**Iterables**

**List**

**Tuple**

**Strings**

**zip() object**

**Sequences**

**enumerate() object**

**generator comprehension object**

**Iterators**

**Dictionaries**

**Sets**

**Function *-*** *a named bit of code that we can call(run) over and over*→ **sorted(object)**  
- function can be either built-in, or user-defined (UDF). User defined functions need to be defined (written) with **def** keyword before usage. Built-in functions can be used right away. There are a total of 66 of built-in functions in Python. See FUNCTIONS.

**Method** - *a function which belongs to an object and can change it.*→ **object.sort()  
-** methods are different from attributes by the means of their parentheses. Unlike functions, they can change the state of the object. Thus printing **object** after using **object.sort()** will no longer print the original **object**

**Parameter** - *placeholder for an argument specified in function definition*→ **range(start, stop, step)**  
- parameters can be required or optional. Optional parameters have default argument values. For example in **range()** **start** is optional with default value 0. It specifies position at which to start counting. **stop** is required, specifies position where to stop. **step** specifies how to increment. Default is by 1.

**Argument** - *value in parentheses passed to a function when calling it*→ **range(0,10,2)**  
- arguments can be positional or keyword-based (kwargs). Here, **(0,10,2**) are positional arguments filling in for **(start, stop, step)**. Kwargs are in fact arguments wit h displayed parameters **(start=0, stop=10, step=2)**.

Parameters with a **\*** in front of them can take multiple *positional* arguments, which are passed to the function as a tuple. Parameters with **\*\*** in front of them can take multiple *keyword* arguments which are passed to the function as a dictionary. See ARGS and KWARGS.

**Expression** - *piece of syntax that can be evaluated and produces a value*→ **2; 4+1; 'a'**; **'w'+4; true==false**; **sorted(); object.sort()**; **lambda x:x**  
- it's a combination of **variables**, **operators**, **functions** and **values** that give back a return value.

**Statement** - *an action or command to execute.*→ **a=1**; **if/else/elif**/**for**; **class; import; return; assert, raise, def**- statements for the most part consist of reserved keywords (see KEYWORDS). Main types of statements are assignment statement and conditional branching

**Calling** - *running a previously defined function or method.*→ **print(x)**- when function is called, it's with arguments (**x** above), which are essentially 'real' values passed into parameter placeholders of the function. See **Argument**

**Iteration** - *process of taking each item from something one after another*→ **x=['a', 'b', 'c']; print([i for i in x])**  
- iteration is also used for pointing at the actual instance of the loop. For example **for x in range(4)** consists of 4 iterations of the loop.

**Iterable** - *object that can return its members one at a time via looping*→ **[1, 'a']; 'Hello'; (2, 'b'); {'foo'}; {1:'one', 2:'two'}**- iterables are objects consist of sequences and iterators as well as dictionaries and sets (as these are not sequences). All of them can be used in a for-loop but only sequences can be accessed through indexing.

**Iterator** - *object, containing values that can be iterated upon*  
→ **map object at 0x001F12**, **generator object at 0x000001F9** (technically any object with a **\_\_next\_\_()** method)  
- iterators can't be indexed. Calling **print()** on an iterator results in *object id* like the one seen above. Its contents can only be displayed either via **for-loop**, or using a construction function like **list()**. Iterators utilize **iter()** and **\_\_next\_\_** methods returns the next value in the iteration and updates the state to point at another next value. When using for-loop (which isn't an infinite loop) it raises **StopIteration** exception when the **next()** method reaches the end of the iterable.

**Sequence** - *ordered collection of values in container defining their type*→  **'Hello'; [1, 'a']; (2, 'b')**- sequences support indexing. Strings are sequences as well, but sets and dictionaries are not sequences (dictionaries do support indexing, but not in a numeric way). Using **print()** on a sequence will display its contents.

**Generator** - *a function that returns an iterator*→ **range(0,2); *generator comprehensions***- the defining characteristic of a generator is that it uses **yield** keyword. Generator can be created out of custom made functions or comprehensions. They are slightly more taxing performance-wise than iterators, but they're easier to implement.

**Delimiter** - *one or more characters that separate text strings*→ **[a, b, c]; [a b c]; [a; b; c]; [a | b | c]**  
- Any character or a sequence of them can be made into delimiter using methods like **str.split()**. Delimiters that are shown above are commonly used in files and usually determine their type (see below).

**Delimited files** - *files with multiple fields per line a.k.a. tables*→ **file.csv; file.tsv; file.sql**- Type of a delimiter determines the type of delimited files. '**csv**' and '**tsv**'

mean 'comma separated values' (**,**)and 'tab separated values' () respectively. **sql** separates its values by colon (**;**).

**Interpreter** - *a program that reads and executes a given code*  
→ ***Python interpreter***, ***RE interpreter***, ***Math interpreter***- when using modules it's important that the default Python interpreter gets to read and execute the code first. Library interpreters like one from RE, or Math modules only get to read the code after Python one is done executing it. Because of that, in case there are metacharacters in the code, the other interpreters are likely to receive it in unreadable state. This can be remedied via backslashes or raw strings. See STRINGS 2

**Variable** - *a container for storing a data value.*→ **a=5; b='Tom'; x,y,z='one', 'two', 3; foo=[1,'bar']**- must start with a letter or underscore char. It can only contain alpha-numeric characters and underscores. Variables that are created outside of a function (all the above) are *global* and can be used by any function. To create a global variable inside a function, **global** keyword must be used.

**Metacharacter** - *a character with special meaning for the interpreter*  
**→ \, ^, $, \*, +, []**- used in conjunction with normal characters, they are used for defining patterns for searching in strings or text manipulation. They are similar in nature to **escape sequences.** See REGEX MODULE.

**Escape sequence** - *a sequence with special meaning for the interpreter*→ **\n; \b \s; \S; \d; \D**- they are combinations of the backslash character and a single following character that gives them both a single special meaning. See STRINGS 2

**Newline** - *escape sequence that marks the end of line and start of another*→ **print('Hello\nWorld!')**

- this escape sequence is equivalent to *Return* key - string in above example will be printed in two lines one for **'Hello'** and one for **'World'**. By default **print** function implicitly adds an invisible newline character at the end of a string. This can be changed by adding a keyword argument **end=''** to **print** function and using anything preferable as a separator.

**Constructor** - prepares the new object for use

**Attribute** - *a value which belongs to an object*  
→ **object.something\_defined\_by\_yourself**- attributes are essentially just names bound to initial object

**Immutables** - *unmodifiable after created.*→ **(a,1); {birth:1988}; 1; 1.5; TRUE; 'Hello'**- consist of tuples, strings, keys in dictionaries, ints, floats and booleans.

A set of 33 reserved words in Python that define the structure of the Python language. Keywords cannot be used as names for variables, functions or other identifiers. The following list excludes the more prominent ones such as logical and boolean operators:

**as** - Creates an alias.

**assert** - Tests for debugging if a condition in your code returns True.

**break** - Stops a loop from continuing; see FOR-LOOPS

**class** - Defines a class.

**continue** - Continues to the next iteration of a loop; see FOR-LOOPS

**def** - Defines a function.

**del** - Deletes an object.

**elif** - Conditional statement; see FOR-LOOPS.

**else** - Conditional statement; see FOR-LOOPS.

**except** - Used with exceptions, what to do when an exception occurs

**for** - Creates a for-loop; see FOR-LOOPS

**from** - Defines source of an imported module; see IMPORTING DATA

**global** - Declares a global variable. Used inside functions or classes

**if** - Creates a conditional statement; see FOR-LOOPS

**import** - Imports a module; requires specification of source through **from**

**in** - Checks if a value is present in a list, tuple, etc.

**is** - Tests if two variables are equal.

**lambda** - Creates an anonymous function; see LAMBDA FUNCTION

**None** - Represents a null value.

**nonlocal** - Declares a non-local variable.

**pass** - A statement that will do nothing. Used as a placeholder.

**raise** - Raises an exception.

**return** - Exits the function and returns a value.

**try** - Creates a try...except statement. See **except**

**while** - Creates a while loop.

**with** - Used to simplify exception handling; see IMPORTING DATA

|  |  |  |
| --- | --- | --- |
| Integer | **3** | **int()** |
| Float | **2.24** | **float()** |
| Complex | **1+2i** | **complex()** |
| \*String | '**one'** | **'', str()** |
| \*Tuple | **(1, 'one')** | **(,), tuple()** |
| \*List | **[1, 'one']** | **[], list()** |
| Set | **{1, 'one'}** | **{}, set()** |
| \*Dictionary | **{1:'first', 'one':2}** | **{x:y}, dict([x:y])** |

**Integers** (whole numbers), **floats** and **complex** numbers falls under **numbers** category. Numbers are strictly single and numeric objects. **Numbers**, **strings**, and **tuples** are **immutable** data types, meaning they cannot be modified inside themselves (actually they can, but this involves creating a new ). Lists, sets, and dictionaries are mutable. Lists, tuples, strings, sets and dicts are iterables and at the same time lists, tuples and strings are also sequences.

**Strings** are enclosed in quotation marks. They can be either text or numeric based. Their elements - characters - can be accessed through index - for example element [**0**] in string **'Hello'** is **'H'**. See STRINGS.

**Lists** are collections of objects (or list items) that are ordered through indexing and changeable. They allow duplicate members. See LISTS**.**

**Tuples** are ordered through indexing as well, but their elements are unchangeable. Also allow duplicate members. See TUPLES.

**Sets** are unordered lists where items have random order (therefore sets have no indexes). They don't allow duplicate members.

**Dictionaries** are special kind of lists with key-value pairs, where keys fill the role of indexes. No duplicate members. See DICTIONARIES

Indexing means referring to an element of a sequence (**i.e. string, tuple or list**) by its position within that sequence. This is done via appending square brackets with a number inside after the sequence itself or its name (note that lists are defined by square brackets [] too, which can be confusing). For instance **example[1]** equals **'0.5'** in the case below. Indexes can be positive or negative.Positive indexes start with 0:

**positive index 0 1 2 3 4 5 6 example = ['a', 0.5, ('x','y'), 'f', [1, 2], 10, 'g'] #  
negative index -7 -6 -5 -4 -3 -2 -1**

Negative indexes yield the sequence elements from the end of the sequence i.e. opposite side. **example[-3]** equals **[1, 2]**#. Note that for negative indexes doesn't apply that the first element's index is equal to **0**.

## Slicing

Slicing is an advanced form of indexing, which adds a second argument to the index operator. This second argument is called *stop* position and marks the index of an item to stop at without including it in the slice. Thus **[2:4]** equals just indexes **[2]** and **[3]**, not **[4]**.

**mylist[2:4] → ('x','y'), 'f'**

*Start* position index must always be on the **left** side of the *stop* position within the iterable, otherwise empty list is returned. **[4:2]**, **[-3:-5], [-2:3], [1:-6]** all return empty in case of the table above, as their start positions (**4, -3, -2** and **1**) all happen to be on the right side**.** Leaving either side of the colon blank means start from (or go to) the end of the list. For negative index the rule of not including the *stop* position applies too:

**mylist[:4]='a', 0.5, ('x', 'y'), 'f' mylist[4:]=[1, 2], 10, 'g'  
mylist[:-4]='a', 0.5, ('x', 'y') mylist[-4:]='f', [1, 2], 10, 'g'**

Positive and negative indexes can be combined, but special attention needs to be given to actual size of the sequence:

**mylist[1:-1]=0.5, ('x', 'y'), 'f', [1, 2], 10 mylist[2:-4]=('x', 'y')  
mylist[-4:2]=[]** *-4 is on the right of 2 so it returns an empty list. This might not always be the case - it depends on the size of the sequence*

## Stepping

Advanced form of slicing; adds third argument, **step**, setting the interval at which elements are being counted in the slice (default being **[::1]**)#. Negative step values reverse the direction in which the slicer iterates, which also means *start* position now needs to be on the right of *stop* one:

**mylist[::2]='a', ('x', 'y'), [1, 2], 'g' mylist[::-2]='g', [1, 2], ('x', 'y'), 'a'  
mylist[1:4:2]=0.5, 'f' mylist[-2:-4:-1]='f', 'e'*(-2 on the right of -4***

## Nested indexes

With the exception of strings, sequences can also contain another, nested sequences which utilize indexes as well. These can be addressed via appending another index after the one referencing that nested sequence:

**example = ['a', 0.5, ('x','y'), 'f', [1, 2], 10, 'g']  
example[2][1] → 'y'**

**example[2]** refers to the tuple **('x', 'y')**. Then **[1]** to the **'y'** of (**'x', 'y')**

## Indexing in strings

Strings are a special kind of sequences, that can contain only characters - which are in fact one-element strings themselves. Strings are still sequences though so indexing applies here as well:

**print('teststring'[3]) → 't'**

## Using indexes with functions/methods

Indexes are not restricted to the objects in their prime state: they can be placed behind functions or methods and affect their modified state unless that method or function returns **None** like .**append()** for example.

**example = 'Martin 19'  
1.) print(sorted(example))[2]) → '9'  
2.) print(example.split(' ')[2]) → '19'**

Example #1 prints '**9**' because **sorted()** sorted the string with order being whitespaces - numbers - letters. In #2 whitespaces are removed and numbers and letters split. '**19'** thus becomes standalone indexable object.

