SoftPython

Introductive guide to data cleaning and analysis with Python 3

David Leoni, Alessio Zamboni, Marco Caresia

Oct 25, 2020

Copyright © 2020 by David Leoni, Alessio Zamboni, Marco Caresia.

SoftPython is available under the Creative Commons Attribution 4.0 International License, granting you the right to copy, redistribute, modify, and sell it, so long as you attribute the original to David Leoni, Alessio Zamboni, Marco Caresia and identify any changes that you have made. Full terms of the license are available at:

http://creativecommons.org/licenses/by/4.0/

The complete book can be found online for free at:

https://en.softpython.org

CONTENTS

Preface

Introductive guide to coding, data cleaning and analysis for Python 3, with many worked exercises.

> **WARNING: THIS ENGLISH VERSION IS IN-PROGRESS**
>
> Completion is due by end of 2020
>
> Complete Italian version is here: it.softpython.org[1]

Nowadays, more and more decisions are taken upon factual and objective data. All disciplines, from engineering to social sciences, require to elaborate data and extract actionable information by analysing heterogenous sources. This book of practical exercises gives an introduction to coding and data processing using Python[2], a programming language popular both in the industry and in research environments.

News

October 3, 2020: updated *References* page

Old news: link

[1] https://it.softpython.org
[2] https://www.python.org

CHAPTER ONE

OVERVIEW

1.1 Intended audience

This book can be useful for both novices who never really programmed before, and for students with more techical background, who a desire to know about about data extraction, cleaning, analysis and visualization (among used frameworks there are Pandas, Numpy and Jupyter editor). We will try to process data in a practical way, without delving into more advanced considerations about algorithmic complexity and data structures. To overcome issues and guarantee concrete didactical results, we will present step-by-step tutorials.

1.2 Contents

- *Overview*: Approach and goals
- *References*

1.2.1 A - Foundations

1. *Installation*
2. *Tools and scripts*
3. *Basics*
4. Strings
    1. *introduction*
    2. *operators*
    3. *methods*
5. Lists
    1. *introduction*
    2. *operators*
    3. *methods*
6. *Tuples*
7. *Sets*
8. Dictionaries
    1. *introduction*

    2. *operators*
    3. *methods*
9. Control flow
    1. *if conditionals*
    2. *for loops*
    3. *while loops*
10. *Sequences and comprehensions*
11. *Matrices: lists of lists*
12. *Error handling and testing*

1.2.2 B - Data analysis

1. *Data formats (CSV, JSON)*
    - common data formats presentation (CSV, JSON, binary files)
    - Data conversion, errors and ill-formatted data
2. *Graph formats*
    - graph matrices, adjacency lists, networkx
3. *Binary relations*

1.3 Authors

David Leoni (main author): Software engineer specialized in data integration and semantic web, has made applications in open data and medical in Italy and abroad. He frequently collaborates with University of Trento for teaching activities in various departments. Since 2019 is president of CoderDolomiti Association, where along with Marco Caresia manages volunteering movement CoderDojo Trento to teach creative coding to kids. Email: david.leoni@unitn.it Website: davidleoni.it[3]

Marco Caresia (2017 Autumn Edition assistent @DISI, University of Trento): He has been informatics teacher at Scuola Professionale Einaudi of Bolzano. He is president of the Trentino Alto Adige Südtirol delegatioon of the Associazione Italiana Formatori and vicepresident of CoderDolomiti Association.

Alessio Zamboni (2018 March Edition assistent @Sociology Department, University of Trento): Data scientist and software engineer with experience in NLP, GIS and knowledge management. Has collaborated to numerous research projects, collecting experinces in Europe and Asia. He strongly believes that *'Programming is a work of art'*.

Massimiliano Luca (2019 summer edition teacher @Sociology Department, University of Trento): Loves learning new technologies each day. Particularly interested in knowledge representation, data integration, data modeling and computational social science. Firmly believes it is vital to introduce youngsters to computer science, and has been mentoring at Coder Dojo DISI Master.

[3] https://davidleoni.it

1.4 License

The making of this website and related courses was funded mainly by Department of Information Engineering and Computer Science (DISI)[4], University of Trento, and also Sociology[5] and Mathematics[6] departments.

All the material in this website is distributed with license CC-BY 4.0 International Attribution https://creativecommons.org/licenses/by/4.0/deed.en

Basically, you can freely redistribute and modify the content, just remember to cite University of Trento and the authors[7]

Technical notes: all website pages are easily modifiable Jupyter notebooks, that were converted to web pages using NB-Sphinx[8] using template Jupman[9]. Text sources are on Github at address https://github.com/DavidLeoni/softpython-en

1.5 Acknowledgments

We thank in particular professor Alberto Montresor of Department of Information Engineering and Computer Science, University of Trento to have allowed the making of first courses from which this material was born from, and the project Trentino Open Data (dati.trentino.it[10]) for the numerous datasets provided.

Other numerous intitutions and companies that over time contributed material and ideas are cited in this page

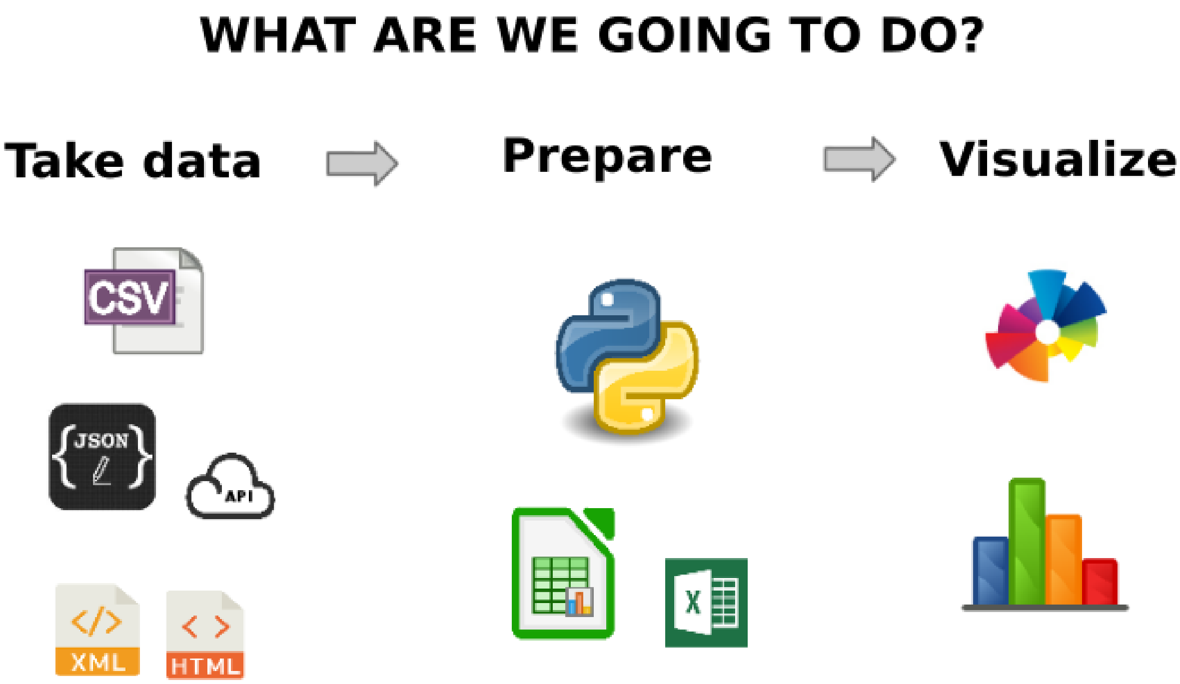
[4] https://www.disi.unitn.it
[5] https://www.sociologia.unitn.it/en
[6] https://www.maths.unitn.it/en
[7] https://en.softpython.org/index.html#Authors
[8] https://nbsphinx.readthedocs.io
[9] https://github.com/DavidLeoni/jupman
[10] https://dati.trentino.it

CHAPTER

TWO

OVERVIEW

To start with we will spend a couple of words on the approach and the goals of the book, then we will deep dive into the code.



2.1 Chapters

The tutorial mostly deal with fundamentals of PYthon 3, data analysis (intended more like raw data processing than statistics) and some applications (dashboard, database, ..)

