropna(axis=0, subset=["RAIN"], inplace=True)

**Remove rows with missing values in specific columns** df.dropna(axis=0, subset=[“TEMPERATURE”, "RAIN"], inplace=True)

**Remove columns with missing values / drop columns with missing values** df.dropna(axis=1, inplace=True)

**Replacing missing values**

**Replace missing values with -1** df.fillna(value=-1, inplace=True)

You can’t specify a column using the subset parameter. To target specific columns you will need to extract them out and then apply fillna().

**NUMPY Replace values with NaN** from numpy import nandf.replace(-1, nan, inplace=True)



**SCIKIT LEARN Replace missing values with mean using SimpleImputer (imputer.fit, imputer.transform)**





There are other options for the *strategy* parameter including 'mean', 'median', 'most\_frequent', and 'constant'.

**SCIKIT LEARN Replace missing values with nearest neighbor using KNNImputer**







**Outliers**

We can use tools like *standard deviation (SD or* *σ [sigma])*  and *interquartile range*.

**Mean**

**Mean** mean = df.mean(axis=0)



**Standard deviation (SD, σ, sigma)**

**Standard deviation** sd = df.std(axis=0)

When calculating standard deviation with Pandas, it will be assumed to be a sample and therefore will calculate with 1 degree of freedom by default as shown in this formula:



To get a sense of how standard deviations play a role in omitting outliers, consider the graphic below. 1 standard deviation away from the mean (average) will capture 68% of the expected data points assuming a normal distribution. 2 standard deviations will capture 95%, and 3 standard deviations will capture 99.7%. With a standard deviation, The lower the standard deviation, the more aggressively outliers will be removed.



For smaller samples, cutting off at two standard deviations will be more common. This means we would declare any data on the tails outside those two standard deviations to be outliers and become candidate for removal.

Let's inspect the outliers outside two standard deviations. Multiply the standard deviation by 2 and subtract/add from the mean respectively to get the lower and upper bounds. Then we can compose a condition to identify the outliers by checking for weights less than or greater than these lower and upper bounds respectively.



**Remove outliers that fall outside the two standard deviations**

df = df[(lower < df[‘column’]) & (df['column'] < upper)]

df

**PANDAS NUMPY Remove outliers that fall outside 2.25 standard deviations:**

Don’t forget the (axis=0)**[0]** for the mean and std.



**Interquartile range outliers**

There is a lot of data that does not follow the nice bell curve shape of the normal distribution. Another way you can approach outliers in these cases is to use the Interquartile Range method, or IQR. This is the difference between the 75th and 25th percentile. When referring to the quarterly percentiles (0, 25, 50, 75, and 100). we refer to them as quartiles. A 50 percent quartile would be the middle-most value (the median), or the average of the two most-centered values.

Using the IQR, you will define a cutoff by a factor 𝑘 below or above the 25th and 75th percentile respectively. A common value for 𝑘 is 1.5, whereas a value of 3.0 would be used for more extreme cutoffs.

**Calculate percentile**

**NUMPY** In Python, we can use the **percentile() function** in NumPy to find a given percentile in a datastet.



**Get IQR:**





As you see above, the k value might be too generous for this dataset if we are looking to remove outliers. Maybe there are not extreme enough outliers in this dataset or this technique is just not warranted. But we can try to experiment lowering that k value to see how low the threshold must be before outliers removed. Below, I find a k value of 1.1 removes an outlier, with an index of 11 and weight of 54.

You can also use this technique on multidimensional data, by specifying an IQR policy for each field you want to target the removal of outliers.

**NUMPY Interquartile range outliers full walkthrough:**







[**LocalOutlierFactor**](https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.LocalOutlierFactor.html)

From a machine learning perspective, you can treat outliers as a classification. If they are far away from the rest of the datapoints in a multidimensional space, they can be detected as outliers. However, this becomes less reliable on higher dimensional problems due to curse of dimensionality. By leveraging logic that measures how far neighboring data points are, we can leverage the LocalOutlierFactor.

**SCIKIT LEARN Import LocalOutlierFactor** from sklearn.neighbors import LocalOutlierFactor







**Dates and times**

Check the data type of the date columns using *df.dtypes*. It might be *object* instead of *datetime64*. If a date-column has the right data type assigned then you can extract information from that column. Like day of the week and so on.

**Change data type of the date column (parse data)**  parsed\_col = pd.to\_datetime(df[‘column’])

parsed\_col

**Extract day of week** parsed\_col.dt.dayofweek

**Format dates columns as dates while importing csv:**



**Datetime conversion**

[Dataframe pandas conventions.](https://docs.python.org/3/library/datetime.html#strftime-and-strptime-behavior)

strftime() and strptime() are used to write a datetime to a formatting string, and parse a datetime from a formatted string respectively. The format codes come from the standard C conventions. Here are a few common ones, many of which we will use in this notebook. Refer to the link above to see all format codes.