## Using indexes in for-loops/comprehensions

Indexes in for-loops/comprehensions allow to perform operations over elements inside iterables which are themselves part of another iterables over which the for-loops/comprehensions are being performed:

**example = [(1, 2), (3, 4), (5, 6)]  
print([i[1] for i in example]) → [2, 4, 6]**

Here indexing was used to pick all the second elements from the tuples inside the list. **i** represents one element from the list - i.e. the tuple. The tuple is then indexed to specify the second element. An alternative to this process is multiple assignment (see MULTIPLE ASSIGNMENTS), which instead of accesing the object as a whole and then picking it's element via index, access the elements themselves and picks the name assigned to the desired element:

**print([j for i, j in example]) → [2, 4, 6]**

Generally if there are more than 3 it makes better sense to use indexes#.

## String definition

String is an iterable with Unicode characters as its elements. It's created by enclosing characters in quotes **''** (doesn't matter if they're single **''** or double ones **""**). Tripling these quotes will make the string multiline (it will be able to be printed on multiple lines depending on its input):

**print('''Word → Word  
wrap''') wrap**

## Strings as iterables

Even if they don't look like ones (with no brackets/parentheses although quotes could be filling that role) strings are iterables with their elements being their characters. Since characters are not an object type in Python though, these elements are the individual one-character strings instead. Most of the things that apply to other iterables apply to string as well. With the sole exception that all its elements must consist of these one-character strings. I.e. if a + operation (concatenating new object to the string) is performed, that object must be a string as well:

**print('example' + 34) → #error#  
print('example' + str(34)) → example34**

### For-loops with strings

As mentioned before strings are basically lists of characters, aka little strings. Thus for-loop on a string will iterate through these little strings:

**for i in 'example':  
 print(i) → 'e'; 'x'; 'a'; 'm'; 'p';'l'; 'e'**#

Thanks to this interaction, for-loops can be used when searching for certain characters or expressions in text, like in example on next page:

**example = 'the rocket came back from mars'**

**vowels = [i for i in example if i in 'aeiou']**

**print(vowels) → ['e', 'o', 'e', 'a', 'e', 'a', 'o', 'a']**

As individual characters are treated similarly to list items, Python will also search each of them individually. Thus when presented with a string as iterable, it will iterate over individual characters, checking for all of them in the **sentence**. See ADVANCED EXAMPLESfor more.

**for i in range(0, len('example'), 2):**

**odd\_letter += 'example'[i] → print(odd\_letter) → eape**

For-loop has been used to modify - populate string with letters. String is immutable - doesn't allow operations like update, delete, or append. But string can still be modified via creation of a new string and naming it the same. **+=** does exactly that. See MATHEMATICAL OPERATIONS.

## String indexing

Strings, being sequences, allow indexing as well as slicing and stepping:

**'word'[1] = 'a' → #error#**

**'word'[2] → r 'word'[::-1] → drow**

**example="Hello, World!"   
print(example[2:5]) → llo**

Indexing for strings can also supplement modifying strings in a way:

**word = 'Python is cool'  
word = word[0:3:-1]+'h'+ word[11:13] + 'n' + word[6:]**

**'Python'** was changed to **'tyPhoon'** by reversing the step on first three letters, adding **'h'**, **'oo'** from the word **'cool'** and **'n'**. Rest is the same.

## String methods

Some methods require deeper understanding to combine them with specific keywords like **in**. See ADVANCED EXAMPLES for more:

**def check\_for\_name(sentence, name):**

**return name.lower() in sentence.lower()**

**lower()** brings all characters to lowercase. In order to search **sentence** for the presence of **name** in any form (mixed upper/lowercase like '**pYtHoN**') all characters are brought to lowercase prior to search.

## String interpolation (inserting code)

|  |  |  |
| --- | --- | --- |
| **a='interpolation'** | **b='example'** |  |
| **print('This is NOT an ', b,' of string ', a)** | | **No interpolation** |
| **print('This is an %s of string %s'%(b, a))** | | **% format** |
| **print('This is an {} of string {}'.format(b, a))** | | **sys.format** |
| **print(f'This is an {b} of string {a}')** | | **f-string#** |
| **from string import Template c=Template('This is an $x of string $y') print(c.substitute(x=b, y=a)** | | **string.Template (requires assigning variables to strings)** |

Interpolation means substituting values of variables into placeholders inside a string. It's useful when there's need to put a variable into the string without cutting it in two. This can be done in four ways: **%-format, sys.format(), f-string** and **string.Template**:

**f-string**s interpolate right inside the string. So if the interpolated code contains another string, an error occurs. It doesn't happen in example above, since variable name was used. **f'This is an {'interpolation'} …'** would not work. Use double quotes for the string itself and single quotes for the interpolated string in code. See PRACTICAL EXAMPLES

## Escape sequences

Strings may contain characters that will cause an error, or make Python behave unexpectedly because they will mess with how it's supposed to interpret them. Good example of that is quotation mark (**'**):

1.) **print('Python says 'Hello, World!'')**2.) **print('Python says** **\'Hello, World!\'')**

Script #1 causes an error because quotation mark is interpreted in Python script as a string boundary. Adding a backslash character (**\**) in front of the mark *escapes* it (makes Python treat it as part of the string).  
*Escape sequences* are combinations of the backslash character and a single following character that gives them both a single special meaning:

**\’ print('Say \'hi\'!') → 'Say 'hi'!'**

**\\ print('C:\\Python\\name') → 'C:\Python\name'**

**\n print('Ends\nline') → 'Ends  
 line'**

**\t print('Hello\tWorld!') → 'Hello World!'**

**\b print('P\bython') → 'yton'**

## Raw strings (R-strings)

These are used to prevent Python parser from interpreting backslash (**\**) and by extension all escape sequences in strings as metacharacters. Double backslash has similar effect, but requires individual use for each case. They're especially useful in cases where external modules like **RE** are used, since they have their own parsers, which only get to read the provided strings after Python one interprets them#:

**print(r'C:\Python\name') → 'C:\Python\name'**

## String functions

**len()** - Returns number of characters in the string  
**max()** - Returns the highest character. These are sorted by lowercase and uppercase alphabet respectively, with capital **Z** being the highest.  
**min()** - Ditto as for **max()**. Lowest character is lowercase **a**.  
**sorted()** - Returns list with single characters, sorted by rule like **max()**

## String methods

**.capitalize()** - Converts first character to upper case  
**.title()** - Converts the first character of each word to upper case  
**.casefold()** - Converts the string into lowercase. Stronger than **lower()**  
**.count()** - Returns the number of times a given value occurs in a string  
**.endswith()** - Returns *true* if the string ends with the specified value  
**.find()** - Searches the string for a value and returns its position in string  
**.index()** - Ditto as above; raises an exception if the value isn't found **.format()** - Formats specified values in a string and insert them inside a string's placeholder, which is defined using curly brackets **{}**. Adding certain symbols in can adjust the alignment, format, or add separators.  
**.islower()** - Returns *true* if all characters in the string are lower case **.isnumeric()** - Returns *true* if all characters in the string are numeric **.join()** - Adds elements of iterable to the end of a string or its characters **.lower()** - Converts the whole string into lower case **.lstrip()** - Removes all spaces or punctuation from the left side of string **.replace()** - Replaced a specified value with another specified one **.rfind()** / **.rindex()** - Searches the string for a value and its last position **.rstrip()** - Removes all spaces/punctuation from the right side of string **.split()** - Splits the string at the specified value. Returns a list of strings **.swapcase()** - Swaps cases, lower case becomes upper and vice versa **.upper()** - Converts the whole string into upper case

## List definition

List is a collection of objects - a sequence, that is ordered (i.e. it can utilize indexes) and changeable. Its elements may be in various data types including another lists, tuples or dictionaries. Lists are written in Python with square brackets **[]** (this can be confused with indexes which use the same type of brackets).

**newlist=['a','e','i','o']** - list of 5 strings.  
**biglist=['a', 2, 3.5, [0,1], ('b','c'), {6,7}, {x:y}] -** list of all data types

List constructor/converter is **list()**:  
**list(('a','e','i','o'))** - converts/creates a list from a tuple

## Mutable lists

Mutability of the list means that an instance of it can be modified at will. This effectively just means that it can utilize several methods, that are only available to it and no other object type#. Immutable objects can be actually 'modified too via concatenation (adding an object to another object of a same type), although that will create a new instance of that object. This new instance can rewrite the old instance though; that's what += is for).

## Creating a list using automated methods

Besides creating a list and populating it manually or using a constructor, there are following methods, which can create lists in automated fashion:

### For-loops

Can create lists upon instantiating an empty list and then looping over an iterable or a range of elements and appending each of those to that list. Function below appends squares of numbers 1-10 to **squares** list:

**for i in range(10): squares = []**

**squares.append(i\*i) → [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]**

### Map() function

Creates map objects, by passing a supplied function and an iterable. It works like batch running of a function. See MAP()/FILTER()

**def get\_price\_with\_tax(i): tax\_rate = .08**

**return i\*(1+tax\_rate)  
input = [1.09, 23.56, 57.84, 4.56, 6.78]**

**final\_prices=list(map(get\_price\_with\_tax, input))**

In this example an input list and a tax rate is defined along with a function that uses the **tax\_rate** value. **map()** function then proceeds to pass all the input values through the **get\_price\_with\_tax** function. Note that the **map()** object needs to be wrapped in some constructor function, which can be **list()** like in the example above, **set()** or **tuple()**.

### List comprehension

List comprehensions define the list and via for-loop on one line it's contents. The output is always a new list. See COMPREHENSIONS.

**squares = [i \* i for i in range(10)]**

## Nested objects in lists

Lists can contain any type of object, including another lists, tuples, sets, or dictionaries. These objects are considered as a single item (that is without considering their own elements). This is important in the context of indexes (see INDEXING) and/or for-loops (see FOR-LOOPS), that both need to utilize another instance of themselves in order to address the elements inside the nested sequences:

**example = [1, 2, [3, 4]] 1.) example[2][1] → 4**

**2.) for i in example:   
 for j in i:  
 print(j) → 3, 4**

In example **#1** two indexes were used to address the second element inside the nested list. First index **[2]** addressed the third element of the **example** list (that being the nested list) and then the second index **[1]** addressed the second element of that nested list.See INDEXING.  
In **#2** two for-loops (one nested inside the other) were utilized to print the elements of the nested list. Note that the first loop iterated over *all* elements of the **example** list (including the ints **1, 2**) but **print()** is in the second loop, which only found elements in the last element - nested list.

## Modifying lists

Mutability of the lists allows for *item assignment* - changing element in the list via addressing it via index and then assigning it a new object:

**example[2] = 5 → print(example) → [1, 2, 5]  
example[-1] = [3, 4] → print(example) → [1, 2, [3, 4]]**

## Extennding lists

Item assignment cannot extend lists though. Following methods can. Note, that **insert()**, **.append()** and **extend()** are unique to the list:

**mylist = [1, '23', 'Hi', 2.6, ['abc', 2]]  
1.) mylist + [True, 2] → [1, '23', 'Hi', 2.6, ['abc', 2], True, 2]  
2.) mylist.insert(2, True) → [1, '23', True, 'Hi', 2.6, ['abc', 2]]**. **3.) mylist.append({2: 1}) → [1, '23', 'Hi', 2.6, ['abc', 2], {2: 1}]  
4.) mylist.extend({2: 1}) → [[1, '23', 'Hi', 2.6, ['abc', 2], 2]**

**1.)** A simple *concatenation#* i.e. adding another list to an existing one will create a new instance of the list that consists of both the original and the added one. This new instance can overwrite the instance of the original list if desired though - that's what += is for. Operation like this can actually be done with any object type (sets and dicts use union | instead though), including tuple with another tuple for example. **2.)** .**insert()** method inserts a specified object at a given position (**2** in the example above) in the list instance. It can be anything from int to dict. 3.) **.append()** is same as **insert()** but it always adds to the end of the list.  
**4.)** **.extend()** adds elements from a given sequence to the list instance. Note that if the given sequence happen to be a dictionary, it will only add the keys.

List functions

**len(list)** - Returns number of elements in the list **max(list)** - Returns largest element of the list, or if two or more lists are provided, it returns the largest list.  **min(list)** - Returns smallest element of the list. Otherwise ditto as **max()  
sorted()** - Returns a list with the elements in sorted manner

## List methods

**.append()** - Adds an element to the end of the list. Unique to the list **.extend()** - Adds elements from another iterable to the end of the list **.insert()** - Adds an element to the specified position **.pop()** - Removes an element at specified position and returns that value **remove()** - Removes an item with the specified value  
**.count()** - Returns the number of elements with a specified value **.sort()** - Sorts the list. Can be specified by key or by reversing order  
**.index()** - Returns index of the first element with a specified value

## Dictionary definition

Dictionaries are special kind of iterables with key/value pairs. They can only contain dictionary entries: keys and values. Keys in dictionaries substitute number indexing in lists. As such they must be unique as well as of immutable object type (ints, Booleans, strings, floats and even tuples). Values on the other hand can be of any object type including another dict; they can also repeat as well as be identical to keys:

**example = {True: {1, 2}, (1, 1): [2, 3], 0.5: 0.5, 'Hi': {5: 6, 7: 8}}**

## Creating dictionaries

Dictionaries can be created with the use of the following 4 ways:

**1.) {1:'apple', 'two':[2,4]}** - using curly brackets  
**2.) dict(one='apple', two=[2,4])** - using **dict()** function#  
**3.) dict([(1,'apple'), ('two',[2,4])])** - using **dict()**+ list of key/value pairs  
**mylist=[1, 'two']  
4.) example2.fromkeys(mylist, 1) → {1:1, 'two':1}**

**.fromkeys()** method creates keys out of elements of an iterable. First parameter specifies this iterable and the second assigns all of them a specified value (int **1** in the example above).