What are *not* about:

- object oriented programming theory
- algorithms, computational complexity
- performance
  - no terabytes of data …
- advanced debugging (pdb)

- testing is only mentioned
- machine learning
- web development is only mentioned

2.2 Why Python?

- **Easy** enough to start with
- **Versatile**, very much used for
  - scientific calculus
  - web applications
  - scripting
- **widespread** both in the industry and research environments
  - Tiobe[11] Index
  - popularity on Github[12]
- **Licence** open source & business friendly[13]
  - translated: you can sell commercial products based on Python without paying royalties to its authors

[11] https://www.tiobe.com/tiobe-index/
[12] https://madnight.github.io/githut/#/pull_requests/2020/1
[13] https://docs.python.org/3/license.html

2.3 Approach and goals

If you have troubles with programming basics:

- **Exercise difficulty**: ✪ , ✪✪
- Read SoftPython - Parte A - Foundations[14]

If you already know how to program:

- **Exercise difficulty**: ✪✪✪, ✪✪✪✪
- Read Python Quick Intro[15] and then go directly to Part B - Data Analysis

Other guides: you can find links to further material in *References* page

2.4 Doesn't work, what should I do?

While programming you will surely encounter problems, and you will stare at mystierious error messages on the screen. The purpose of this book is not to give a series of recipes to learn by heart and that always work, as much as guide you moving first steps in Python world with some ease. So, if something goes wrong, do not panic and try following this list of steps that might help you. Try following the proposed order:

1. If in class, ask professor (if not in class, see last two points).
2. If in class, ask the classmate who knows more
3. Try finding the error message on Google
    - remove names or parts too specific of your program, like line numbers, file names, variable names
    - Stack overflow[16] is your best friend
4. Look at Appendix A - Debug from the book Think Python[17]
    - Syntax errors[18]
        - I keep making changes and it makes no difference.[19]
    - Runtime errors[20]
        - My program does absolutely nothing.[21]
        - My program hangs.[22]
        - Infinite Loop[23]
        - Infinite Recursion[24]
        - Flow of Execution[25]

[14] https://en.softpython.org/index.html#A---Foundations
[15] https://en.softpython.org/quick-intro/quick-intro-sol.html
[16] https://stackoverflow.com
[17] http://greenteapress.com/thinkpython2/html/thinkpython2021.html
[18] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#235
[19] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec236
[20] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec237
[21] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec238
[22] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec239
[23] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec240
[24] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec2241
[25] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec242

    - When I run the program I get an exception[26]
    - I added so many print statements I get inundated with output[27]
  - Semantic errors[28]
    - My program doesn't work[29]
    - ve got a big hairy expression and it doesn't do what I expect.[30]
    - ve got a function that doesn't return what I expect.[31]
    - I'm really, really stuck and I need help.[32]
    - No, I really need help.[33]

5. Gather some courage and ask on a public forum, like Stack overflow or python-forum.io - see *how to ask questions*.

2.4.1 How to ask questions

IMPORTANT

If you want to ask written questions on public chat/forums (i.e. like python-forum.io[34] DO FIRST READ the forum rules (see for example How to ask Smart Questions[35]

In substance, you are always asked to clearly express the problem circumstances, putting an explicative title to the post /mail and showing you spent some time (at least 10 min) trying a solution on your own. If you followed the above rules, and by misfortune you still find programmers who use harsh tones, just ignore them.

2.5 Installation and tools

- If you still haven't installed Python3 and Jupyter, have a look at *Installation*

2.6 Let's start !

- **If you already have some programming skill**: you can look Python quick start
- **If you don't have programming skills**: got to Tools and scripts[36]

[26] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec243
[27] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec244
[28] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec245
[29] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec246
[30] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec247
[31] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec248
[32] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec249
[33] http://greenteapress.com/thinkpython2/html/thinkpython2021.html#sec250
[34] https://python-forum.io/index.php
[35] https://python-forum.io/misc.php?action=help&hid=19
[36] https://en.softpython.org/tools/tools-sol.html

CHAPTER THREE

REFERENCES

3.1 Allen Downey - Think Python

Talks a lot, step by step, good for beginners

- online[37]
- printed[38]
- PDF[39]
- Interactive edition[40]

License: Creative Commons CC BY Non Commercial 3.0[41] as reported in the original page[42]

3.2 W3Resources website

Contains many simple exercises on Python basics, do them!

- Basic1[43], Basic2[44], String[45], List[46], Dictionary[47], Tuple[48], Sets[49], Condition Statements and Loops[50], Functions[51],Lambda[52], CSV Read Write[53]

[37] http://www.greenteapress.com/thinkpython/html/
[38] https://www.amazon.it/Think-Python-Like-Computer-Scientist/dp/1491939362/ref=sr_1_1?ie=UTF8&qid=1537163819&sr=8-1&keywords=think+python
[39] http://greenteapress.com/thinkpython2/thinkpython2.pdf
[40] https://runestone.academy/runestone/static/thinkcspy/index.html
[41] http://creativecommons.org/licenses/by-nc/3.0/deed.it
[42] http://greenteapress.com/wp/think-python-2e/
[43] https://www.w3resource.com/python-exercises/
[44] https://www.w3resource.com/python-exercises/basic/
[45] https://www.w3resource.com/python-exercises/string/
[46] https://www.w3resource.com/python-exercises/list/
[47] https://www.w3resource.com/python-exercises/dictionary/
[48] https://www.w3resource.com/python-exercises/tuple/
[49] https://www.w3resource.com/python-exercises/sets/
[50] https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php
[51] https://www.w3resource.com/python-exercises/python-functions-exercises.php
[52] https://www.w3resource.com/python-exercises/lambda/index.php
[53] https://www.w3resource.com/python-exercises/csv/index.php

3.3 Software Carpentry

Software Carpentry[54] is a website full of free educational resources, there is definitely a lot of good stuff to discover.

We highlight these tutorials:

- Programming with Python[55]: Nice tutorial with many exercises about processing a csv with topics: python basics, numpy, csv
- Plotting and programming with Python[56] More advanced, uses pandas

You may find other stuff in Community Developed Lessons for Jupyter[57] and Python[58]

3.4 LeetCode

Website with collections of exercises sorted by difficulty and acceptance rate. You can generally try sorting by *Acceptance* and *Easy* filters.

- leetcode.com[59]

We put here a selection.

3.4.1 LeetCode - Strings

Check string problems[60] sorted by *Acceptance* and *Easy*. In particular:

- Shuffle Strings[61]
- Increasing Decreasing String[62]
- Detect Capital[63]
- Unique email addresses[64]
- Robot return to origin[65]
- String matching in an Array[66]
- Reverse Words in a String III[67]
- Unique Morse codes[68]
- Goat Latin[69]

[54] https://software-carpentry.org/lessons/
[55] https://swcarpentry.github.io/python-novice-inflammation/
[56] https://swcarpentry.github.io/python-novice-gapminder/
[57] https://carpentries.org/community-lessons/#jupyter-notebook
[58] https://carpentries.org/community-lessons/#python
[59] https://leetcode.com
[60] https://leetcode.com/tag/string/
[61] https://leetcode.com/problems/shuffle-string/
[62] https://leetcode.com/problems/increasing-decreasing-string/
[63] https://leetcode.com/problems/detect-capital/description/
[64] https://leetcode.com/problems/unique-email-addresses/description/
[65] https://leetcode.com/problems/robot-return-to-origin/description/
[66] https://leetcode.com/problems/string-matching-in-an-array/
[67] https://leetcode.com/problems/reverse-words-in-a-string-iii/description/
[68] https://leetcode.com/problems/unique-morse-code-words/description/
[69] https://leetcode.com/problems/goat-latin/description/

- Count Binary Substrings[70]

3.4.2 LeetCode - Lists

Check array problems[71] sorted by *Acceptance* and *Easy*. In particular:

- Average Salary Excluding the Minimum and Maximum Salary[72]
- Contains Duplicate[73]
- Majority Element[74]
- Maximum Gap[75]
- Can Make Arithmetic Progression From Sequence[76]
- Max consecutive ones[77]
- Missing number[78] - has many possible solutions
- Move Zeros[79]
- K Closest Points to Origin[80] (use lambda functions[81])
- Rotated Digits[82]
- Filter Restaurants by Vegan-Friendly, Price and Distance[83] (to sort use lambda functions[84])
- Largest Perimeter Triangle[85] hint: you don't actually need to try many combinations …
- H-Index[86]
- Sort array by parity 1[87]
- Sort array by parity 2[88]
- Relative sort array[89]
- Insert Intervals[90] (use lambda functions[91])
- Merge Intervals[92] (use lambda functions[93])

[70] https://leetcode.com/problems/count-binary-substrings/description/
[71] https://leetcode.com/tag/array/
[72] https://leetcode.com/problems/average-salary-excluding-the-minimum-and-maximum-salary/
[73] https://leetcode.com/problems/contains-duplicate/description/
[74] https://leetcode.com/problems/majority-element/description/
[75] https://leetcode.com/problems/maximum-gap/
[76] https://leetcode.com/problems/can-make-arithmetic-progression-from-sequence/
[77] https://leetcode.com/problems/max-consecutive-ones/description/
[78] https://leetcode.com/problems/missing-number/description/
[79] https://leetcode.com/problems/move-zeroes/description/
[80] https://leetcode.com/problems/k-closest-points-to-origin/
[81] https://docs.python.org/3/howto/sorting.html#key-functions
[82] https://leetcode.com/problems/rotated-digits/description/
[83] https://leetcode.com/problems/filter-restaurants-by-vegan-friendly-price-and-distance/
[84] https://docs.python.org/3/howto/sorting.html#key-functions
[85] https://leetcode.com/problems/largest-perimeter-triangle/
[86] https://leetcode.com/problems/h-index/
[87] https://leetcode.com/problems/sort-array-by-parity/description/
[88] https://leetcode.com/problems/sort-array-by-parity-ii/
[89] https://leetcode.com/problems/relative-sort-array/
[90] https://leetcode.com/problems/insert-interval/
[91] https://docs.python.org/3/howto/sorting.html#key-functions
[92] https://leetcode.com/problems/merge-intervals/
[93] https://docs.python.org/3/howto/sorting.html#key-functions

- Sort colors[94]
- Find all numbers disappeared in an array[95]
- Degree of an array[96]
- The k Strongest Values in an Array[97] a bit convoluted but doable
- Array partition 1[98] actually a bit hard but makes you think
- Distant Barcodes[99]
- Reorganize String[100] think first when the task is *not* possible, for the rest is like previous one

3.4.3 LeetCode - Sets and Dictionaries

Check dictionary problems[101] sorted by *Acceptance* and *Easy*.