**ORDER\_DATE\_TM2 column has values like 22-Jan-22 4:08 PM. Correct them to a normal date type:**







**Get all records where day of the week is Tuesday using dt.dayofweek**



**Filter dates**



**Or filter between a specific date and time (option 1)**



And option 2



**Timezones**

**Pytz Library for timezones**

import pytz

pytz.common\_timezones

**Look up a timezone in the pytz library**

tz = pytz.timezone('Europe/Amsterdam')

tz

<DstTzInfo 'Europe/Amsterdam' LMT+0:20:00 STD>

**Check if the column has timezone information using** [**.dt.tz**](https://pandas.pydata.org/docs/reference/api/pandas.Series.dt.tz.html)



**Assign a timezone to a column**



**Convert to a different timezone**



**Convert to UTC**



**Regular expressions (Python *re* library)**

[**re — Regular expression operations**](https://docs.python.org/3/library/re.html)

[**An introduction to regular expressions**](https://www.oreilly.com/content/an-introduction-to-regular-expressions/)

****

Regex quantifiers:

Table

Description automatically generated

[**From:**](https://regex101.com/)

 Graphical user interface, application

Description automatically generated Text

Description automatically generated with medium confidence



[From:](https://regexr.com/)

Graphical user interface, application

Description automatically generated

[Easiest way to remember Regular Expressions (Regex)](https://towardsdatascience.com/easiest-way-to-remember-regular-expressions-regex-178ba518bebd)

**Wrangling text / data cleaning**

These are the common string operations in Pandas we can use. Note that these typically accept a regular expression as a pattern, and we will cover this.

**Function Description *(functions in bold used below)***

count() Counts the number of instances in a pattern

**contains()** Returns a boolean True/False indicating whether a string contains a pattern

replace() Replaces the found patterns in a string with another specified string.

**fullmatch()** Determines if the entire string matches the pattern

split() Splits a string into separate strings using the pattern as the separator

extract() Finds all occurrences of a pattern and packages them into columns

**findall()** Finds all occurrences of a pattern and packages them into a list

**Fullmatch()**

Graphical user interface, application, Word

Description automatically generated

Same but without coloring:

Graphical user interface, text, application, email

Description automatically generated

**Look for a value in a column.** Look up the value *outlook.com* using the *str.contains()* function in a column. Step 1:

Graphical user interface, text, application

Description automatically generated

**Step 2.** Before showing the values you need to decide on the *NaN* values. If you want them to be treated as *not outlook.com* then add *na=False*. If you don’t add this code you’ll get an error.

Graphical user interface, application

Description automatically generated with medium confidence

**Look for a value in a column alternative.** Using str.fullmatch()

Graphical user interface, text, application, email

Description automatically generated

Show values that don’t match our condition using == False:

Graphical user interface, text, application

Description automatically generated

Only include rows in a dataframe that have a valid phone number and an IP address:

Table

Description automatically generated

Extract email domains from the email column using **str.findall()**



Gather unique domains from one column using str.join() and unique()

Graphical user interface, text, application

Description automatically generated

Show rows that have Eddy as first name:

Graphical user interface, text, application, email

Description automatically generated

Find unique values in a column:

Text

Description automatically generated

**Replace values / replace matched values**



**Splitting columns / splitting text into different columns**

Splitting emails into two columns using str.split

Graphical user interface, text, application, email

Description automatically generated

When you use regular expression features like look-aheads, it opens up more powerful splitting capabilities based on surrounding characters.

Split data in a column into 3 columns (split one column into 3 columns):

Input column:



Code: -, are the symbols used for splitting.

Graphical user interface, text, application

Description automatically generated

**Plot data / Visualize data**

**MATPLOTLIB**

Plot every row in the column weight using .hist()

Chart, histogram

Description automatically generated

**Recursion**

Graphical user interface, text, application

Description automatically generated

**Fibonacci sequence**

In mathematics, the **Fibonacci sequence is a sequence in which each number is the sum of the two preceding ones**. Numbers that are part of the Fibonacci sequence are known as Fibonacci numbers, commonly denoted Fn . The sequence commonly starts from 0 and 1, although some authors start the sequence from 1 and 1 or sometimes (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the first few values in the sequence are:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144.

**Generators (part of iterators / loops / iterations)**

Alternative to iter([1, 2, 3]) is a function with the yield keyword:

A screenshot of a computer

Description automatically generated with low confidence

We can only yield inside a function.

Putting a yield anywhere in a function makes it return generators. Generators are iterators with some more features that we don't need to care about.

Actually we don't even need to use iter() and next() most of the time, but I think it's nice to know how for loops work.

A screenshot of a computer

Description automatically generated with medium confidence

If we have other lines of code between yield then they will be extracted this way:

A screen shot of a computer code

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with low confidence

A screenshot of a computer

Description automatically generated with medium confidence

[**Intertools**](https://docs.python.org/3/library/itertools.html) **— functions creating iterators for efficient looping.**

For example, itertools.count(1) does the same thing as our count().

A screenshot of a computer

Description automatically generated with low confidence

**Iterable / iterator / for loops**

* An **iterable** is something that we can for loop over.
* An **iterator** is an iterable that remembers its position.
* For loops create an iterator of the iterable and call its \_\_next\_\_ method until it raises a StopIteration.
* Functions that contain yields return generators. Calling next() on a generator runs it to the next yield and gives us the value it yielded.
* [The itertools module](https://docs.python.org/3/library/itertools.html) contains many useful iterator-related things.

**List comprehensions**

List comprehensions are a **shorthand way** of creating and working with lists.