## Dictionary indexing

Dictionaries are indexed through their keys. Identified is probably a better word though, because even though dict indexes look similarly to sequence ones, they're different. Numeric indexes provide sorting of the sequence elements with their number series. With keys this is obviously impossible as well slicing and stepping. There is also no range to the dict  
indexes. Using an index that is out of range in a sequence will result in

an error but in dictionary it will create a new entry instead (see EXTENDING DICTIONARIES - MODIFYING/ADDING ITEMS).

**print(example['Hi']) → {5: 6, 7: 8}**

Dictionary **example** from the previous page contains a key **'Hi'**. Using **'Hi'** as index for **example** yields a value of that key, i.e. dict **{5: 6, 7: 8}**

### Indexing in nested dictionaries/tuples/lists

Similarly to numeric indexing, addressing nested element is a matter of appending another index behind the one addressing the sequence with desired nested element. In example below value **8** from the nested dictionary in dictionary **example** from the previous page is addressed:

**print(example['Hi'][7]) → 8**

## Extending dictionaries - Modifying/Adding items

Dictionaries can only be extended with another dictionaries i.e. key: value pairs (a value can be of any object type though). This can be done by **.update()** method, **key indexes method** or .**setdefault()**.

### Via .update() method

**.update()** is used less often. It's useful for concatenating two dicts for example, It's also the only way of adding an entry with actual key: value format. See PRACTICAL EXAMPLES:

**2.) example2.update({10:4, (2, 0.5): (2, 0.5)})  
print(example2) → {1:1, 'two': 1, 10: 4, (2, 0.5): (2, 0.5)}**

Via key indexes   
As mentioned dicts are indexed with their keys instead of numbers so they're unordered. This makes it possible to use indexes as vessels for

new entries. Assigning the non-existant key a value will result in a new entry in a dictionary. Key **42** in example below is not in **example2** on the previous page, but if it gets assigned a value like tuple **(5, 10)**, new entry will be created for **example2**. If the given key is in the dictionary already, assigning a new value to it rewrites the previous one it had, just like assigning a value to a index in list would.

**1.) example2[42]=(5, 10)**  
**print(example2) → {1: 1, 'two': 1, 42: (5, 10)}  
example2['two']=3  
print(example2) → {1: 1, 'two': 3, 42: (5, 10)}**

### Via setdefault() method

Finally **.setdefault()** works by 'forcing' a key and a value in a dict if it isn't there already. It takes two parameters: a **key** to search for; if the key isn't found, it's added into the dict along with the second param - **value**.

**3.) print(example2.setdefault(10, 1)) → 4  
 print(example2.setdefault(11, 1)) → 1  
print(example2) → {1:1, 'two': 1, 10: 4, (2, 0.5): (2, 0.5), 11: 1}**

## Get() method

Like **.setdefault()**, **get()** takes two params. **key** to search for and a **value**. If the given key doesn't exist in the dictionary **.gets()** is used on, it just returns a given value without creating a new entry:

**example2.get(44, 'no key') → no key  
example2.get(42, 'no key') → (5, 10)**

## For-loops with dictionaries + using dict method

When using a for-loop with a dictionary its important to know that it returns its **keys** (method **dict.keys** returns the same). I.e. for dealing with

dictionaries using for-loops, **.values()** method, that returns values or even better **.items()** which returns both keys and values should be used most of the time, as it's the most flexible way to access dict elements:

**print([i for i in example2]) → [1, 'two', 10, (2, 0.5), 11]  
print([i for i in example2.items()]) ↓** [**1:1, 'two': 1, 10: 4, (2, 0.5): (2, 0.5), 11: 1]**

Using **.items()** is recommended as keys and values can be easily filtered out via either multiple assignment (where only one of the elements is specified for the output) or alternatively indexes:

**print([key for key, value in example2.items()])  
print([i[1] for i in example2.items()])**

Values themselves can also be addressed via the key index:

**print([example2[i] for i in example2]) → [1, 1, 4, (2, 0.5, 1]**

### Adressing nested dictionaries

In case a value of dictionary is another dictionary, it has to be noted again that looping over a dict returns keys. Thus using **dict.values()** will just return keys from the nested dict. In order to actually address nested values the aforementioned key index method needs to be utilized:

**example3 = {'person': {'name':'Martin', 'month':3, 'year':1988}}  
print([x for x in example3.values()] → ['name', 'month', 'year']  
print([y['month'], y['year'] for x, y in example3.items()]) → [3, 1988]**

### Creating new dictionaries with for-loops

As mentioned, key indexes as well as **.update()** and **.setdefault()** can create new entries in a dict. Thus aside from addressing dict values they can also extend it or create a new one:

**for key, value in example2.items(): new\_dict = {}**

**new\_dict[value] = key   
for key, value in example2.items():  
 new\_dict.setdefault(value, key)***both for-loops return* **→** [**1: 11, 4: 10, (2, 0.5): (2, 0.5)]**

In this example both methods ended up filling the **new\_dict** with keys made up from values from **example3** and values from keys. There is a major caveat though. As mentioned earlier keys in the dicts are unique, so if one key index is assigned a value multiple times, all the previous values are lost. **example2** had keys **1:1, 'two': 1** and **11:1**. Only the last one had been 'converted'; previous two were rewritten. In such case a list needs to be provided in order to hold all the values:

**for key, value in example2.items():  
 new\_dict.setdefault(value, []). append(key)  
for key, value in example2.items():  
 new\_dict[value] = new\_dict.get(value, []) + [value]**

**.setdefault()** is specifically for cases like this. It's possible to get similar result using key indexes with **.get()** method, but since **get()** would be a value for the key index, we wouldn't be able to utilize **.append()** since object with **append()** method always returns **None**#.   
Using **get()** for creating entries via index method is more viable when dealing with immutables (that can't use **.append()** anyway). There is another problem though: the new entry does not 'register' until the key is assigned a value. Both **.get()** and **.setdefault()** set their behavior off condition of finding a key, so this has major consequences:

**counts = {} sentence = 'Python Python'**

**for word in sentence.split():**

**counts[word] = counts.get(word, 0) + 1**

This loop counts words in the **sentence**. Each word that isn't already in the dict creates a key in the **counts** dict. This key then gets the value **0** (since without a value assigned it doesn't exist yet), and **1** gets added to it. Each subsequent appearance of this word makes **.get()** return the value of the key, which now exists, and each time **1** gets added to this value. Result is **counts = {'Python': 2}**. See PRACTICAL EXAMPLES for more use cases.

### Deleting entries with Pop() method

See PRACTICAL EXAMPLES for use cases

## Dictionary functions

**all()** - Returns TRUE if all keys in the dictionary are not 0.  **any()** - Returns TRUE there's a key in the dictionary, that isn't 0  
**len()** - Returns number of items (key/value pairs) in the dictionary **sorted()** - Sorts dictionary. Can be specified by key or reversing order

## Dictionary methods

**.fromkeys()** - Returns a dictionary with the specified keys and value

**.update()** - Updates the dictionary with the specified key/value pairs  
**.pop()** - Removes an element with the specified key and returns its value

**.popitem()** - Removes the last inserted key value pair

**.items()** - Returns a list containing a tuple for each key value pair **.get()** - Returns value of specified key Or if its not there, specified value  
**.keys()** - Returns a list containing the dictionary's keys

**.values()** - Returns a list of all the values in the dictionary.

**.clear()** - Removes all elements from the dictionary

**.copy()** - Returns a copy of the dictionary  
**.setdefault()** - Returns the value of the specified key. If the key does not exist: insert the key with the specified value

## Tuple definition

A tuple is a collection of items that is ordered and **unchangeable**. The items may be in various data types including another nested lists, tuples or dictionaries. Tuples are generally used for different data types and lists for similar ones. Tuples are written with parentheses **()**.

**bigtuple=('a', 2, 3.5, [0,1], ('b','c'), {6,7}, {x:y})** - different data types

Tuple constructor/converter is **tuple()**:  
**tuple(['a','e','i','o'])** - converting/creating a tuple from the list

## Editing tuples - Adding items

Since tuples are immutable, all methods for editing are non-functional. Adding items can be done by concatenating a tuple with another tuple:

**tuple1 = (1,2) tuple2=(3,4)  
tuple1+tuple2 = (1,2,3,4)** - result contains combined items

## Packing & Unpacking tuples

Tuples can be assigned both a single name of the whole tuple and multiple names based on the number of items in the tuple. This process is called **multiple assignment** (see MULTIPLE ASSIGNMENTS for more details), or **tuple packing/unpacking**, since it's frequently used with tuples. It can be utilized with any iterable though**.** In case both the name for the whole iterable and names individually assigned to the items are used, all of them persist:

**example = 'one', 2, [0.5, 'k']  
first, second, third = example  
print(second, first, example, sep=',,,') → 2,,,'one',,, 'one', 2, [0.5, 'k']**

Note that the three objects, created in the first row are considered a tuple even though there are no parentheses. Objects separated by commas without specified type (like being enclosed in brackets for example) are implicitly considered a tuple of items. Printing **second** will yield int **2**.

Deep unpacking

If one of the items is a collection of another items, then in order to reach the items inside that collection, another tuple can be added in, mimicking the state from the right side of the '**=**'. Note that left side of the **'='** in the example below actually looks like this: **(color, (x, y, z))**:

**color, (x, y, z) = ("red", (1, 2, 3))**

Following is another example of deep unpacking:

**points = ((1, 2), (-1, -2)) (x1, y1), (x2, y2) = points**

**print(x1 == -x2 and y1 == -y2) → True**

## Tuple functions

**all()** - Returns true if all keys are true (aren't None) or if tuple is empty

**any()** - Returns true if any key of the tuple is true. Empty = False.

**len()** - Returns length of the tuple or size of the tuple

**max()** - Returns the largest element of a given tuple or if two or more tuples are provided, it returns the largest tuple.

**min()** - Returns smallest element of given tuple. Ditto as **max()**

**sorted()** - input elements in the tuple and return a new sorted list

## Tuple methods

**.count()** - Returns the number of times a value appears in a tuple.  
**.index()** - Searches a given value in the tuple and returns its position

## Multiple assignment definition

Every iterable can be assigned a single name as a whole or multiple names as its individual elements (as long as there are all of them; assigning more or less will result in error). This second approach is called multiple assignment, also known as iterable unpacking or tuple unpacking since it's frequently used with tuples:

**mylist = (1, 2, 3, 4) first, second, third, fourth = (1, 2, 3, 4)  
print(mylist[2]) → 3 print(third) → 3  
mystring = 'Hi!' one, two, three = 'Hi!'  
print(mystring[1]) → 'i' print(two) → 'i'**

As seen above multiple assignments are an alternative to indexes. Getting a single element out of an iterable means either accessing it as a whole with an index, or through its parts and just use the desirable one. For an iterable with less than 5 elements it's beneficial to use multiple assignments instead of indexes for the reason of code readability:

**def format\_date(mdy\_str): def format\_date(mdy\_str):**

**x = mdy\_str.split('/') m, d, y = mdy\_str.split('/')**

**return f'{x[2]}-{x[0]}-{x[1]}' return f'{y}-{m}-{d}'**

Python implicitly considers every couple of objects or names (like **10, 20** or **first, second**), that are not enclosed in standard or curly brackets a tuple. All examples shown below are thus identical:

**first, second = 10, 20 (first, second) (10, 20)  
first, second = (10, 20) (first, second) = 10, 20**

Following are examples of multiple assignments with other iterables:

**x, y, z = [10, 20, 30] x, y, z = 'Man' x, y = {'day': 1, 'month': 2}  
print(y, z) → 20, 30 print(y)→ 'a' print(y) → 'month'**

## Usage with for-loops

For-loops is where multiple assignments shine the most; especially when used in conjunction with dictionaries or functions that produce two or more items per result such as **zip()** or **enumerate()**:

**list1 = ['a1', 'a2', 'a3']; list2 = ['b1', 'b2', 'b3']; list3 = ['c1', 'c2', 'c3']  
for a, b, c in zip(list1, list2, list3): for i, a in enumerate(list1):  
 print(a, b, c) print(i, a)**

**for i, (a, b) in enumerate(zip(list1, list2)):**

**print (i, a, b)**

Since dicts have always at least two outputs (keys and values) multiple assignments are very useful when manipulating with their elements. Especially when the values of the dicts are iterables themselves:

**example = {'a':(2, 5), 'b':(4, 10), 'c':(1, 3)}  
for key, value1, value2 in example.items():**

**print(f"Key {key} has a tuple of values {value1, value2}")**

## Usage with comprehensions

**table\_feed = [('Ann', 100), ('Joe', 200), ('Dac', 300), ('BJ', 400)]  
reversed\_columns = [(j, i) for i, j in zip(column1, column2)]  
reversed\_columns.sort()**

This example simulates the output of a table, consisting of list of tuples. Its 'columns', which are all the items in tuples with index 0 for first column or 1 for second, were switched via list comprehension. After that **.sort()** could be used to sort items by values. Note the parentheses around the first variable specification. It's the only case where Python doesn't implicitly consider two variables as a tuple. List, set, or even in special cases dict can also be specified as an output.