Note: Keep in mind these problems are in section *dictionaries* for good reason: in order to execute fast they often require you to preprocess the data by indexing in it in some way, like i.e. putting strings in a set or as keys in a dicitonary so you can later look them up very fast.

WARNING: if you feel the need to use nested cycles, or search methods on lists/strings like `.index`, `.find`, `in` operator, `.count`, `.replace` on strings, try thinking first whether it is really necessary or you might use the above mentioned preprocessing instead.

Check in particular:

- Replace words[102]
- Word break[103]
- Fair candy swap[104]
- Verifying an alien dictionary[105] Note: you can use lambda functions[106], but it is not strictly necessary
- Least Number of Unique Integers after K Removals[107]
- People Whose List of Favorite Companies Is Not a Subset of Another List[108]

[94] https://leetcode.com/problems/sort-colors/
[95] https://leetcode.com/problems/find-all-numbers-disappeared-in-an-array/description/
[96] https://leetcode.com/problems/degree-of-an-array/description/
[97] https://leetcode.com/problems/the-k-strongest-values-in-an-array/
[98] https://leetcode.com/problems/array-partition-i/description/
[99] https://leetcode.com/problems/distant-barcodes/
[100] https://leetcode.com/problems/reorganize-string/
[101] https://leetcode.com/problemset/all/?search=dictionaries
[102] https://leetcode.com/problems/replace-words/
[103] https://leetcode.com/problems/word-break/
[104] https://leetcode.com/problems/fair-candy-swap/
[105] https://leetcode.com/problems/verifying-an-alien-dictionary/
[106] https://docs.python.org/3/howto/sorting.html#key-functions
[107] https://leetcode.com/problems/least-number-of-unique-integers-after-k-removals/
[108] https://leetcode.com/problems/people-whose-list-of-favorite-companies-is-not-a-subset-of-another-list/

3.5 LeetCode - Matrices

- Matrix Diagonal Sum[109]
- Cells with odd values in a matrix[110]
- Count negative numbers in Sorted matrix[111]
- Lucky Numbers in a Matrix[112]
- The k-weakest rows in a Matrix[113] (use lambda functions[114]
- Matrix Cells in Distance Order[115]
- Toepliz Matrix[116]
- Special Positions in a Binary Matrix[117]
- Reshape the Matrix[118]
- Kth Smallest Element in a Sorted Matrix[119] - there are many possible optimizations, you can make a first version using `sort` on everything, and then think about improving the algorithm
- Set Matrix Zeroes[120] interesting, try avoiding duplicating the matrix
- Search a 2D Matrix[121]
- Search a 2D Matrix ii[122]
- Spiral Matrix[123]
- Spiral Matrix ii[124]
- Matrix Block Sum[125]
- Sort the Matrix Diagonally[126] not fun, but doable

[109] https://leetcode.com/problems/matrix-diagonal-sum/
[110] https://leetcode.com/problems/cells-with-odd-values-in-a-matrix/
[111] https://leetcode.com/problems/count-negative-numbers-in-a-sorted-matrix/
[112] https://leetcode.com/problems/lucky-numbers-in-a-matrix/
[113] https://leetcode.com/problems/the-k-weakest-rows-in-a-matrix
[114] https://docs.python.org/3/howto/sorting.html#key-functions
[115] https://leetcode.com/problems/matrix-cells-in-distance-order/
[116] https://leetcode.com/problems/toeplitz-matrix/
[117] https://leetcode.com/problems/special-positions-in-a-binary-matrix/
[118] https://leetcode.com/problems/reshape-the-matrix/
[119] https://leetcode.com/problems/kth-smallest-element-in-a-sorted-matrix/
[120] https://leetcode.com/problems/set-matrix-zeroes/
[121] https://leetcode.com/problems/search-a-2d-matrix/
[122] https://leetcode.com/problems/search-a-2d-matrix-ii/
[123] https://leetcode.com/problems/spiral-matrix/
[124] https://leetcode.com/problems/spiral-matrix-ii/
[125] https://leetcode.com/problems/matrix-block-sum/
[126] https://leetcode.com/problems/sort-the-matrix-diagonally/

3.6 HackerRank

Contains many Python 3 exercises on algorithms and data structures (Needs to login)

- hackerrank.com[127]

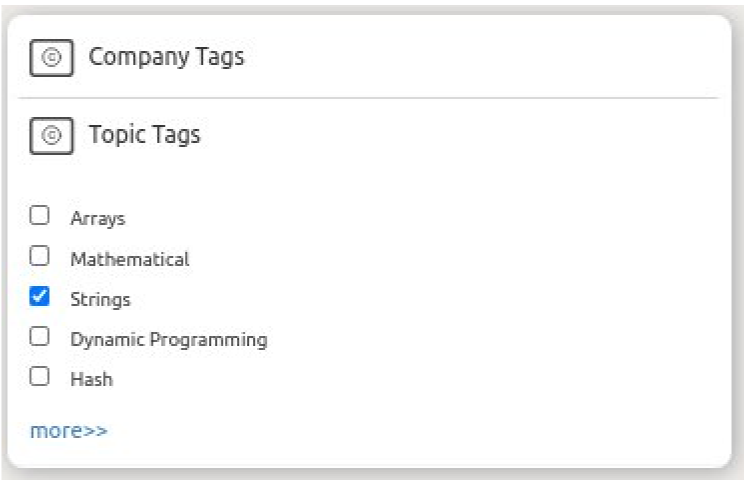
3.7 Geeks for Geeks

Contains many exercises - doesn't have solutions nor explicit asserts but if you login and submit solutions, the system will run some tests serverside and give you a response.

In general for Part A you can filter difficulty by school+basic+easy and if you need to do part B also include medium.

- Example: Filter difficulty by school+basic+easy and topic String[128]

You can select many more topics if you click `more>>` un der Topic Tags:



3.8 Dive into Python 3

More practical, contains more focused tutorials (i.e. manage XML files)

- online version[129]
- printed[130]
- zip offline[131]
- PDF[132]

Licence: Creative Commons By Share-alike 3.0[133] as reported at the bottom of book website[134]

[127] https://www.hackerrank.com
[128] https://practice.geeksforgeeks.org/explore/?category%5B%5D=Strings&difficulty%5B%5D=-2&difficulty%5B%5D=-1&difficulty%5B%5D=0&page=1
[129] http://www.diveintopython3.net/
[130] http://www.amazon.com/gp/product/1430224150?ie=UTF8&tag=diveintomark-20&creativeASIN=1430224150
[131] https://github.com/diveintomark/diveintopython3/zipball/master
[132] https://github.com/downloads/diveintomark/diveintopython3/dive-into-python3.pdf
[133] http://creativecommons.org/licenses/by-sa/3.0/
[134] http://www.diveintopython3.net/

3.9 Introduction to Scientific Programming with Python

Focuses on numerical calculations, you can check first 7 chapters until dictionaries.

By Joakim Sundnes.

- PDF[135] for Python (only theory)
- Exercises[136] – a LOT of stuff, although some exercises are too much into engineering / maths compared to this book
- EXTRA: if you like, it also contains chapters on classes which are certainly useful.

```
[ ]:
```

[135] https://link.springer.com/content/pdf/10.1007%2F978-3-030-50356-7.pdf
[136] https://www.uio.no/studier/emner/matnat/ifi/INF1100/h16/ressurser/INF1100_exercises_5th_ed.pdf

CHAPTER

FOUR

INSTALLATION

We will see whether and how to install Python, additional Python libraries, Jupyter notebook and finally how to manage virtual environments.

Sometimes you don't even need to install!

If you want, you can also directly program online with the following services

NOTE 1: if you want to try one, always remember to check it is using Python **3** !

NOTE 2: As for any online service, whenever it is freely offered do not abuse it. If you try processing a terabyte of data per day without paying a subscription, you risk a denial of service.

Python 3 su repl.it[137]: allows to edit Python code collaboratively with other users, and also supports libraries such as Matplotlib

Python Tutor[138]: allows to execute one instruction at a time while offering a very useful visualization of what is happening 'under the hood'

Google Colab[139]: allows editing collaboratively Jupyter notebooks and save them to Google Drive.

- NOTE 1: it might be you won't be able to access with university accounts (i.e.``@studenti.unitn.it``). In that case, use personal accounts such as `@gmail.com`
- NOTE 2: the 'collaborative' aspect of Colab changed over time, **be very careful at what happens when working in two people over the same document**. Once (2017) changes performed by one were immediately seen by other users, but lately (2019) they seem only visibly when saving - even worse, they overwrite changes others could have done in the meanwhile.

Online Jupyter demo[140]: sometimes it works but it is not always available. If you manage to access, remember to select from the menu *Kernel->Change kernel->Python 3*

[137] https://repl.it/languages/python3
[138] http://pythontutor.com/visualize.html#py=3
[139] https://colab.research.google.com
[140] http://try.jupyter.org

4.1 Installing Python

There are varioues ways to install Python 3 and its modules: there is the official 'plain' Python distribution but also package managers (i.e. Anaconda) or preset environments (i.e. Python(x,y)) which give you Python plus many packages. Once completed the installation, Python 3 contains a command `pip` (sometimes called `pip3` in Python 3), which allows to install afterwards other packages you may need.

The best way to choose what to install depends upon which operating system you have and what you intend to do with it. In this book we will use Python 3 and scientific packages, so we will try to create an environment to support this scenario.

Attention: before installing random stuff from the internet, read carefully this guide

We tried to make it generic enough, but we couldn't test all various cases so problems may arise depending on your particular configuration.

Attention: do not mix different Python distribution for the same version !

Given the wide variety of installation methods and the fact Python is available in laready many programs, it might be you already have installed Python without even knowing it, maybe in version 2, but we need the 3! Overlayinh several Python environments with the same version may cause problems, so in case of doubt ask somebody who knows more!

4.1.1 Windows installation

For Windows, we suggest to install the distribution Anaconda for Python 3.8[141] or greater, which, along with the native Python package manager `pip`, also offers the more generic command line package manager `conda`.

Once installed, verify it is working like this:

1. click on the Windows icon in the lower left corner and search for 'Anaconda Prompt'. It should appear a console where to insert commands, with written something like `C:\Users\David>`. NOTE: to launch Anaconda commands, only use this special console. If you use the default Windows console (`cmd`), Windows will not be able to find Python.
2. In Anaconda console, type:

```
conda list
```

It should appear a list of installed packages, like

```
# packages in environment at C:\Users\Jane\AppData\Local\Continuum\Anaconda3:
#
alabaster                 0.7.7                    py35_0
anaconda                  4.0.0               np110py35_0
anaconda-client           1.4.0                    py35_0
...
numexpr                   2.5                 np110py35_0
numpy                     1.10.4                   py35_0
odo                       0.4.2                    py35_0
...
yaml                      0.1.6                         0
zeromq                    4.1.3                         0
zlib                      1.2.8                         0
```

141 https://www.anaconda.com/download/#windows

3. Try Python3 by typing in the Anaconda console:

```
C:>  python
```

It should appear something like:

```
Python 3.6.3 (default, Sep 14 2017, 22:51:06)
MSC v.1900 64 bit (Intel)[GCC 5.4.0 20160609] on win64
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Attention: with Anaconda, you must write `python` instead of `python3` !

If you installed Anaconda for Python3, it will automatically use the correct Python version by simply writing `python`. If you write `python3` you will receive an error of file not found !

Attention: if you have Anaconda, always use `conda` to install Python modules ! So if in next tutorials you se written `pip3 install whatever`, you will instead have to use `conda install whatever`

4.1.2 Mac installation

To best manage installed app on Mac independently from Python, usually it is convenient to install a so called *package manager*. There are various, and one of the most popular is Homebrew[142]. So we suggest to first install Homebrew and then with it you can install Python 3, plus eventually other components you might need. As a reference, for installation we took and simplified this guide by Digital Ocean[143]

Attention: check if you already have a package manager !

If you already have installed a package manager like for example Conda (in *Anaconda* distribution), *Rudix, Nix, Pkgsrc, Fink,* o *MacPorts,* maybe Homebrew is not needed and it's better to use what you already have. In these cases, it may be worth asking somebody who knows more ! If you already have *Conda/Anaconda*, it can be ok as long as it is for Python 3.

— 1 Open the Terminal

MacOS terminal is an application you can use to access command line. As any other application, it's available in *Finder*, navigation in *Applications* folder, and the in the folder *Accessories*. Frotm there, double click on the *Terminal* to open it as any other app. As an alternative, you can use *Spotlight* by pressing *Command* and *Space* to find the Terminal typing the name in the bar that appears.

— 2 Install Homebrew by executing in the terminal this command:

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/
↪master/install)"
```

— 3 Add `/usr/local/bin` to `PATH`

In this passage with an unsettling name, once Homebrew installation is completed, you will make sure that apps installed with Homebrew shall always be used instead of those Mac OS X may automatically select.

142 https://brew.sh/
143 https://www.digitalocean.com/community/tutorials/how-to-install-python-3-and-set-up-a-local-programming-environment-on-macos

— 3.1 Open a new Terminal.

— 3.2 From within the terminal, digit the command

```
ls -a
```

You will see the list of all files present in the home folder. In these files, verify if a file exists with the following name: `.profile` (note the dot at the beginning):

- If it exists, go to following step
- If it doesn't exist, to create it type the following command:

  ```
  touch $HOME/.profile
  ```

— 3.3 Open with text edit the just created file `.profile` giving the command:

```
open -e $HOME/.profile
```

— 3.4 In text edit, add to the end of the file the following line:

```
export PATH=/usr/local/bin:$PATH
```

— 3.5 Save and close both Text Edit and the Terminal

— 4 Verify Homebrew is correctly installed, by typing in a new Terminal:

```
brew doctor
```

If there aren't updates to do, the Terminal should show:

```
Your system is ready to brew.
```

Otherwise, you might see a warning which suggest to execute another command like `brew update` to ensure the Homebrew installation is updated.

— 5. Install python3 (Remember the '3' !):

```
brew install python3
```

Along with python 3, Homebrew will also install the internal package manager of Python `pip3` which we will use in the following.

— 6 Verify Python3 is correctly installed. By executing this command the writing `"/usr/local/bin/python3"` should appear:

```
which python3
```

After this, try to launch

```
python3
```

You should see something similar:

```
Python 3.6.3 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on mac
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

To exit Python, type `exit()` and press Enter.

4.1.3 Linux installation

Luckily, all Linux distributions are already shipped with package managers to easily install applications.

- If you have Ubuntu:
  1. follow the guide of Dive into Python 3, chapter 0 - Installare Python[144] in particular by going to the subsection installing in Ubuntu Linux[145]
  2. after completing the guide, install also `python3-venv`:

     ```
     sudo apt-get install python3-venv
     ```

- If you *don't* have Ubuntu, read this note[146] and/or ask somebody who knows more.

To verify the installation, try to run from the terminal

```
python3
```

You should see something like this:

```
Python 3.6.3 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

4.2 Installing packages

You can extend Python by installing several free packages. The best way to do it varies according to the operating system and the installed package manager.

ATTENTION: We will be using *system commands*. If you see `>>>` in the command line, it means you are inside Python interpreter and you must first exit: to do it, type `exit()` and press Enter.

In what follows, to check if everything is working, you can substitute `PACKAGENAME` with `requests` which is a module for the web.

If you have Anaconda:

- click on Windows icon in the lower left corner and search `Anaconda Prompt`. A console should appear where to instert commands, with something written like `C:\Users\David>`. (NOTE: to run commands in Anaconda, use only this special console. If you use the default Windows console (cmd), Windows, will not be able to find Python)
- In the console type `conda install PACKAGENAME`

If you have Linux/Mac open the Terminaland give this command (`--user` install in your home):

- `python3 -m pip install --user PACKAGENAME`
- **NOTE:** If you receive errors which tell you the command `python3` is not found, remove the `3` after `python`

144 https://diveintopython3.problemsolving.io/installing-python.html
145 https://diveintopython3.problemsolving.io/installing-python.html#ubuntu
146 https://diveintopython3.problemsolving.io/installing-python.html#other

INFO: there is also a system command `pip` (or `pip3` according to your system). You can directl call it with `pip install --user PACKAGENAME`

Instead, we install instead with commands like `python3 -m pip install --user PACKAGENAME` for uniformity and to be sure to install packages for Python 3 version

4.3 Jupyter Notebook

4.3.1 Run Jupyter notebook

A handy editor you can use for Python is Jupyter[147]:

- If you installed Anaconda, you should already find it in the system menu and also in the Anaconda Navigator.
- If you didn't install Anaconda, try searching in the system menu anyway, maybe by chance it was already installed
- If you can't find it in the system menu, you may anyway from command line

Try this:

```
jupyter notebook
```

or, as alternative,

```
python3 -m notebook
```

ATTENTION: `jupyter` is NOT a Python command, it is a *system* command.

If you see written >>> on command line it means you must first exit Python insterpreter by writing 'exit()` and pressing Enter !

ATTENTION: If Jupyter is not installed you will see error messages, in this case don't panic and *go to installation*.

A browser should automatically open with Jupyter, and in the console you should see messages like the following ones. In the browser you should see the files of the folders from which you ran Jupyter.