## Arithmetic operators

|  |  |  |
| --- | --- | --- |
| **7%3 = 1** | *modulus* | Returns the remainder of a division. 7/3=2(+**1**) |
| **2\*\*3 = 8** | *exponent* | Returns number multiplied by itself. 2\*2\*2=**8** |
| **9//2 = 4 9//-2 = -5** | *floor  divisor* | Returns result without numbers behinddecimal point. If one of the operands is negative, result is rounded upwards. |

## Comparison operators

Classic comparison operators are '>' and '<'. Python also has these:

**a==b → FALSE** *equal* - If the operands are equal, condition is TRUE  
**a!=b → TRUE** *not equal* - If the operands are equal, it's FALSE  
**a<>b → TRUE** *not equal* - same as a!=b  
**>= ; <=** - Note that the equalizer sign is always at the second position

## Logical/Membership/Identity operators

Python uses classical logical operators **AND**, **OR**, **NOT**. Membership operators test for membership in sequences, like strings, lists or tuples. Identity operators compare the memory locations of two objects.

**x in y** - if x is a member of sequence or string y, evaluates to TRUE. **x not in y** - if x isn't in the sequence or string y, evaluates to TRUE.

**x is y / x is not y** - differs from == operator by not comparing the variables themselves, but instead if they point to the same object or not

## Assignment operators

= += -= \*= /= %= \*\*= //=

These operators assign values to variables. The simplest is '**=**' used in for example '**a=[b,c]**'. The rest perform a mathematical operation and **then** assign the result to the first variable. **a+=b** is thus equal to **a=a+b**

**range()** function is used to generate a sequence of numbers over time. It has 3 parameters: **range([start,] stop [,step])**. Only **stop** parameter is necessary. By itself **range()** returns a range object (not an iterator). To get it to actually display its sequence for printing purposes it needs to be wrapped in a constructor function like **list()**:

**1.) print(range(5))** → **range(5)**  
**2.) print(list(range(5)))** → **[0, 1, 2, 3, 4]**

In these examples only **stop** argument is provided. **stop** is non-inclusive i.e. similarly to indexes (see PYTHON INDEXING) it doesn't actually include the provided number in the output and stops counting before reaching it. **range()** can also accept **start,** or **,step** arguments. Note the comma with them: since **range()** doesn't use colon (**:**) like indexes, these arguments are separated from **stop** argument via commas instead:

**1.) print(list(range(1,5))** → **[1, 2, 3, 4]  
2.) print(list(range(1,5,2))**  → **[1,3]**

**start** parameter says where to start counting from (default being **0**), while **step** parameter says *how* to count i.e. what will be the increments (default being by **1**). In the example above (#2) counting starts from number **1** with the increments of **2**. Since counting stops at number **4**, as the **stop** parameter is non-inclusive, only **1** and **3** make it into result.

## Negative range() values

Using negative values as **range()** arguments works similarly to negative index slicing (see SLICING in PYTHON INDEXING):

**print(list(range(-5,-1)))** → **[-5, -4, -3, -2]  
print(list(range(5,-1)))** → **[]**  
**print(list(range(5,-1,-1)))** → [**5, 4, 3, 2, 1, 0** ]

A for-loop is used to repeat a block of code a fixed number of times, that is given by a sequence (which can be a list, tuple, dictionary, set, and string) or alternatively a sequence function like **range()**. In case of strings it loops over individual characters they consist of:

**1.) for i in 'Python': 2.) example=[] for i in 'Python':  
 print(i) example.append(i)**

Result is 6 individual 1-character strings (#1). Should they be made into a list, it needs to be instatiated first and the results appended into it (#2).

The **in** keyword serves two purposes. In the example above it points at the sequence that the for-loop is iterating through. Secondly it can question the presence of given value in the sequence. See IF, ELIF, ELSE STATEMENTS.

## Range() function in for-loops

**range()** is often used in cases where a specific number of iterations is required. Importantly while **range(2)** indeed consists of 2 iterations, it starts by default from 0 and excludes the last number, 2, so the values are **0,1**. Similarly to indexes, it's possible to specify the argument by adding second and possibly third parameter, like **range(1,6,2)**:

**for x in range(6): for x in range(2,6): for x in range(1,6,2): print(x) print(x) print(x)**

Note that it uses comma **,** instead of colon : unlike indexes (See **PYTHON INDEXING**). It can also be combined with **len()** to iterate through a sequence using indexing. See **RANGE() FUNCTION**.

**for i in range(len(x)): x=['pop', 'rock', 'jazz']**

**print("I like", x[i])**

## If, Else\* and Elif statements

if is a conditional statement, that decides whether an underlying block of code will be executed. Only when **if** condition is met, it is done so:

**x=10; y=5 if x>y: if x in [2,4,6]:**  **print('x is bigger than y') print('inside')**

Here text will be printed, as the condition has been met. In the second case it will not as number **10** is not in that list.

elsestatement in Python is optional and can have two meanings depending on whether it's a complement to **if** statement (**if**..**else**) or is standalone. In the first case, **else** contains the block of code that executes in case the condition for **if** is NOT met. If it does, it's omitted:

**if 'yth' in 'Python':  
 print('Found!')  
else:  
 print('Not found!') → Found!**

If the **else** statement is not used in **if..else** combo, its block of code is executed after the for-loop finishes its last iteration. It's omitted only if the **break** statement is used to stop the loop (see the next page). In case below it doesn't get to number **5** (as it goes through **0-4**) and thus prints **else** block. Printing **if** and **break**ing out would require **range 6** or higher.

**for i in range(5):**

**if i == 5:**

**print("Too big - I'm giving up!")**

**break**

**else:#**

**print("Completed successfully")**

elif statement is an optional continuation of **if** statement, used when there's more conditions then just one to check for meeting the criteria. Unlike **if** and **else**, there can be more **elif**'s per one expression:

**if 'eth' in 'Python':  
 print('Found eth')  
elif 'ith' in 'Python':  
 print('Found ith')  
elif 'yth' in 'Python':  
 print('Found yth')**

### Shorthand If..Else statements

**if..else** statement (including standalone **if** but not **elif** or standalone **else**) can be written in one line similar to comprehensions (see next page)

**print('Found!') if 'yth' in 'Python' else print('Not found!')**

## Break and Continue statements

The **break** statement stops the loop after given criteria are met. **continue** statement stops the *current iteration* of the for loop after given criteria are met and the loop continues with the next iteration:

**fruits = ['apple', 'banana', 'cherry']**

1.) **for x in fruits:** 2.) **for x in fruits:  
 print(x) if x=='banana':**

**if x=='banana': continue**

**break print(x)**  
**→['apple', 'banana'] →['apple', 'cherry']**

1.) **break** stops the loop after second iteration. '**banana**' is the second iteration, yet is still in the result, as **print(x)** is above **break** in the code. 2.) **print(x)** is purposely below **continue** to skip printing **'banana'** from the second iteration and continue to third iteration - printing **'cherry'**.

Comprehensions are effectively for loops that append their results into an iterable. That iterable can be list, set, dict or a generator. They behave exactly the same as for-loops - they can have multiple elements, it's possible to have nested comprehensions and even comprehensions that are expressions of another comprehensions (as they always return an iterable of some sort). Every comprehension includes three elements and can have one optional. Each one has the following syntax:

**[expression for member in iterable if condition]**

**1.)** **expression** - Can be the item itself, an operation with it, function or method call, or an iterable (which includes another comprehension).  
**2.)** **member** - the element from an iterable the expression is about   
**3.)** an **iterable**. Can be a sequence or generator like **range(1,10)  
4.)** *optional* - **a simple condition (no elif, else statement!)**

**outcome=[]**

**1.) for var in range(1,10):**

**if var % 2 == 1:**

**outcome.append(var\*2) ↓  
2.)** **outcome=[var\*2 for var in range(1,10) if var % 2 == 1]**

**var\*2** - *expression* **var** - *member***range(1,10)** - *iterable* **if var % 2 *== 1*** - *condition*

In example #1 a for-loop creates a list out of given range of numbers 1 to 9 with conditional statement allowing only odd numbers which are doubled. In #2 same result is reached via list comprehension. Note that it basically copies it with the exception of list definition and appending.

## Comprehension expressions and members

Expressions define the output- what's gonna happen to the member(s),

a.k.a element(s) of the given iterable. The options are infinite - ranging from simple arithmetical operations like in the example above to functions (like [**len(i) for i …]**), methods (like **[i.capitalize() for i …]**), Boolean checks like the one below, to even another iterables which also includes another comprehensions.

**example = ['at', 'to', 'on', 'in']  
example2 = [10, 4, 2]  
boolean\_check = [i == 'to' for i in example] → [False, True, False]  
value\_check = [i > 3 for i in example2] → [True, True, False]**

Using a Boolean operator instead of mathematical one, expressions can perform checks on the given iterables like the ones above#. Expression can also take form of iterables. Example below results in a list of lists:

**[x, x[0] for x in example]→ [['at', 'a'], ['to', 't'], ['on', 'o'], ['in', 'i']]**

### Expressions with multiple assignment / indexes

Comprehensions can take multiple assigned members to perform operations on (see MULTIPLE ASSIGNMENT). In such case the expression may or may not contain all of these members.

* ***single member expressions***

**example3 = {1: 2, 3: 4, 5: 6}  
print([char1 for char1, char2 in example]) → [a, t, o, i]  
print([value for key, value in example3]) → [2, 4, 6]**

In both of these examples multiple members are defined, but only one of them featured in an expression (and thus output). The same result can also be achieved with indexes (see INDEXES):

**print([num[0] for num in example]) → [a, t, o, i]  
print([i[1] for i in example3]) → [2, 4, 6]**

* ***multiple member expressions***

**example4 = [[1, 2], [3, 4], [5, 6]]**  
**1.) print([a + b for a, b in example4]) → [3, 7, 11]  
2.) print([i[0] + i[1] for i in example4]) → [3, 7, 11]  
3.) print([a, b for a, b in example4]) → --invalid syntax--  
4.) print([(a, b) for a, b in example4]) → [(1, 2), (3, 4), (5, 6)]**

If the expression includes multiple members, special attention needs to be paid towards the fact that the expression must be a single object. In examples **#1** and **#2** above **a + b** and **i[0] + i[1]** are both single object expressions even though they include multiple members because an action is performed over them, which gives a single object result. Note that in the example **#4** **(a, b)** are enclosed in parentheses which is done in order for the two objects **a, b** to become a single object - a tuple.

## Comprehension conditions

**print([i for i in range(1,10) if i < 5]) → [1, 2, 3, 4]  
print([i if i < 5 else 0 for i in range(1, 10)]) → [1, 2, 3, 4, 0, 0, 0, 0, 0]**

Conditions are optional, but very useful. In the case of comprehensions they can also be very confusing. This is because when they contain a more complex statement (like **if..**else) the syntax of the comprehension changes as seen in the example above. A simple **if** condition is at the end of the comprehension. Adding **elif** or **else** to it puts the condition at the beginning instead.   
Therefore it's recommended to use comprehensions only with simple **if** conditions, that puts them at the end of comprehension statements.

**print([get(i, 0) for i in range(1, 10) if i<5]) → [1, 2, 3, 4, 0, 0, 0, 0, 0]**

## Nested comprehensions

Comprehensions can utilize another comprehensions within themselves. This can be done in two ways: either there are two or more loops within comprehension, or the comprehension contains another as its expression:

|  |  |  |
| --- | --- | --- |
| **1** | **2** | **3** |
| **4** | **5** | **6** |
| **7** | **8** | **9** |

**1.) for row in matrix#:  
 for i in row:  
 flattened.append(i)  
  
2.) flattened = [i for row in matrix for i in row]**

In the example above a comprehension (#2) with nested for-loop was used to flatten the matrix (see MATRICES). Note that the order of for-loops inside the comprehension follows the one in for-loops themselves (#1). The parent iterable (in above example **matrix**) always come first.

**[[char for char in str if char in 'aeiouAEIOU'] for str in example] ↓  
 → [['a'], ['o'], ['o'], ['i']]**

Comprehension can also take another comprehension as an expression. In such cases the order is reversed - the 'main' comprehension prepares the iterable for the nested comprehension which will return the result. In the example above 'main comprehension' gets a string via looping over **example** and the nested comprehension checks elements of strings (i.e. characters) against a list of vowels thus filtering them out#. Note that it made a list of lists.

**example5=[['at','to','on','in'], ['of','be','it','by'], ['no','or','he','so']]  
print([[j for j in i if j in 'aeiouAEIOU'] for n in example for i in n])↓  
→ [['a'], ['o'], ['o'], ['i'], ['o'], ['e'], ['i'], [], ['o'], ['o'], ['e'], ['o']]**

## List comprehensions

All of the above **example**s were **list** comprehensions with their output being list of objects. There are 3 more types with different outputs.

Since strings are sequences as well, comprehensions can be used with them in a similar manner to for-loops. Note that for-loop below will not produce the same result as **.append** method is not used:

**letters=[letter for letter in 'Hello'] → ['H', 'e', 'l', 'l', 'o']  
for letter in letters:   
 print(letter) → 'H'; 'e'; 'l'; 'l'; 'o'#**

## Generator comprehensions

Allow for better memory allocation by generating values one by one. Their output is an iterator object, through which a for-loop (or another comprehension) can print results. They are written with round brackets:

**example = range(1000)  
outcome=(var\*\*2 for var in example if var % 2 == 1)  
print([i for i in outcome])**

## Set comprehensions

Aside from the curly brackets and set rules (no duplicates) the sets are identical to lists. Note that the result is randomized but only per different input (not per execution of the script):

**outcome={var\*2 for var in example if var % 2 == 1} #**

## Dictionary comprehensions

These can be used to create dicts out of other dicts as well as lists:

**outcome={var:var\*\*2 for var in example if var % 2 != 0} ##  
outcome2 = {i:(v,k) for (i, (k,v)) in enumerate(outcome.items())}**

First script makes a dict with odd numbers as keys and their squares as values. **outcome2 s**waps values and keys and gives both new keys.

**Lambda** itself is a keyword in Python (see KEYWORDS). The lambda function is a simpler version of a for-loop (similar to comprehensions, see COMPREHENSIONS) with advantage of no need for name definition as it's designed as a throwaway (*anonymous*) function for one-time use. It can take any number of parameters, but can only have one expression and can't contain any statements. Its syntax consists of the keyword itself, declaration of variables, colon (**:**) and declaration of actual operation with variables:

|  |  |
| --- | --- |
| **def add(x, y):** | 1.) **add=lambda x, y: x + y** |
| **return x+y → add(5,3)** | **add(5, 3)** |
| 2.) **(lambda x, y: x + y)(5, 3)** |

Both lambda functions #1 and #2 are functionally the same as declaring a simple function (left side). The utility of the basic form (#1) may not seem like much, but lambda can also be written by putting it in parentheses and adding the arguments to the same row (#2). Lambdas can also be used with functions and methods as shown below:

1.) **(lambda x:2\*len(x))('Python') → 12**2.) **(lambda x:int(x.strip('.')))('...8...') → 8**

1.) - used with a **len()** function to return twice the length of the input.  
2.) - used with **.strip()** method and **int()** function to get rid of the dots in the string and return the integer **8**.

Lambdas are often used as arguments in functions that take another functions as arguments like **sorted()**. See MAP()/FILTER() for more.

**sorted(range(-4,5), key=lambda x: x\*\*2) → [0,-1,1,-2,2,-3,3,-4,4]**

The lambda function above squares its input. It is then used as a key for **sorted()** function. This means the squared numbers provided by lambda are used just as proxy values by which the input values are sorted.

## Map() function

**map()** is a built-in function in Python, that calls a specified function or method for each element of a given iterable. It's effectively a batch processing of multiple values that are each run through a single method or function. It has a similar purpose to comprehensions, which are usually considered better. **map()** has two parameters: 1st for function or method and 2nd for the iterable, that will provide the values:

**words=['at', 'the', 'on', 'in'] 1.) print(list(map(len, words)**  2.) **print(list(map(int, ['1', '2', '3', '4'])**

1.) **map()** is used to get the number of characters of each element. The **len()** function is without parentheses because the function is not called, just referenced for **map()** which will do it itself. Also **list()** is used to print result, because **map()** returns an iterator object, that requires passing to a constructor function in order to create an iterable.   
2.) **map()** is used to convert each string number into an int number.

## Using map() with method

When the function that's supposed to be applied to all elements of a given iterable is actually a method, the correct syntax in **map()** or **filter()** is the object type of the input object with a dot and the method itself. Below is a comparison of **.capitalize** method when applied to a single string and when mapped to multiple ones:

**print('python'.capitalize())  
print(str.capitalize, words)**

**map()** is used with **.capitalize** method to make all first characters of the words capital. Since it's not called, just referenced, a standard method syntax **words.capitalize()** can't be used. Instead the syntax of this

method is stylized into a function **str.capitalize** which is then referenced. This can be done even with methods that require arguments, but it's usually a very awkward code so it's used rarely:

**print(list(map(str.replace, words, 'o'\*len(words), '0'\*len(words))))#**

Here **map()** needs to be provided with as many arguments as there are elements in the source (hence they are multiplied by the **words** length) and these are listed last, i.e. after the source. However using **map()** with **lambda** as in example below (or alternatively using comprehensions) is much easier for this particular task:

**print(list(map(lambda x:x.replace('o', '0'), words)))**

## Multiple arguments for map()

**map()** can also be provided with more than just one iterables as arguments for the function it's supposed to apply. If it's supplied with multiple iterables, then the transformation function must take as many arguments as the iterables passed in. This means functions like **len()** or **str.capitalize** are out of option as they take only one argument:

**print(list(map(pow, [1, 2, 3], [4, 5, 6, 7]))) → [1, 32, 729]**

**pow()** takes two arguments, *x* and *y*, and returns *x* to the power of the *y*. Note the different length of the two input lists. In such cases **map()** is applied to elements as long as they have their match in the other list. Thus result is [1⁴, 2⁵, 3⁶]. This technique allows for merging two or more iterables using different kinds of math operations. Since built-in functions which accepts 2+ arguments are few and far between though, most of the multi-argumented **map()** functions are used with **lambdas** - simple user-defined functions for one-time usage (see LAMBDA FUNCTION**)**:

**print(list(map(lambda x, y, z: x + y - z, [1,2,3,4], [5,6,7], [8, 9])))  
→ [-2, -1]**

Here **map()** is used to extend the lambda to all items in the list. Lambda and its input aren't closed in parentheses like in examples 2 pages back, as they're once again not called, just referenced.

Filter() function

**filter()** is a built-in function that calls a specified function/method for each element of a given iterable and collects those elements, for which that function returns TRUE. It shares many characteristics with **map()** including having two parameters: function and iterable, referencing mechanism behind functions/methods and returning iterator objects. For the sake of comparison **map()** applies a function to every member of an iterable, while **filter()** does the same, but only returns items, for which the function returned TRUE. The provided function thus must contain either comparison, or logical/membership/identity operators:

**list1=[2, 'j', 'u', 'n', 'e'] list2=[7, 'e', 4, 5, 'n', 'u', 'j', 8]**1.) **print(list(filter(lambda i:i in list1, list2))) → ['j', 'u', 'n', 'e']**2.) **print(list(filter(lambda i:i[0]>'m', list1))) → ['n', 'u']**

1.) This function returns only common elements from both lists.   
2.) First character of a string in the list must be bigger than **'m'**.

If the **filter()** is supplied a keyword **None** instead of a function, it will filter all the elements that evaluate to **None, False** or **0**. See below:

**example=['abc', 0, None, True, False, 1-1, 2%2, 3]**  
**print(list(filter(None, example) → 'abc', True, 3**

**zip()** function takes iterables, aggregates them in a tuple and return a zip object. This object, similarly to **range()** one, can then be wrapped into a constructor function like **list()** in order to display its contents:

**list1 = [1, 2, 3] list2 = ['a', 'b'] tuple3 = (0.5, 3.4)  
print(list(zip(list1, list2, tuple3))) → [(1, 'a', 0.5), (2, 'b', 3.4)]**

As seen above, if one of the collections has more items, they're ignored. If just one list is provided, its items will be wrapped in tuples in the returned list. No arguments for **zip()** function result in an empty list. Items can be converted back to their unzipped form via **zip()** with a star (**\***) operator. The process is similar to unpacking tuples (see TUPLES):

**zipped = list(zip(list1, list2, tuple3))#  
list1, list2, tuple3 = zip(\*zipped)**

The above example returns **list1, list2** and **tuple3** in their original state (shown in the previous example on the top of this page). Note that unzipping is not required in this case as the original values were already declared. The above script has just rewritten them with the same ones.

## Pair calculating and Dictionaries

**costs=[500, 250, 120] income=[700, 680, 830]  
1.) dict(zip(costs, income))  
2.) for a, b in zip(costs, income):  
 print(f'Profit = {b-a}')**

#1 - as dictionaries are two-item tuples and zipping two lists produces the same results, all that is left is to wrap the **zip()** object into **dict()**.   
#2 **zip()** is used with for-loops to perform operations over two or more iterables. This is where the function comes most handy. See ADVANCED EXAMPLES for more.

This built-in function prints a given object to the console screen or a text stream file. Aside from the basic objects (see OBJECT TYPES) it can also print functions as well as methods:

**print('a', 'b', 'c', sep=';', end='END')  
print('example') → a;b;cENDexample**

By default **print()** has hidden keyword arguments (*kwargs*) set as: **sep=' '**, **end='\n'.** Every time it's used a newline (**\n**) is used at the end of the line and if it receives more objects to print, they'll be separated by whitespace(**' '**). In above example kwargs are supplied, so **print()** on first line separates individual items with colon instead of whitespace and since **/n** has been replaced with **'END'**, next **print()** continues on the same line.

**from datetime import datetime  
print(''time'+'now=', 2\*datetime.now().strftime("%H:%M:%S"))**

As seen above, **print()** can print multiple objects at once that can be multiplied via **\*** sign, concatenated via **+** sign, or just put next to each other (with comma in between). Note only basic objects of the same type can be concatenated - *<string>*+ *<int>* won't work.

## Printing files and writing into them

Printing file contents requires opening the file via **open()** function (see next page). If the contents have newlines themselves, it's beneficial to remove the default newline via utilizing **end** kwarg as shown above.  
Kwarg **file='',** that isn't utilized by default like the previous two, points   
at a target file into which **print()** contents will be written. First the file needs to be opened via **open()** with a **'w'** argument. This causes Python   
to either write into an existing file or create a new one:

**with open('newfile'.txt', 'w') as example:**

**print('New text in a file', file = example)**

This built-in function returns two values: **count** of the current iteration  
and **value** of the item at the current iteration. These are returned in the form of an iterator, which can be looped over. **enumerate()** has one optional argument, **start**, that can edit the starting number of the count. By default **start**=0, emulating Python indexing style:

**for x, y in enumerate(example):**

**if '\r' in y:**

**yield '\\r in line ', x+1.  
 if y[:-1].rstrip(' \t') != y[:-1]  
 yield 'trailing whitespace in line ', x+1**

In above example **enumerate()** is used with conditional statements to check for tabs and return characters (see STRINGS 2). Indexes are printed with **+1** as Python starts counting from 0. The second conditional compares the end of the line when **.rstrip()** isused and when it's not.

**input=string.ascii\_lowercase  
print([y for x, y in enumerate(input, start=1) if not x%2])**

Here **enumerate()** creates 'indexes', that are filtered by the condition to only allow even ones. If the **input** is an alphabet, then it only prints [**b,d,f…]**. This is useful in cases the input is in form of an iterator as sequences can simply substitute **enumerate()** with index - **input[1::2]**.

**list1 = [5, 1, -10, 3, 3] list2 = [5, 10, -10, 3, 5]  
print([i for i, j in enumerate(zip(list1, list2)) if j[0] == j[1]])**

In this example there are two lists containing some similar values and the task is to get indexes of the pairs with them. **zip()** creates the pairs and **enumerate()** assigns them numbers so they create another pair of one value(index) and two item tuple (pairs) - hence **i, j.** Indexes on **j** then extract individual tuple items and compare them.

While functions **do** some specific thing, classes **are** some specific thing - namely **objects**. Every object is a class of some sort, discoverable by typing **print(type(object))**. This includes ints, lists, iterator objects etc.  
Class groups functions (as class methods) and data (as properties) into a logical unit revolving around some type of thing. It's these methods and data that define the object's behavior. This is why lists for example have **.append()** method or that ints and floats can use division operator.

## Class instances

As noted before class methods and data define the object's *potential* behavior. *Potential*, because it's up to the user which situation he will force on the object and make it *behave*. A list can be defined and used without calling **.append()** even though it has this method just int can be without using division operator. When a list or int (or any other object) are defined (like **some\_list = []; some\_int = 5**) it's a class instance that gets defined. It's up to what the user does with the instance, or how the instance is defined to make use of the methods at its disposal.

## Class definition

Classes are defined by **class** call, followed by the class name. This new object cannot be left purposeless, so further customization is required. Inside a class one of the three things can be defined: **class variable**, **class method** and **class magic/dunder method**.

## **Class variable**

Class variables are defined within the body of the class but not within the body of any specific method (see CLASS METHODS). Class variables's values are shared by all object instances of a class. This is opposed to instance variables whose values can differ across each instance of a class.

**class My\_class:  
 example = 10**

## Class magic/dunder method

Dunder (short for ***d****ouble* ***under****scored*) methods are recognized by the styling of their name definition - double underscore on each side of it. Their names are **predefined and unique** unlike standard methods where no rules apply to the names. This is because each dunder method is tied to certain condition which makes Python call it. For example **\_\_init\_\_** gets called every time a class instance is called. There are many dunder methods with many being object specific, so only the most important will be covered here:

### \_\_INIT\_\_

As mentioned, **\_\_init\_\_** is called upon every class instance call. It's because it *creates* the instance itself. This means that instance variables should go into the body of **\_\_init\_\_** method. While they can also be defined in all the subsequent user-defined methods, it's a good practice to have them all defined in **\_\_init\_\_** for code clarity purposes.

**class My\_class:  
 def \_\_init\_\_(self):  
 self.example = 10**

In this example **\_\_init\_\_** method

**class SearchEngineEntry:**

**codecademy = SearchEngineEntry("www.codecademy.com")**

**wikipedia = SearchEngineEntry("www.wikipedia.org")**

**print(codecademy.url)**

**# prints "www.codecademy.com"**

**print(wikipedia.url)**

**# prints "www.wikipedia.org"**

**f**

→ equal to game.move\_current\_block\_down. Game has it as a function, this is a variable instead. The string is an argument for the class instance call and print calls this class instance call with an attribute url, which has been assigned to the argument of the class instance call. Therefore it returns the argument - the string.