If no browser starts but you see a message like the one here, then copy the address you see in an internet browser, preferebly Chrome, Safari or Firefox.

```
$ jupyter notebook
[I 18:18:14.669 NotebookApp] Serving notebooks from local directory: /home/da/Da/prj/
↪softpython/prj
[I 18:18:14.669 NotebookApp] 0 active kernels
[I 18:18:14.669 NotebookApp] The Jupyter Notebook is running at: http://localhost:
↪8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888
[I 18:18:14.669 NotebookApp] Use Control-C to stop this server and shut down all␣
↪kernels (twice to skip confirmation).
[C 18:18:14.670 NotebookApp]
```

(continues on next page)

[147] http://jupyter.org/

(continued from previous page)

```
    Copy/paste this URL into your browser when you connect for the first time,
    to login with a token:
        http://localhost:8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888
```

ATTENTION 1: in this case the address is `http://localhost:8888/?token=49d4394bac446e291c6ddaf349c9dbffcd2cdc8c848eb888`, but youts will surely be different!

ATTENTION 2: While Jupyter server is active, you can't put commands in the terminal !

In the console you see the server output of Jupyter, which is active and in certain sense 'it has taken control' of the terminal. This means that if you write some commands inside the terminal, these **will not** be executed!

4.3.2 Saving Jupyter notebooks

You can save the current notebook in Jupyter by pressing `Control-S` while in the browser.

ATTENTION: DO NOT OPEN THE SAME DOCUMENT IN MANY TABS !!

Be careful to not open the same notebook in more the one tab, as modifications in different tabs may overwrite at random ! To avoid these awful situations, make sure to have only one tab per document. If you accidentally open the same notebook in different tabs, just close the additional tab.

Automated savings

Notebook changes are automatically saved every few minutes.

4.3.3 Turning off Jupyter server

Before closing Jupyter server, remember to save in the browser the notebooks you modified so far.

To correctly close Jupyter, *do not* brutally close the terminal, Instead, from the the terminal where you ran Jupyter, hit `Control-c`, a question should appear to which you should answer `y` (if you don't answer in 5 seconds, you will have to hit `control-c` again).

```
Shutdown this notebook server (y/[n])? y
[C 11:05:03.062 NotebookApp] Shutdown confirmed
[I 11:05:03.064 NotebookApp] Shutting down kernels
```

4.3.4 Navigating notebooks

(Optional) To improve navigation experience in Jupyter notebooks, you may want to install some Jupyter extension, like `toc2` which shows paragraph headers in the sidebar. To install:

Install the Jupyter contrib extensions[148]:

1a. If you have Anaconda: Open Anaconda Prompt, and type:

```
conda install -c conda-forge jupyter_contrib_nbextensions
```

1b. If you don't have Anaconda: Open the terminal and type:

```
python3 -m pip install --user jupyter_contrib_nbextensions
```

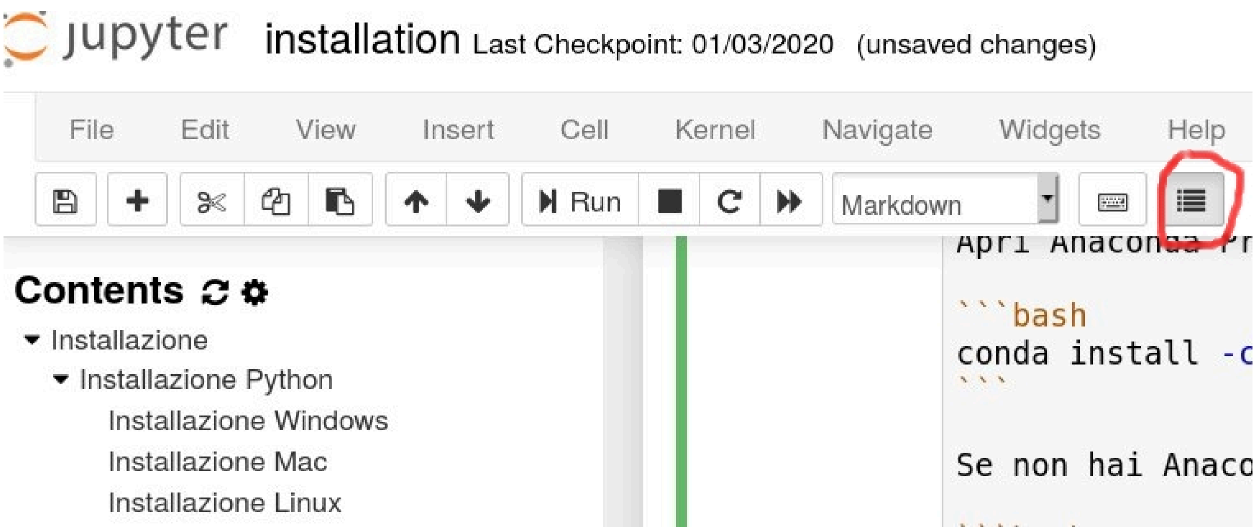
2. Install in Jupyter:

```
jupyter contrib nbextension install --user
```

3. Enable extensions:

```
jupyter nbextension enable toc2/main
```

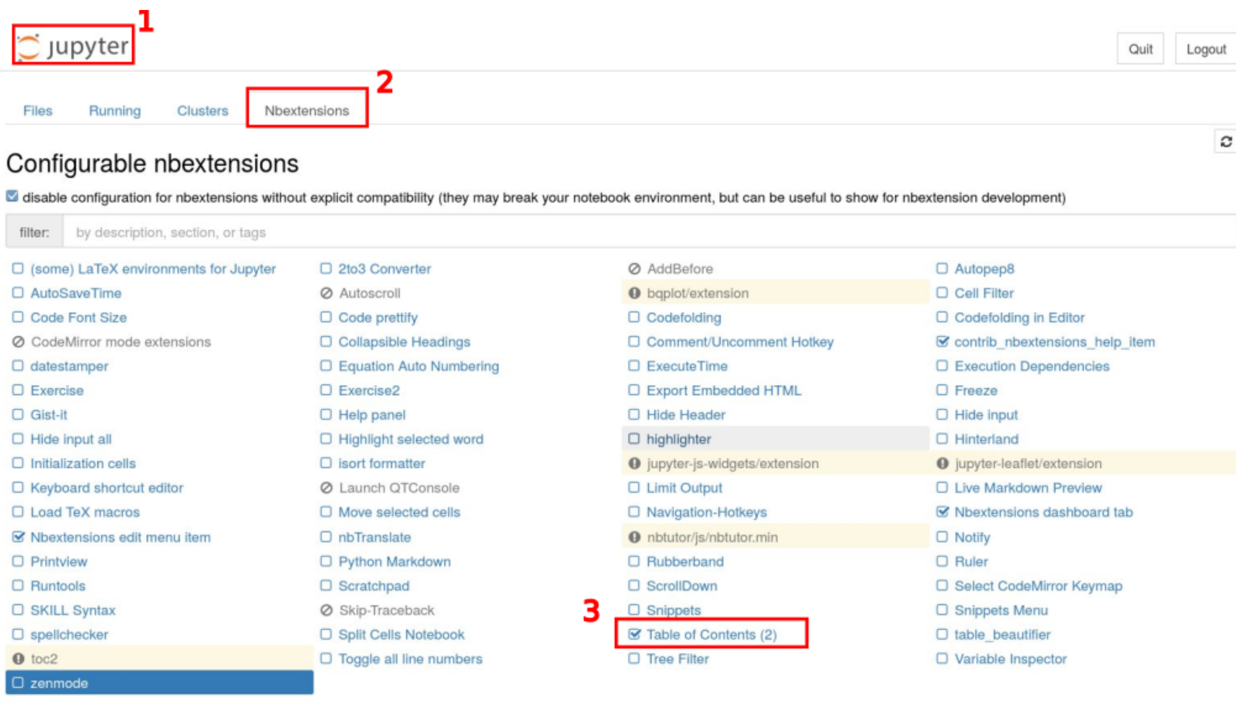
Once installed: To see table of contents in a document you will have to press a list button on the right side of the toolbar:



If by chance you don't see the button:

1. go to main Jupyter interface
2. check `Nbextensions` tab
3. make sure `Table of Contents(2)` is enabled
4. Close Jupyter, reopen it, go to a notebook, you should finally see the button

[148] https://github.com/ipython-contrib/jupyter_contrib_nbextensions



4.3.5 Installing Jupyter notebook - all operating systems

If you didn't manage to *find and/or start Jupyter*, probably it means we need to install it!

You may try installing Jupyter with `pip` (the native package manager of Python)

To install, run this command:

```
python3 -m pip install --user jupyter -U
```

Once installed, follow the section

Una volta installato, segui la sezione *Run Jupyter Notebook*

ATTENTION: you DON'T need to install Jupyter inside *virtual environments* You can consider Jupyter as a system-level application, which should be independent from virtual environments. If you are inside a virtual environment (i.e. the command line begins with a writing in parenthesis like (myprj)``), exit the environment by typeing``deactivate`)

HELP: if you have trouble installing Jupyter, while waiting for help you can always try the online demo version[149] (note: it's not always available) or Google Colab[150]

[149] https://try.jupyter.org/
[150] http://colab.research.google.com/

4.4 Projects with virtual environments

> **WARNING: If these are your first steps with Python, you can skip this section.**
>
> You should read it if you have already done personal projects with Python that you want to avoid compromising, or when you want to make a project to ship to somebody.