## Datapath manipulation

Both Python and R programs use a certain path as their working directory, where they store the scripts and additional files.

|  |  |  |
| --- | --- | --- |
| P - **import os** | R - |  |
| **os.getcwd()** | **getwd()** | get current working directory |
| **os.listdir()** | **list.files** | list all files in current WD |
| **os.chdir()** | **setwd()** | change current WD |
| **os.mkdir()**  **globals()** | **dir.create**  **ls()** | make new WD  lists all defined objects in an environment |
|  |  |  |

## Importing data using Python built-in functions

Data can be in any of the popular formats - **.csv, .txt, xlsx, Rdata** etc. While importing external files, following points should be checked:  
- Existence of a header row  
- Treatment of special values as missing values  
- Date type variable in consistent date format.  
- No truncation of rows while reading external data

**x=open('file.txt', rt, encoding='utf-8') /  
/ with open('file.txt', rt, encoding='utf-8') as x:**

Default way of loading datafile into Python is through **open()** function. **open()** function should always be used with **with** statement like in the second row in the example above, because it guarantees closing it after it's done. This function describes how the file will be used - **rt** means **'read'** and **'text'** (optionally **w** can be used alongside **r** for **'write'** and **b** instead of **t** for **'binary'**). It returns a file object with methods like **read(), readline(), write()** or **seek()**, that can be read in two ways: reading the whole block of text, or reading it line by line. Reading the whole file loads up faster but with a risk of running out of memory.

**with open('file.txt', rt, encoding='utf-8') as x:  
 print(x.read()) print(list(x)**

**read()** method reads the stream of data from the beginning to the end of file. **x.read().splitlines()** can be used to mimic line-by-line reading. **list()** as well as **set()**, or **tuple()** wraps the file object into a container but doesn't actually read anything - only displays its contents.

**print(x.readlines()) for i in x.readlines():  
 y=i.rstrip()  
 print(y)**

**readlines()** stores the file content in a list, where each line is an item. At the end of each line (minus the last one) a newline character **(\n)** is created. A for-loop using **r.strip()** can be added to get rid of **(\n)**.

**for i in x: y=[] for i in x:  
print(i) y.append(i)  
 print(y)**

A simple for-loop can iterate over all lines and print them either in a top-down or next-to-each other list.

**for i in x:  
 y=x.readline()  
 print(y.rstrip())**

**readline()** gives a similar output to a simple for-loop aside from a different iterator method. It includes newline characters at the end of each line. These can be dealt with using **rstrip()** method.

## Importing data using csv library

Files with multiple fields on each line (*tabular data*) are called **delimited** files. The built-in function can process them, but it's likely to mess up, especially if there are hundreds of fields per line. Csv library can handle delimited files, if they are in .csv, .tsv or .txt format.

|  |
| --- |
| **import csv with open('file.txt', rt, encoding='utf-8') as x: y=csv.reader(x, delimiter=',') a=[], b=[] for i in y: y=csv.dictReader(x) print(i) for i in y:  a.append(i['colname1'])  b.append(i['colname2'])** |

**csv.reader()** reads all lines and returns a reader object with each line as list of strings. If there's a header row, **csv.DictReader()** can be used to read each line as dict with headers as keys. This is done by appending the actual names of the columns to individual lists. Data in each column can then be accessed by calling it's fieldname in the header row.

## Importing data using pandas library

Pandas library is useful when the filesize is big. It doesn't require the **open()** function; loads the file content into pandas dataframe by itself, automatically recognizing header rows and datatypes of each field.

|  |  |  |  |
| --- | --- | --- | --- |
| **import pandas pd.read\_table()** | **pd.read\_csv()** | **pd.read\_sql()** | **pd.read\_excel()** |

All four functions read a specific filetype into a pandas dataframe. **pd.read\_excel()** by default loads only the first sheet. However **pd.ExcelFile()** can be called first and the sheetname specified as an argument for **pd.read\_excel(). read\_table** reads delimited files like .tsv. See DATAFRAMES as pandas will be used for them often.

## Keyword arguments (Kwargs)

Keyword arguments in general are just distinction of the argument we are about to specify like **col='Age'** instead of just **Age** parameter as the second input (rows come usually first). However arbitrary keywords are separate dictionaries, which can specify arguments as well. These dictionaries may be called inside the function as well as outside of it. This is done by calling the name of a dictionary inside the function:

**g=sns.FacetGrid(df, row='Name', col='Age', hue='Name')  
kws=dict{s=50, linewidth=0.5, edgecolor='black'}  
g.map(plt.scatter, 'Age', 'IncomeGroup', \*\*kws)**

For calling the kwarg inside the function simply put a dictionary inside the function that can take them (this is specified in their Help menu):

**sns.set\_style('darkgrid', {'axes.facecolor':'black'})**

|  |  |
| --- | --- |
| **(lambda a,b,c: a+b+c)(1, 2, 3)** | **(lambda \*arg: sum(arg))(1,2,3)** |
| **(lambda a,\*,b=0,c=0:a+b+c)(1,b=2,c=3)**  **(lambda \*\*kwarg: sum(kwarg.values()))(x=1,y=2,z=3)** | |

Just like regular functions, lambdas support multiple ways of passing arguments. Following examples all result in output of

## Operations over elements of 2+ sequences:

### Zip() + For-loop / list comprehension combo

**first = [1, 2.3, 'pyt', [0, 4]] second = [2, 0.5, 'hon', [9, 7]]  
result = [x + y for x, y in zip(first, second)]  
→ result = [3, 2.8, 'python', [0, 4, 9, 7]]**

Zip function aggregates the elements of both functions into two-item tuples over which then for-loop performs desired operations and appends them into the result list. As seen above, these elements can be ints, floats, strings and even other lists.

### map() + lambda combo

**first = (1, 2.3, 'pyt', [0, 4]) second = (2, 0.5, 'hon', [9, 7])  
result = tuple(map(lambda x, y: x + y, first, second))**

Map simply applies a given function on each item of given iterable(s).

## Operations over list elements with indexes

### Removing middle elements of list

This can't be done by one slice, because slicing will actually return the part that is supposed to be deleted, and delete everything else. Thus two slices must be performed and then joined together via + operator:

**example = [4, 8, 15, 16, 23, 42]  
example[:1] + lst[3:] → [4, 16, 23, 42]**

### Sorting & Slicing

In order to sort a list of tuples by an element of in the tuples, the easiest way is to have this element as the first in the tuple. To sort **zip(names, ages)** by ages, **zip(ages, names)** can be done instead via list comprehension:

**example[names, ages for ages, names in zip(ages, names)]  
example.sort()** or **sorted(example)**

In case the tuples should remain untouched, a **lambda** must be utilized:

## **print(sorted(example, key=(lambda example: example[1])))**

## Periodical filtering of returned values

Use the % operator. For even numbers :

**values = []  
for index, value in enumerate(iterable, start=1):**

**if not index % 2:**

**values.append(value)**

## Creating comparison bar chart using comprehensions

A comparison chart consists of two or more datasets, whose elements are displayed in chart side-by-side touching each other. The bars have a width of 0.8 (a standard for the graphing library Matplotlib).

**values\_1 = [2\*index for index in range(5)]  
values\_2 = [2\*index + 0.8 for index in range(5)]**

**values\_1** generates 5 bars with 0.8 width and 2 gap (calculated from the same point as width, so it's actually 2-0.8=1.2). **values\_2** adds other bars to the left of the ones **values\_1** generated. Finally at midpoints of each double-bar x-ticks labels need to be generated via comprehensions:

**midpoints = [(x1+x2)/2 for (x1, x2) in zip(values\_1, values\_2)]**

## Filtering ints from mixed num/char strings list

*Using list comprehensions:*  
**example = ['306M', '2B', '65.8M', '326M', '60.3B', '208M']**  
**chars = [[char for char in i if char in '0123456789'] for i in example]  
floats = [float(''.join(num)) for num in chars]**

First list comprehension is composed of two, where the first one is an expression of second one. It extracts lists of single digits that comprise the actual numbers. Second comprehension joins these ints together. Note the **float()** instead of **int()** used, because numbers are decimal.

*Using .rstrip() with indexing:*  
**floats = [float(char.rstrip(char[-1])) for char in example]**

In this specific case where the chars are at end of string this is usable.

*Using .replace()***floats = [char.replace('M', '').replace('B', '') for char in example]**

*Using RE module:*  
**import re  
floats = [''.join(i) for i in [re.findall(r'\d', char) for char in damages]]**

## Getting unique letters out of a string

**import string   
alphabet = string.ascii\_letters   
unique\_letters=[i for i in set(example) and set(alphabet)]**

This list comprehension uses comparison operator **and** to extract unique letters out of given **example**. **and** operator allows only elements contained in both inputs and **set()** only unique elements, so the result are unique letters in **example** (since **alphabet** is the whole alphabet).

## Counting multiple categories of items

The most efficient way to do this is with the use of dicts. Each key in the dict represents a category, each value it's count:

### With get() method

**def word\_count(sentence): counts = {}**

**for word in sentence.split():**

**counts[word] = counts.get(word, 0) + 1**

This function counts the number of unique words. Each new words the for-loop encounters makes a new entry in **counts** dict as new key index. Value is then the result of **.get()** method which returns either a **0** or last value assigned to that exact key. Note that the new key is not created per se, until it has a value. Therefore even though **counts[word]** will create a key, the **.get()** will still return **0** as the key is valueless. The reason it returns **0** with **+1** and not just returns **1** instead is that the number wouldn't increment as the same value would keep getting returned upon encountering more instances of the same word.

### With .setdefault() method

**def word\_count(sentence): counts = {}  
 for word in sentence.split():  
 counts.setdefault(word, 0) + 1**

## Creating multiple categories of items

Once again the most efficient way to do this is with dicts. Although this time were creating new entries, instead of counting them, the code is almost the same, because even when we counted, we still created a dict with entries. They just had values as incrementing ints. This time, the values will be extending lists:

## Copying an object

When copying an object, the method **object.copy()** must be used or the all:all (**[:]**) index must be utilized! Otherwise the same object will end up being referenced with another variable!

**example = [1, 2, 3] example[1, 2, 3]  
new\_example = example new\_example=example[:]  
new\_example[2] = 6 new\_example[2] = 6  
print(example) → [1, 2, 6] print(example) → [1, 2, 3]**

Getting object informations

**type(object)** - returns *'int'* or *'float'* or *'str'* or *'list'* or *'tuple'* or *'set'* or *'dict'*

## Creating list of lists via for-loop

**value\_list = []**

**for index,value in enumerate(input\_list):**

**if index % 3:**

**value\_list.append([value])**

**import re example='Python'  
re.search('yth', example)# → <sre\_Match object; span=(1, 3)>**

RE, RegEx or **reg**ular **ex**pression) is used when searching for patterns in given strings.

## RegEx metacharacters

There are 10 RE metacharacters: **[]** , **.** , **\** , **^** , **$** , **\*** , **+** , **{}** , **|** , **().** They are characters with special meaning that are used to specify the conditions of pattern searching for RE interpreter. Note that Python has its own metacharacters that it gets to execute the provided code first. If there's a Python metacharacter in string, it needs to be escaped or raw strings must prevent Python interpreter from executing on it. See S**TRINGS** 2.

|  |
| --- |
| **[]** 1.) **re.findall('th[aos]', 'Python') → ['tho']**  2.) **re.findall('th[aos]', 'Pythan') → ['tha']**  3.) **re.findall('th[c-i]n', 'Pythen') → ['then']**  4.) **re.findall('yt[-k.]', 'Pyt-hon') → ['yt-']** |
| . 5.) **re.findall('ab.cd', 'abxcd') → ['abxcd']**  6.) **re.findall('ab.cd', 'ab&cd') → ['ab&cd']**  7.) **re.findall('ab.cd', 'ab\ncd') → []** |

The metacharacters above fall into ***single character matches*** category. Brackets specify character class. Any character inside will be considered for matching. In case #1 if either '**Python'**, **'Pythan'** or **'Pythsn'** are found in text, it returns **match**. Brackets accept hyphen (**-**), that specifies range of characters. **c-i** means **c,d,e,f,g,h,i** (#3). **0-2** means **0,1,2**. Should '-' be one of the characters, a backslash needs to be used (see next page), or it needs to be placed as first (#4) or last character in the brackets.   
Dot (**.**) matches any single character except for newlines (#7). There has to be a character in the place of '**.**'Thus for example **'abcd'** won't match.

|  |
| --- |
| **^ (\A)** 1.) **re.findall('^Pyt', 'Python') → ['Pyt']**  2.) **re.findall('^Pyt, 'honPyt') → []** |
| **$** **(\Z)** 1.) **re.findall('hon$', 'Python') → ['hon']** 2.) **re.findall('hon$', 'honPyt') → []** |

***Anchors*** don’t match any characters in the searched string. Instead they point at a particular location in it where a match must occur. Caret (**^)** stipulates that substring, which *follows* it must be present at the start of the searched string (i.e what searched string *begins* with). Conversely dollar (**$**) has a text *preceding* it, for which RE seeks a match at the *end*.

|  |
| --- |
| **\*** 1.) **print(re.findall('Pyt-\*hon', 'Python'))** zero or more repetitions |
| **+** 2.) **print(re.findall('Pyt-+hon', 'Pyt-hon'))** one or more repetitions |
| **?** 3.) **print(re.findall('Pyt-?hon', 'Python'))** zero or one repetition |
| **{}** 4.) **print(re.findall('Pyt-{2}hon', 'Pyt--hon'))** exactly 2 repetitions  5.) **print(re.findall('Pyt-{2,4}hon', 'Pyt---hon'))** 2 or 3 repetitions |

***Quantifiers*** indicate how many repetitions the preceding regex pattern must have for the match to succeed. In above example '**\*'** and '**?**' returns match even though the searched string has no dash character in it, because both allow zero (aka none) repetitions. Number of repetitions can be specified in curly brackets as well as numbers range (#5).

|  |
| --- |
| **|** 1.) **re.findall('Pyt|hon', 'Python') → ['Pyt', 'hon']** |
| **/** 1.) **re.findall('[ab\-e]', '123-456') → ['-']** |

Pipe (|) is ***alternating*** metacharacter. It's an equivalent of logic keyword OR. In example above, RE is searching for **'Pyt'** and **'hon'** at the same time. Since **findall** was used, it lists both (see RE MODULE 4).  
Backslash is ***escaping*** metacharacter. It stops Python and RE interpreters (see STRINGS 2) from treating characters as metacharacters - ***escapes***

them) like in case on the previous page, where RE would otherwise seek for characters **'a','b','c','d','e'** (because of **b-e**). Alternatively it can also introduce special sequences like the ones below:

## Special sequences

Also called *escape sequences* because they all consist of a backslash character, which escapes other characters. All of them have two forms - lowercase and capitals which do exact opposite of each other.