When we start a new project with Python, we usually notice quickly that we need to extend Python with particular libraries, like for example to draw charts. Not only that, we might also want to install Python programs which are not written by us and they might as well need their peculiar libraries to work.

Now, we could install all these extra libraries in a unique cauldron for the whole computer, but each project may require its specific versions of each library, and sometimes it might not like versions already installed by other projects. Even worse, it might automatically update packages used by old projects, preventing old code from working anymore. So it is PRACTICALLY NECESSARY to separate well each project and its dependencies from those of other projects: for this purpose you can create a so-called *virtual environment* .

4.4.1 Creating virtual environments

- **If you installed Anaconda**, to create virtual environments you can use its package manager `conda`. Supposing we want to call our project `myprj` (but it could be any name), to put into a folder with the same name `myprj`, we can use this command to create a virtual environment:

  ```
  conda create -n myprj
  ```

  The command might require you to download packages, you can safely confirm.

- **If you *don't have* Anaconda installed**, to create virtual environments it's best to use the native Python module `venv`:

  ```
  python3 -m venv myprj
  ```

Both methods create the folder `myprj` and fill it with all required Python files to have a project completely isolated from the rest of the computer. But now, how can we tell Python we want to work right with that project? We must *activate* the environment as follows.

4.4.2 Activate a virtual environment

To activate the virtual environment, we must use different commands according to our operating system (but always from the terminal)

Activate environment in Windows with Anaconda:

```
activate myprj
```

Linux & Mac (without Anaconda):

```
source myprj/bin/activate
```

Once the environment is active, in the command prompt we should see the name of that environment (in this case `myprj`) between round parenthesis at the beginning of the row:

```
(myprj) some/current/folder >
```

The prefix lets us know that the environment `myprj` is currently active, so Python comands we will use all use the settings and libraries of that environment.

Note: inside the virtual environment, we can use the command `python` instead of `python3` and `pip` instead of `pip3`

Deactivate an environment:

Write in the console the command `deactivate`. Once the environment is deactivated, the environment name `(myprj)` at the beginning of the prompt should disappear.

4.4.3 Executing environments inside Jupyter

As we said before, Jupyter is a system-level application, so there should be one and only one Jupyter. Nevertheless, during Jupyter execution, we might want to execute our Python commands in a particular Python environment. To do so, we must configure Jupyter so to use the desired environment. In Jupyter terminology, the configurations are called *kernel*: they define the programs launched by Jupyter (be they Python versions or also other languages like R). The current kernel for a notebook is visibile in the right-upper corner. To select a desired kernel, there are several ways:

With Anaconda

Jupyter should be available in the Navigator. If in the Navigator you enable an environment (like for example Python 3), when you then launch Jupyter and create a notebook you should have the desired environment active, or at least be able to select a kernel with that environment.

Without Anaconda

In this case, the procedure is a little more complex:

— 1 From the terminal þactivate your environment](#Activate-a-virtual-environment)

— 2 Create a Jupyter kernel:

```
python3 -m ipykernel install --user --name myprj
```

NOTE: here `myprj` is the name of the *Jupter kernel*. We use the same name of the environment only for practical reasons.

— 3 Deactivate your environment, by launching

```
deactivate
```

From now on, every time you run Jupyter, if everything went well under the `Kernel` menu in the notebook you should be able to select the kernel just created (in this example, it should have the name `myprj`)

NOTE: the passage to create the kernel must be done only once per project

---

NOTE: you don't need to activate the environment before running Jupyter!

During the execution of Jupyter simply select the desired kernel. Nevertheless, it is convenient to execute Jupyter from the folder of our virtual environment, so we will see all the project files in the Jupyter home.

4.5 Further readings

Go on with the page Tools and scripts[151] to learn how to use other editors and Python architecture.

[151] https://en.softpython.org/tools/tools-sol.html

CHAPTER

FIVE

A - FOUNDATIONS

5.1 Tools and scripts

5.1.1 Download exercises zip

Browse files online[152]

REQUISITES:

- **Having Python 3 and Jupyter installed:** if you haven't already, see Installation[153]

5.1.2 Python interpreter

In these tutorials we will use extensively the notebook editor Jupyter, because it allows to comfortably execute Python code, display charts and take notes. But if we want only make calculations it is not mandatory at all!

The most immediate way (even if not very practical) to execute Python things is by using the *command line* interpreter in the so-called *interactive mode,* that is, having Python to wait commands which will be manually inserted one by one. This usage *does not* require Jupyter, you only need to have installed Python. Note that in Mac OS X and many linux systems like Ubuntu, Python is already installed by default, although sometimes it might not be version 3. Let's try to understand which version we have on our system.

Let's open system console

Open a console (in Windows: system menu -> Anaconda Prompt, in Mac OS X: run the Terminal)

In the console you find the so-called *prompt* of commands. In this *prompt* you can directly insert commands for the operating system.

WARNING: the commands you give in the prompt are commands in the language of the operating system you are using, **NOT** Python language !!!!!

In Windows you should see something like this:

[152] https://github.com/DavidLeoni/softpython-en/tree/master/tools
[153] https://en.softpython.org/installation.html

```
C:\Users\David>
```

In Mac / Linux it could be something like this:

```
david@my-computer:~$
```

Listing files and folders

In system console, try for example to

Windows: type the command `dir` and press Enter

Mac or Linux: type the command `ls` and press Enter.

A listing with all the files in the current folder should appear. In my case appears a list like this:

LET ME REPEAT: in this context `dir` and `ls` are commands of *the operating system*, **NOT** of Python !!

Windows:

```
C:\Users\David> dir
Arduino                   gotysc                          program.wav
a.txt                     index.html                      Public
MYFOLDER                  java0.log                       RegDocente.pdf
backupsys                 java1.log
BaseXData                 java_error_in_IDEA_14362.log
```

Mac / Linux:

```
david@david-computer:~$ ls
Arduino                          gotysc                          program.wav
a.txt                            index.html                      Public
MYFOLDER                         java0.log                      ␣
↪RegistroDocenteStandard(1).pdf
backupsys                        java1.log                      ␣
↪RegistroDocenteStandard.pdf
BaseXData                        java_error_in_IDEA_14362.log
```

Let's launch the Python interpreter

In the opened system console, simply type the command `python`:

WARNING: If Python does not run, try typing `python3` with the `3` at the end of `python`

```
C:\Users\David> python
```

You should see appearing something like this (most probably won't be exactly the same). Note that Python version is contained in the first row. If it begins with `2.`, then you are not using the right one for this book - in that case try exiting the interpreter (*see how to exit*) and then type `python3`

```
Python 3.5.2 (default, Nov 23 2017, 16:37:01)
[GCC 5.4.0 20160609] on windows
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

CAREFUL about the triple greater-than >>> at the beginning!

The triple greater-than >>> at the start tells us that differently from before now the console is expecting commands *in Python language*. So, the system commands we used before (`cd`, `dir`, …) will NOT work anymore, or will give different results!

Now the console is expecting Python commands, so try inserting `3 + 5` and press Enter:

WARNING DO NOT type >>>, only type the command which appears afterwards!

```
>>> 3 + 5
```

The writing `8` should appear:

```
8
```

Beyond calculations, we might tell PYthon to print something with the function `print("ciao")`

```
>>> print("ciao")
ciao
```

Exiting the interpreter

To get out from the Python interpreter and go back to system prompt (that is, the one which accepts `cd` and `dir`/`ls` commands), type the Python comand `exit()`

After you actually exited the Python interpreter, the triple >>> should be gone (you should see it at the start of the line)

In Windows, you should see something similar:

```
>>> exit()
C:\Users\David>
```

in Mac / Linux it could be like this:

```
>>> exit()
david@my-computer:~$
```

Now you might go back to execute commands for the operating system like `dir` and `cd`:

Windows:

```
C:\Users\David> dir
Arduino                    gotysc                            program.wav
a.txt                      index.html                        Public
MYFOLDER                   java0.log                         RegDocente.pdf
backupsys                  java1.log
BaseXData                  java_error_in_IDEA_14362.log
```

Mac / Linux:

```
david@david-computer:~$ ls
Arduino                          gotysc                         program.wav
a.txt                            index.html                     Public
MYFOLDER                         java0.log                      ␣
↪RegistroDocenteStandard(1).pdf
backupsys                        java1.log                      ␣
↪RegistroDocenteStandard.pdf
BaseXData                        java_error_in_IDEA_14362.log
```

5.1.3 Modules

Python Modules are simply text files which have the extension **.py** (for example `my_script.py`). When you write code in an editor, as a matter of fact you are implementing the corresponding module.

In Jupyter we use notebook files with the extension `.ipynb`, but to edit them you necessarily need Jupyter.

With `.py` files (alse said *script*) we can instead use any text editor, and we can then tell the interpreter to execute that file. Let's see how to do it.

Simple text editor

1. With a text editor (*Notepad* in Windows, or *TextEdit* in Mac Os X) creates a text file, and put inside this code

```
x = 3
y = 5
print(x + y)
```

2. Let's try to save it - it seems easy, but it is often definitely not, so read carefully!

WARNING: when you are saving the file, **make sure the file have the extension** `.py` **!!**

Let's suppose to create the file `my_script.py` inside a folder called `MYFOLDER`:

- **WINDOWS**: if you use *Notepad*, in the save window you have to to set *Save as* to *All files* (otherwise the file will be wrongly saved like `my_script.py.txt` !)
- **MAC**: if you use *TextEdit,* before saving click *Format* and then *Convert to format Only text:* **if you forget this passage, TextEdit in the save window will not allow you to save in the right format and you will probably end up with a file .rtf which we are not interested in**

3. Open a console (in Windows: system menu -> Anaconda Prompt, in Mac OS X: run the Terminal)

the console opens the so-called *commands prompt*. In this *prompt* you can directly enter commands for the operating system (see *previous paragraph*

WARNING: the commands you give in the prompt are commands in the language of the operating system you are using, **NOT** Python language !!!!!

In Windows you should see something like this:

```
C:\Users\David>
```

In Mac / Linux it could be something like this:

```
david@my-computer:~$
```

Try for example to type the command `dir` (or `ls` for Mac / Linux) which shows all the files in the current folder. In my case a list like this appears:

LET ME REPEAT: in this context `dir` / `ls` are commands of the *operating system,* **NOT** Python.

```
C:\Users\David> dir
Arduino                    gotysc                          program.wav
a.txt                      index.html                      Public
MYFOLDER                   java0.log                       RegDocente.pdf
backupsys                  java1.log
BaseXData                  java_error_in_IDEA_14362.log
```

If you notice, in the list there is the name MYFOLDER, where I put `my_script.py`. To *enter* the folder in the *prompt,* you must first use the operating system command `cd` like this:

4. To enter a folder called MYFOLDER, type `cd MYFOLDER`:

```
C:\Users\David> cd MYFOLDER
C:\Users\David\MYFOLDER>
```

What if I get into the wrong folder?

If by chance you enter the wrong folder, like `DUMBTHINGS`, to go back of one folder, type `cd ..` (NOTE: `cd` is followed by one space and TWO dots `..` *one after the other*)

```
C:\Users\David\DUMBTHINGS> cd ..
C:\Users\David\>
```

5. Mae sure to be in the folder which contains `my_script.py`. If you aren't there, use commands `cd` and `cd ..` like above to navigate the folders.

Let's see what present in MYFOLDER with the system command `dir` (or `ls` if in Mac/Linux):

LET ME REPEAT: inthis context `dir` (or `ls` is a command of the *operating system,* **NOT** Python.

```
C:\Users\David\MYFOLDER> dir

my_script.py
```

`dir` is telling us that inside `MYFOLDER` there is our file `my_script.py`

6. From within `MYFOLDER`, type `python my_script.py`

```
C:\Users\David\MYFOLDER>python my_script.py
```

WARNING: if Python does not run, try typing `python3 my_script.py` with `3` at the end of `python`

If everything went fine, you should see

```
8
C:\Users\David\MYFOLDER>
```

WARNING: After executing a script this way, the console is awaiting new *system* commands, **NOT** Python commands (so, there shouldn't be any triple greater-than >>>)

IDE

in these tutorial we work on Jupyter notebooks with extension `.ipynb`, but to edit long `.py` files it's more convenient to use more traditional editors, also called IDE *(Integrated Development Environment)*. For Python we can use Spyder[154], Visual Studio Code[155] or PyCharme Community Edition[156].

Differently from Jupyter, these editors allow more easily code *debugging* and *testing*.

Let's try Spyder, which is the easiest - if you have Anaconda, you find it available inside Anaconda Navigator.

INFO: Whenever you run Spyder, it might ask you to perform an upgrade, in these cases you can just click No.

In the upper-left corner of the editor there is the code of the file `.py` you are editing. Such files are also said *script*. In the lower-right corner there is the console with the IPython interpreter (which is the same at the heart of Jupyter, here in textual form). When you execute the script, it's like inserting commands in that interpreter.

- To execute the whole script: press `F5`
- To execute only the current line or the selection: press `F9`
- To clear memory: after many executions the variables in the memory of the interpreter might get values you don't expect. To clear the memory, click on the gear to the right of the console, and select *Restart kernel*

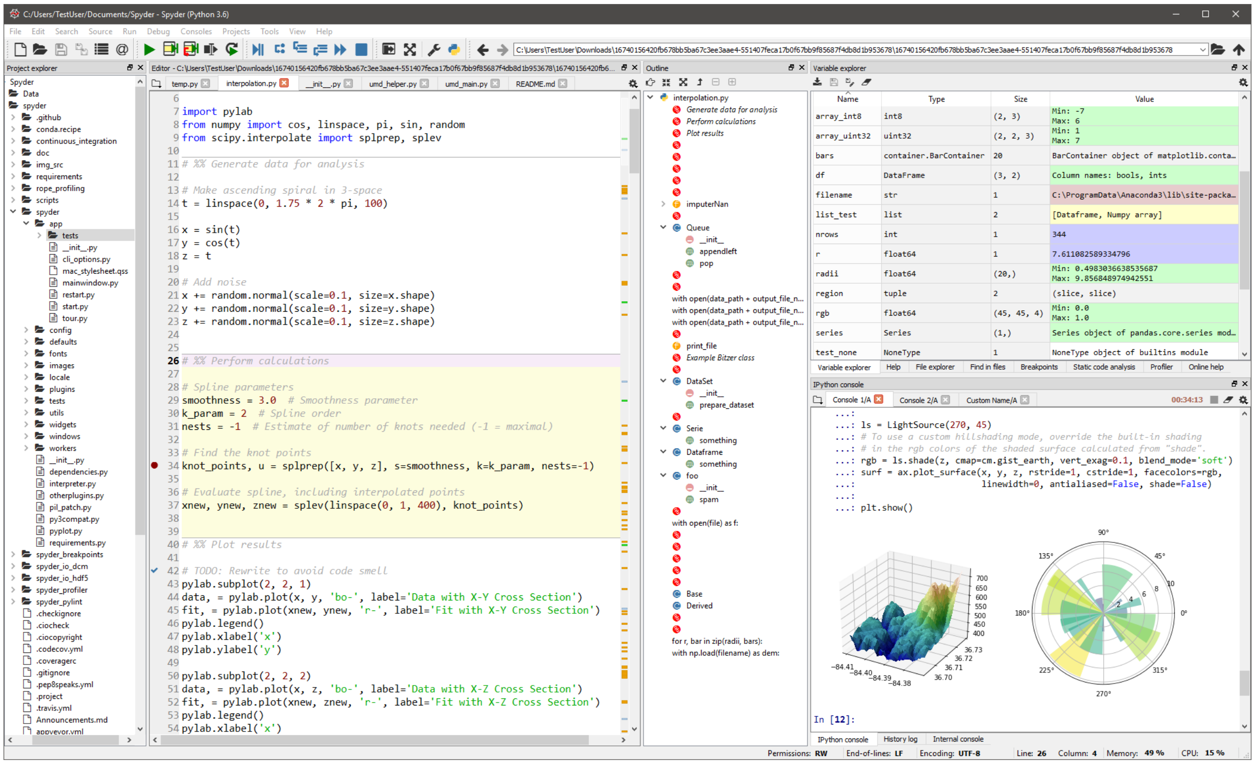
EXERCISE: do some test, taking the file `my_script.py` we created before:

```
x = 3
y = 5
print(x + y)
```

- once the code is in the script, hit `F5`
- select only `print(x+y)` and hit F9
- select only `x=3` and hit F9
- click on th gear the right of the console panel, and select *Restart kernel,* then select only `print(x+y)` and hit F9. What happens?

Remember that if the memory of the interpreter has been cleared with *Restart kernel,* and then you try executing a code row with variables defined in lines which were not exectued before, Python will not know which variables you are referring to and will show a `NameError`.

154 https://www.spyder-ide.org/
155 https://code.visualstudio.com/Download
156 https://www.jetbrains.com/pycharm/download/



5.1.4 Jupyter

Jupyter is an editor that allows to work on so called *notebooks,* which are files ending with the extension `.ipynb`. They are documents divided in cells where in each cell you can insert commands and immediately see the respective output. Let's try opening this.

1. Unzip exercises zip in a folder, you should obtain something like this:

```
tools
    tools-sol.ipynb
    tools.ipynb
    jupman.py
```

WARNING: To correctly visualize the notebook, it MUST be in the unzipped folder.

2. open Jupyter Notebook. Two things should appear, first a console and then a browser. In the browser navigate the files to reach the unzipped folder, and open the notebook `tools.ipynb`

WARNING: DO NOT click Upload button in Jupyer

Just navigate until you reach the file.

WARNING: open the notebook WITHOUT the `-sol` at the end!

Seeing now the solutions is too easy ;-)

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells.Exercises are graded by difficulty, from one star ⊛ to four ⊛⊛⊛⊛

> **WARNING: In this book we use ONLY PYTHON 3**
>
> If by chance you obtain weird behaviours, check you are using Python 3 and not 2. If by chance by typing `python` your operating system runs python 2, try executing the third by typing the command `python3`

If you don't find Jupyter / something doesn't work: have a look at installation[157]

Useful shortcuts:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- when something seem wrong in computations, try to clean memory by running `Kernel->Restart and Run all`

EXERCISE: Let's try inserting a PYthon command: type in the cell below here `3 + 5`, then while in that cell press special keys `Control+Enter`. As a result, the number `8` should appear

[ ]:

```

```

EXERCISE: with Python we can write comments by starting a row with a sharp `#`. Like before, type in the next cell `3 + 5` but this time type it in the row under the writing `# write here`:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[2]:

```python
# write here
```

</div>

[2]:

```python
# write here
```

EXERCISE: In every cell Jupyter only shows the result of last executed row. Try inserting this code in the cell below and execute by pressing `Control+Enter`. Which result do you see?