1.) **re.findall('\w', '#(**[**a.$1@\_&**](mailto:a.$1@_&)**') → ['a', '1','\_']** 2.) **re.findall('\d', 'abc4def') → ['4']** 3.) **re.findall('\s', 'Py$t\*4 o\n') → [' ', '\n']** 4.) **re.findall('\S', ' \n Pyt \n n') → ['P', 'y', 't', 'n']**

**\w** seeks for any alphanumeric characters including capital letters and underscores (its full scale is equal to **[a-zA-Z0-9\_]**. **\d** seeks for **d**igits only and **\s** for white**s**paces characters including newlines. Example #4 shows **\S** doing the opposite of **\s** - returning individual characters.

**(\b)** 1.) **re.findall('\\bth\\b', 'Py th on') → ['th']** 2.) **re.findall('\\bth\\b', 'Python') → []  
(\B)** 3.) **re.findall('\\Bth\\B', 'Python') → ['th']**

**\b** stipulates word boundaries at either start or the end of a string. In example #1 and #2 **\b** is from both sides, simulating searching for the whole word - in this case **'th'**. Conversely **\B** only searches for **'th'** in strings containing additional characters.  
Note double backslashes on these examples. Python and RE parsers both accept **\b** and **\B**. Python one interprets it as a backspace metacharacter (see STRINGS 3). Double backslash skips Python parser and allows RE parser to interpret it as boundary. This is done easier with raw strings:

**re.findall(r'\bth\b', 'Py th on') → ['th']**

## RE methods

**example = 'It was in it since: 30.03.88'**

**print(re.findall('[Ii]t', example)) → [It, it]**This method returns all occurrences that match a given pattern. Output is a list of matches in the found order, separated by commas. It's very useful in cases like finding email addresses in text.

**print(re.search('[Ii]t', example)) → <…Match object; span=(0, 2)…>  
search()** returns the first occurrence of a given pattern in the form of a match object (shown incomplete above) with coordinates of the match. If the object isn't found, it returns **None**. This makes it more suitable for testing a regex or for conditional statement usage.

1.) **print(re.sub('.', '/', example, 1)) → 'It was in it since: 30/03.88'**2.) **print(re.subn('.', '/', example)) → ('It was in set since: 30/03/88', 2)**  
These methods replace patterns in text. **sub()** returns new string. **subn** a tuple that contains new string as well as count of the replacements. Note that the date in example #1 only has the first dot replaced. This is because in this case an optional parameter, ***count*** has been specified as **1**.

**print(re.split(':', example, 1) → ['It was in it since', ' 30.03.88']**This method splits the string where there is a match and returns a list of strings created by splitting the original. An optional parameter, **maxsplit** has been used above, specifying maximum number of splits performed.

**precompiled=re.compile('\d+')  
print(re.findall(precompiled, example)) → ['30', '03', '88']**This method caches the regex for later use. It's beneficial in situations, where there is more use cases for a single regex, which, in case of changing, would require changing regexes in all use cases.

Glob module helps listing files under specified folder. They may be filtered based on extensions or parts of their filenames.

JSON is a standardized format for storing and exchanging data. Process of *writing /reading* data in JSON is called **serialization/deserialization**.  
**Serialization** (writing) is done with **.dump()** and **.dumps()** method. Python objects are translated into JSON in a conversion format below:

|  |  |
| --- | --- |
| **dict** | **object** |
| **list, tuple** | **array** |
| **str** | **string** |
| **int, long, float** | **number** |
| **True/False** | **true/false** |
| **None** | **null** |

**dump()** requires first opening a file with **open()** using **'w'** argument (see **IMPORTING FILES**). It takes two arguments: one for the data and one for the target file. **dumps()** saves the file into memory, so it doesn't need opening any file and just needs to be assigned to a variable:

**data = {**

**'president': {**

**'name': 'Nelson Mandela',**

**'nation': 'African'**

**}**

**}**

1.) **with open('file.json', 'w') as target:** 2.) **data=json.dumps(data)**

**json.dump(example, target\_file)**

**Deserialization** (reading) is done via **load()** and **loads()** method. It uses the same conversion rules as the serialization shown in table above. This means that converting an array back to a tuple will instead result in a list. In such case tuple object can be restored by calling **tuple()** on that list:

1.) **with open('file.json', 'r') as target: 2.)** **data=json.loads(input)  
 data=json.load(target)**

Arrays and lists have the same way of storing data. But arrays can hold only single data type elements, whereas lists can hold multiple ones.

## Creating matrix

P - **np.arange(0,20).reshape(4,5) np.array([[1,2,3],[4,5,6]])**R - **matrix(1:20, 4,5) rbind(c(1,2,3), c(4,5,6))**

|  |  |  |
| --- | --- | --- |
| 1 | 4 | 7 |
| 2 | 5 | 8 |
| 3 | 6 | 9 |

Unlike R, Python doesn't have a built-in data type of matrices. It has **arrays** and **np.arrays** packages instead, which are used for creating *n*-*d*imensional arrays. Matrix is basically just a two dimensional array here.

## Matrix indexing

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

In R, numbers are sorted by the columns:  
**matrix[1:2, 2]**   
**matrix[0:2, 1]**

In Python, numbers are sorted by rows:

Counting starts from 0 and doesn't include last number in case of specifying a range (x:y)

Python can sort by columns by adding a parameter **order = F** behind the dimension numbers specification:  
**np.arange(0,20).reshape(4,5, order='F')**

## Adding row and column

P – **np.append(matname, [['c', 4 8,7]], 0]**If used without the 0 at the end, it will stretch and flatten the matrix. If used with 1, it will add new matrix to the side of the existing one.  
R – **rbind(matname, c(1,4,8,7)**

P – **np.append(matname,[['c'],[4],[8],[7],[1]])**R – **cbind(matname, 'newcolumn')**

Difference between single and double bracket:

**np.array([0,0,0,0].shape – (4, )** list of four items  
**np.array([[0,0,0,0]]).shape** – **(1,4) list of one item, which is another list.**

**import pandas as pd** – Python package necessary for working with dataframes. Pandas library has its own datafile interpreter for importing and will be used for many operations with them.

## Creating a dataframe

P - Method#1 - Create DataFrame from list of lists.  
**df = pd.DataFrame([['Tom', 15], ['Nick', 26], ['Sam', 32]], columns=['Name', 'Age'], index=['One','Two','Three'])**

P - Method#2 - Create DataFrame from dictionary of nparrays/lists  
**df= pd.DataFrame({'Name':['Tom', 'Nick', 'Sam'], 'Age':[15, 26, 32]}, index=['One','Two','Three'])**

#R – Method#1 – Create DataFrame from lists  
**#df←data.frame('Name'=c('Tom', 'Nick', 'Sam'), 'Age'=c(15, 26, #32), index=c('One','Two','Three'), row.names='index')**

#Note that R's indexing works by specifying a column you need to t#ransform into an index.

Dataframe **df** created by these scripts can be seen on the next page.

## **Dataframe explore commands**

P – **df.head() #**R - **head(df)**  
*- first 5-6 rows (can be specified in the parentheses)*  
**df.tail() #tail(df)**  
*- last 5-6 rows (can be specified in the parentheses)***df.info() #str(df)**  
*- informations about object***df.describe() #summary(df)***- math(mean, first quartile etc.)*

## Numbers of rows/columns

P – **len(object) len(object.columns)**#R – **nrow(object) #ncol(object)**

## Renaming rows/columns

**object.index=['a','b','c'] #rownames(object)←c('a','b','c')  
object.columns=['x','y','z'] #colnames(object)←c('x','y','z')**

## Indexation of dataframes

|  |  |  |
| --- | --- | --- |
|  | **Name** | **Age** |
| **1.** | Tom | 15 |
| **2.** | Nick | 26 |
| **3.** | Sam | 32 |

P – dataframe index is impicitly set for rows. Column indexes are accepted when passed as **names** (like **loc** function).

P – **df[0:3] ; df.iloc[0:1,2:3]**P – **df.loc['3.', 'Name':'Age'] ; df.iloc[0,1]**P – **df.iloc[2,:]** – pandas needs **[:,:]** to be filled  
**iloc** essentially makes dataframes indexing the same as for matrices.  
P – **df[['Name','Age'],[2:]]** – Row indexes need to be designated separately. Also when passing more than one name of the column, the names need to be passed as list, therefore 2 double brackets – one is index for **df** and the other is the list.  
#R – **df[3:3, c('Name', ''Age')]** – the same. Note that in R rows must be specified first.  
#R – **df[3, ] ; df[1,2]** – same as with matrices

P – **df.iat[0,3]** – like **.iloc**, but '**:**' cannot be passed as argument  
P – **df.at[3, 'Name':'Age']** – like **.loc  
.iat** and **.iloc** are position based which mean that they are staying true to the position of the index. If you move the structure of the table, they will point to changed index location: **df[::2].iat[1,1] => 32**

## Quick access

The column name must be one word. Column 'A Pie' won't work.  
P – **df.Age[:-2]** R – **df$Age[:-2]**

In Python **[:-2]** means only first three rows displayed. In R it means all rows except first two. This is also why in R we can use **df[-2:-4]**

## Mathematical operations/Adding columns

P – **df['result']=df.Age\*df.Age**P – **df['try'] = [5] –** error length≠index#R - **df['result']←df$Age\*df$Age  
#R – df['try']←c(5)** – fills the column with 5's

#Unlike Python, R, being a vectorized language fills a given vector #with multiplication of the number or pattern of numbers (if given #more than one), if it doesn't match the length of that vector.

## Removing columns

P – **df.drop('Age', 1)** – 1 means y-axis   
P – **df.drop(df.columns[1],1)**  
#R – **subset(df,select = -c('Age')**#R – **df[ ,-2]**for permanent removal just assign the script to a new object – either the same in case you want to permanently remove, or a new one.

P – **df=df[ ,-2]** #R – **df←df[ ,-2]**

## Removing rows

P - **df.drop(df.index[[1,2]])** – like 1:2   
#R - **df←df[-2,]**

P – **df.drop(['n1','n2'])**#R – **df[row.names(df)%in%c('n1','n2')), ]**

## Filtering dataframes

P - **df[df.Age<20]** – requires column statement#R – **df[df$Age<20, ]** – does not require column statement

P – **df[(df.Name!='Sam')&(df.Age<20)]** – requires brackets  
#R – **df[df$Name!='Sam'&df$Age<20]** – does not require brackets  
Boolean queries return lists of TRUE/FALSE, which is why they are used just as indexes.

## Categoricals/Factors

'Categorical' is a special data-type for Pandas, that doesn't allow for numerical operations. In R there's a similar data-type called 'Factor'. These are useful in the following cases:   
1.) We have a string variable, consisting of only a few different values. Converting this variable to a categorical will save some memory.   
2.) The lexical order of a variable is not the same as the logical order (“one”, “two”, “three”). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order.

**cat=pd.categorical(['a','b','c','a','b','c','d'], ['c', 'b', 'a'], ordered=TRUE)** – second argument signifies the categories. Any value not present in them categories (here 'd') will be treated as NaN.  
Ordered=TRUE enables logical ordering, i.e. c<b<a.  
**df.Name=df.Name.astype('category')** – change data type to category **df.Name.cat.categories** – get the categories (all variants of the rows under specific column 'Age').

P - **import matplotlib as plt  
%matplotlib inline** – Python needs to import this package for plotting  
R – base plotting included by default. For advanced operations **library(ggplot2)** should be used.

## Preparing the data

**df.Age.unique() - Lists all unique values of row index/column. Its useful when looking out for possible categoricals.**

**df\_cropped=df[df.Name.isin('Nick', 'Sam')]** - In order to narrow the data, parameter **isin** crops the dataframe for the desired values.

## Preparing a plot visualization

P – **plt.figure(figsize=12,6))** – size of the plot  
**plt.legend(Players, loc='best', bbox\_to\_anchor=(0,0))** – loc means location of the legend box. Best will pick the most non-intrusive one. **bbox\_to\_anchor** is used then getting the box out of plot boundaries

P - **plt.xticks(list(range(0,len(Seasons))), Seasons)** – markings for an **x** axis. First a list of numbers needs to be created, and then it needs to be populated with actual markings.

P – **sns.set\_style(darkgrid)** – Seaborn package needs to be imported (see *Advanced Visualization* – changes background of the plot.

R – **par(mai=c(1, 0.5 , 1, 0.5)** – size of the plot  
**plot(vec, xaxt='n', yaxt(='n')** – hiding the default **x** and **y** ticks. This is required in order to get custom ticks on the axis.

R - **axis(1, at=seq(0,30,by=5), labels = 'xticklabel')** – 1 on the beginning means **x** axis specification (**2** is y). First a list of numbers needs to be created, then it needs to be populated with actual markings:  
**seq(0,30,by=5)** creates a 7 digit vector of [**0,5,10,15,20,25,30]**. These numbers actually represent a distance in mm on the axis.  
**legend('topleft', c('vec1','vec2','vec3'), text.col = c('blue','red','green')**

## Advanced visualization (preparation)

P – **import matplotlib.pyplot as plt  
import seaborn as sns** – Seaborn is an upgrade to matplotlib allowing for more complex and intelligent plotting  
**%matplotlib inline**

**sns.set\_style('darkgrid')** – style signifies the background of the plot. This argument can be written anywhere, since it will affect all sns plots created, including normal pyplots

## Creating a plot visualization

P - **plt.plot(vec, c='red', ls='-.', marker='o', ms=5, lw=2)** – line plot. Good for showing how the value of something changes over time, or compare how several things change over time relative to each other. **vec** – dictionary can be used in the form of an index to **vec** with another index to dictionary itself. See **DICTIONARIES  
c** – color of the chart line. **ls** means line style. **marker** is marker style; **'o'** means a point style. **ms** means marker size and **lw** line width (size).

R – **plot(vec, type='b', col='red', lwd=2, cex=1.5, pch=21, lty='dotted')  
type** is plot type (flowchart, boxchart etc.). **col** – color of the chart line. **lwd** means line width (size). **cex** means marker size. **pch** is plot character (style), and **lty** line type (also style).

## Plot types

**plt.hist(df.Age, bins=3)** – histogram. Good for single columns when we need to see distributions. **bins** specifies number of cylinders.

**plt.scatter(df.Age, df.IncomeGroup)** – scatter plot. Requires input for both axis x and y. Good for identification of correlational relationships. If we were given a particular horizontal value, what a good prediction would be for the vertical value. What will vertical value do, with horizontal rising?

## Plot stacking

|  |
| --- |
| **plt.hist(df[df.Name=='Nick'].Age), bins=5 plt.hist(df[df.Name=='Sam'].Age), bins=5** |

More chart scripts in one cell results in one chart featuring all the scripts. In these scripts boolean indexation is used and then applied to one particular column. Each new histogram will appear in front.

They can be also shown next to each other when put into one list:  
**plt.hist([df[df.Name=='Nick'].Age, df[df.Name=='Sam'].Age])**

or stacked on top of each other:  
**plt.hist([df[df.Name=='Nick'].Age, df[df.Name=='Sam'].Age], stacked=True)**

## Seaborn plot types

**sns.distplot(df.Age, kde=False)** – Seaborn version of a histogram. Displays a line of Gaussian estimate by default; **kde** deletes this line.  
**sns.distplot(df.Age, bins=3)** – number of cylinders in plot

**sns.boxplot(data=df, x='Age', y='IncomeGroup')** – boxplots show the density in a simple format – a cylinder, which represents 25% quartile, 50% quartile (the line inside a cylinder) and 75% quartile. Quartiles represent the number of elements which are less than the percentage of the quartile. In boxplot this means that a lower gap between middle line and one of the quartiles (edge of a cylinder) means higher density in that area.

**sns.lmplot** – special type of plot, needs specification of the attributes.  
**sns.lmplot(data=df, x='Age', y='Income', fitReg=False, hue='Country', height=5, aspect=2)**Lmplot creates a scatterplot with linear regression line along the dots by default; **fitReg** deletes this line. Lmplot has its own size input parameters, so **figure.figsize** will have no effect on it. **height** is the primary size input parameter, **aspect** specifies the ratio of the sides. Dots and lines can be colored by another input with the **hue** parameter.

**sns.jointplot(data=df, x='Age', y='Income', kind='hex')**Jointplot's main advantage is displaying distribution cylinders on the side of x and y axis of the scatterplot it creates by default. The **kind** argument specifies whether it should be a scatterplot, or other type.

**sns.kdeplot(df.Age, df.IncomeGroup, shade=True, shade\_lowest=False, cmap='Reds')**  
Kernal density estimate plot is similar to the scatterplot (lmplot/jointplot), though it doesn't use dots. It shows where the biggest density is and how the data is distributed across two given variables for x and y axes. **shade** adds more color and highlights the densities. **shade\_lowest** removes the least frequent density (0) to make the grid appear instead. **cmap** changes the color of the density map.

**sns.violinplot(df.Age, df.IncomeGroup)**Violinplot works on a very similar principle to boxplot, but can show more information on density. Boxplots are restricted to the information provided by the gap between the quartiles. Violinplots instead widen the areas with more density. The more observations are in certain area, the wider this area will be.

## Subplots/Dashboards

Subplots put one or more plots against each other for comparison.

|  |
| --- |
| **yoursp, yourcharts = plt.subplots(2,2, figsize=(12,6), sharex=True, sharey=True) k1=yourcharts[0,0].hist(df.Age, bins=5)**  **k2=yourcharts[0,1].scatter(df.Age, df.IncomeGroup, bins=5) k3=sns.kdeplot(df.Age, df.IncomeGroup, ax=yourcharts[1,0] k4=sns.violinplot(df.Age, df.Spendings, ax=yourcharts[1,1] k1.set(xlim=(-20,160))** |

A subplot consists of subplot itself and individual plots within. **yoursp** is a given name of the subplot object itself and **yourcharts** is a given name for the list of plots. **plt.subplots** function creates the subplot with parameters **2,2** representing 'rows' and 'columns' for the plots inside. Note that in this case Python numbers logic isn't used, so 2,2 means 4 cells in the subplot, not 6. If **plt.subplots()** is left empty, it can be used as a field for a single plot. **figsize** adjusts the size of the whole subplot (alternatively **set\_size\_inches** can be used). **sharex** and **sharey** evens the sizes of the plots inside for better comparison.

For each individual plot there needs to be an argument specifying what row/column the plot will occupy. This argument differs for native pyplots and seaborn ones. For native (case **k1** and **k2**), the position is specified first with the plot itself as attribute. Since the subplot itself is pyplot package (**plt.subplots**), **plt** is not called again.   
For a seaborn plot (case **k3** and **k4**), the argument **ax** is needed for specification of coordinates.   
**set(xlim=(-20,160))** specifies the length for x axis on first plot and since **sharex=True** also on the second one. Any type of plot can be used.

## Dashboard styling tips

|  |
| --- |
| **yoursp, yourcharts = plt.subplots(1,2, figsize=(12,6))**  **sns.kdeplot(df.Age, df.IncomeGroup, ax=yourcharts[1,0], shade=True, shade\_lowest=True, cmap='Reds')**  **sns.kdeplot(df.Age, df.IncomeGroup, ax=yourcharts[1,0], cmap='Reds') sns.violinplot(df.Age, df.Spendings, ax=yourcharts[1,1], palette='YlOrRd' k1.set(xlim=(-20,160))** |

A **kdeplot** can be made to look better by using doubling the plotchart with no shades. In this script there are two kdeplots occupying the same cell ([1,0]), with only the first one with shading. This softens up the ledges creating a more professional looking chart.

Different set of arguments apply for a **violinplot**, which doesn't use **cmap**, but **palette** instead. **'YlOrRd' means 'Yellow', 'Orange' and 'Red'. These three colors will fade into each other through individual violinplots.  
A good tip for finding out what the color choices offers are is to intentionally write a typo in the cmap= or pallete=. The error message will list all the possible choices for that argument.**

**sns.set\_style()** sets the background for plots, however by default there are only 5 choices. This can be circumvented using the keyword:  
**sns.set\_style('darkgrid', {'axes.facecolor':'black'})** The keyword will set the color to black, but won't override the grid. If there's no need for grid, we can simply change **darkgrid** to just **dark**.

|  |
| --- |
| **yoursp, yourcharts = plt.subplots(1,2, figsize=(12,6)) sns.kdeplot(df.Age, df.IncomeGroup, ax=yourcharts[0,0], shade=True, shade\_lowest=True, cmap='inferno')**  **sns.kdeplot(df.Age, df.IncomeGroup, ax=yourcharts[0,0], cmap='cool') sns.kdeplot(df.Spendings, df.IncomeGroup, ax=yourcharts[0,1], shade=True, shade\_lowest=True, cmap='Blues\_r')**  **sns.kdeplot(df.Spendings, df.IncomeGroup, ax=yourcharts[0,1], cmap='gist\_gray\_r')** |

These two are examples of professional looking **kdeplot**. The first one uses **cmap='inferno'**, while its second version uses **cmap='cool'**. Inferno fades in yellow color in direction to the center, while white-to-blue fade-in from the second version gives it a nice contrast especially on the black background. The second **kdeplot** uses **'Blue\_R'** reverse blue map that fades blue into white towards the center. **gist\_gray\_r** is used for contrast doing the opposite (dunno why?) with the gray color.

## Facet grid

|  |
| --- |
| **yourgrid=sns.FacetGrid(df, row='Name', col='Age', hue='Name') g.map(plt.scatter, 'Age', 'IncomeGroup') g.add\_legend()** |

Facet grid is a seaborn construct that allows splitting a chart into many based on a given input. It works similarly to subplots in a way that it first requires to create a facet grid itself and then populate it. Unlike subplots though, facet grids are not populated by another charts. Instead they use a map function to create many versions of one chart, which differ by a certain given input.   
  
On the first row, **sns.FacetGrid** is used to create the grid itself. The **row** and **col** arguments indicate by which column of the **df** will be the rows and columns of the chart cells sorted.

**g.map** populates the cells. Number of arguments differs by the plot type: **plt.scatter** here requires input for both x and y axis. In this case the resulting 4 charts demonstrate the relationship between age and income group over given age and name. Since age is being used as both facet grid and plot input, the results will create a vertical line, as x axis becomes irrelevant. **g.add\_legend** adds a legend box to the side

## Coordinates and Diagonals for facet grid

Coordinates are sizes of the axes. These are easily set with limitations on x and y axes (Note: both the coordinates and diagonals are tied to facet grids and should be written in the same cells as facet grids.

**yourgrid.set(xlim=(0,100), ylim=(0,100))**Diagonals are lines inside the charts, that allow for better understanding how for example the dots inside a scatterplot align. If they are aligned over the diagonal, we can assume younger people have bigger income, if under, it means experience shows in salaries ☺:

**for a in yourgrid.axes.flat:  
 a.plot(0,100), (0,100), c='gray', ls='--')  
axes is a summary designation of all the charts inside the facet grid. By adding the attribute flat, the set of charts will be taken as a flat one -dimensional list, so there is no need to address multiple dimensions of the matrix, which the facet grid actually creates. plot is a standard plotline command, which creates a line, based on a given input. Note that in this case, the numbers given are not the coordinates on the chart, but instead specify a number from which to start and where to end. The resulting line is made as division between x and y axis.**

## Plot functions

Before writing the whole script it's useful to find out what the iterations will look like by printing the result of iterations of given element. It can be done by writing the for-loop as below only with **print(man)** instead of **plt.hist**.

|  |
| --- |
| **for man in df.Name.cat.categories:  plt.hist(df[df.Name==man].Age, bins=5, label=man, stacked = True)** |

This displays histogram of age for every name category in dataframe. The column it uses obviously must be first converted to 'category' data-type, if not done already. The last histogram will be displayed at the very front. **stacked** won't work in script above, because it's passed as part of invidual charts instead of one big containing all of them.

|  |
| --- |
| **genrelist=[], labels=[] for man in df.Name.cat.categories:  genrelist.append(df[df.Name==man].Age)  labels.append(man) plt.hist(genrelist, bins=5, label=labels, stacked=True) plt.legend()** |

This histogram uses lists, created by for-loop, which enables stacking. It's actually not necessary to create a second list for labels, as the same result can be achieved by **label=df.Name.cat.categories**.

|  |  |  |
| --- | --- | --- |
| Numeric | **1, 0.5** | **numeric()** |
| Integer | **5L, as.integer(1.4)** | **int()** |
| Logical | **TRUE, FALSE** | **logical()** |
| Character | '**one'** | **character()** |
| Complex | **1+2i** | **complex()** |
| Vector | **c(1,7), c('one'), c(1.3:5.1)** | **c()** |
| List | **list(1, 'one', TRUE, c('jan, 'feb'))** | **list()** |
| Matrix | ***-see Matrices-*** | **matrix()** |
| DataFrames | ***-see Data Frames-*** | **dataframe()** |
| Array | **array(1:24, dim=c(3,4,2))** | **array()** |

R

R - **1. ebt ← numeric()   
2. for a in seq(1, length(revenue)){  
3. ebt ← c(ebt, (revenue – expenses))  
4. print(ebt)**

R can however do it simply by **revenue – expenses** as it's designed to automatically iterate through all elements in the list.

**<Akuli>if thething[0] has a clear meaning, as in "it's the x coordinate", then go with tuple**

**<Akuli> if it's just a list of mine locations, could have different lengths etc, then it should be a list**