```python
3 + 5
1 + 1
```

[3]:

```python
# write here
```

EXERCISE: Let's try now to create a new cell.

- While you are with curson the cell, press `Alt+Enter`. A new cell should be created after the current one.

[157] https://en.softpython.org/installation.html#Jupyter-Notebook

- In the cell just created, insert `2+3` and press `Shift+Enter`. What happens to the cursor? Try the difference swith `Control+Enter`. If you don't understand the difference, try pressing many times `Shift+Enter` and see what happens.

Printing an expression

Let's try to assign an expression to a variable:

[4]:
```python
coins = 3 + 2
```

Note the assignment by itself does not produce any output in the Jupyter cell. We can ask Jupyter the value of the variable by simply typing again the name in a cell:

[5]:
```python
coins
```

[5]:
```
5
```

The effect is (almost always) the same we would obtain by explictly calling the function `print`:

[6]:
```python
print(coins)
```

```
5
```

What's the difference? For our convenience Jupyter will directly show the result of the last executed expression in the cell, but only the last one:

[7]:
```python
coins = 4
2 + 5
coins
```

[7]:
```
4
```

If we want to be sure to print both, we need to use the function `print`:

[8]:
```python
coins = 4
print(2 + 5)
print(coins)
```

```
7
4
```

Furthermore, the result of last expression is shown only in Jupyter notebooks, if you are writig a normal `.py` script and you want to see results you must in any case use `print`.

If we want to print more expressions in one row, we can pass them as different parameters to `print` by separating them with a comma:

[9]:
```python
coins = 4
print(2+5, coins)
```

```
7 4
```

To `print` we can pass as many expressions as we want:

[10]:
```python
coins = 4
print(2 + 5, coins, coins*3)
```

```
7 4 12
```

If we also want to show some text, we can write it by creating so-called *strings* between double quotes (we will see strings much more in detail in next chapters):

[11]:
```python
coins = 4
print("We have", coins, "golden coins, but we would like to have double:", coins * 2)
```

```
We have 4 golden coins, but we would like to have double: 8
```

QUESTION: Have a look at following expressions, and for each one of them try to guess the result it produces. Try cerifying your guesses both in Jupyter and another editor of files `.py` like Spyder:

1. ```python
   x = 1
   x
   x
   ```

2. ```python
   x = 1
   x = 2
   print(x)
   ```

3. ```python
   x = 1
   x = 2
   x
   ```

4. ```python
   x = 1
   print(x)
   x = 2
   print(x)
   ```

5. ```python
   print(zam)
   print(zam)
   zam = 1
   zam = 2
   ```

6. ```python
   x = 5
   print(x,x)
   ```

7. ```python
   x = 5
   print(x)
   print(x)
   ```

8. ```python
   carpets = 8
   length = 5
   print("If I have", carpets, "carpets in sequence I walk for", carpets * length,
   ↪"meters.")
   ```

9. ```python
   carpets = 8
   length = 5
   print("If", "I","have", carpets, "carpets","in", "sequence","I", "walk", "for",␣
   ↪carpets * length, "meters.")
   ```
   ```

Exercise - Castles in the air

Given two variables

```
castles = 7
dirigibles = 4
```

write some code to print:

```
I've built 7 castles in the air
I have 4 steam dirigibles
I want a dirigible parked at each castle
So I will buy other 3 at the Steam Market
```

- **DO NOT** put numerical constants in your code like `7`, `4` or `3`! Write generic code which only uses the provided variables.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```
castles = 7
dirigibles = 4
# write here
print("I've built",castles, "castles in the air")
print("I have", dirigibles, "steam dirigibles")
print("I want a dirigible parked at each castle")
print("So I will buy other", castles - dirigibles, "at the Steam Market")
```

```
I've built 7 castles in the air
I have 4 steam dirigibles
I want a dirigible parked at each castle
So I will buy other 3 at the Steam Market
```

</div>

[12]:
```
castles = 7
dirigibles = 4
# write here
```

```
I've built 7 castles in the air
I have 4 steam dirigibles
I want a dirigible parked at each castle
So I will buy other 3 at the Steam Market
```

5.1.5 Visualizing the execution with Python Tutor

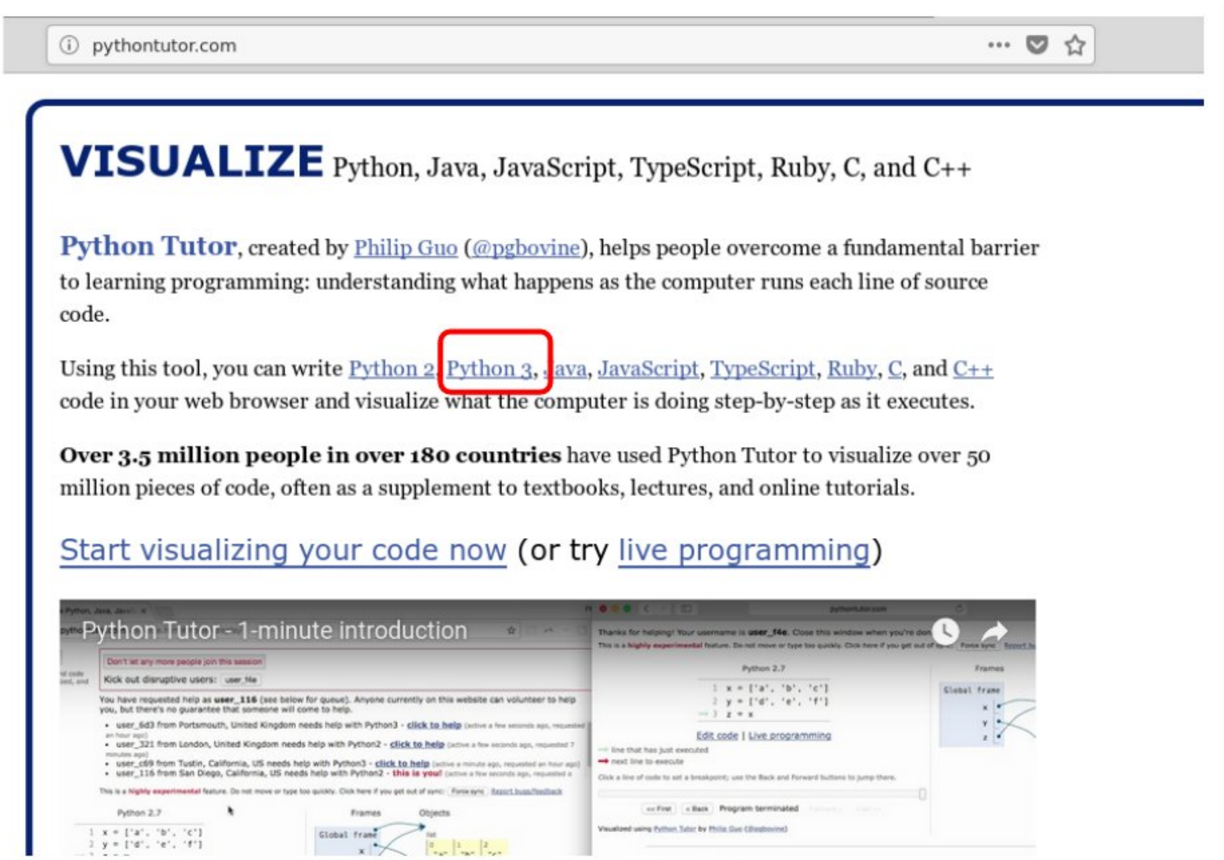
We have seen some of the main data types. Before going further, it's good to see the right tools to understand at best what happens when we execute the code.

Python tutor[158] is a very good website to visualize online Python code execution, allowing to step forth and *back* in code flow. Exploit it as much as you can, it should work with many of the examples we shall see in the book. Let's try an example

Python tutor 1/4

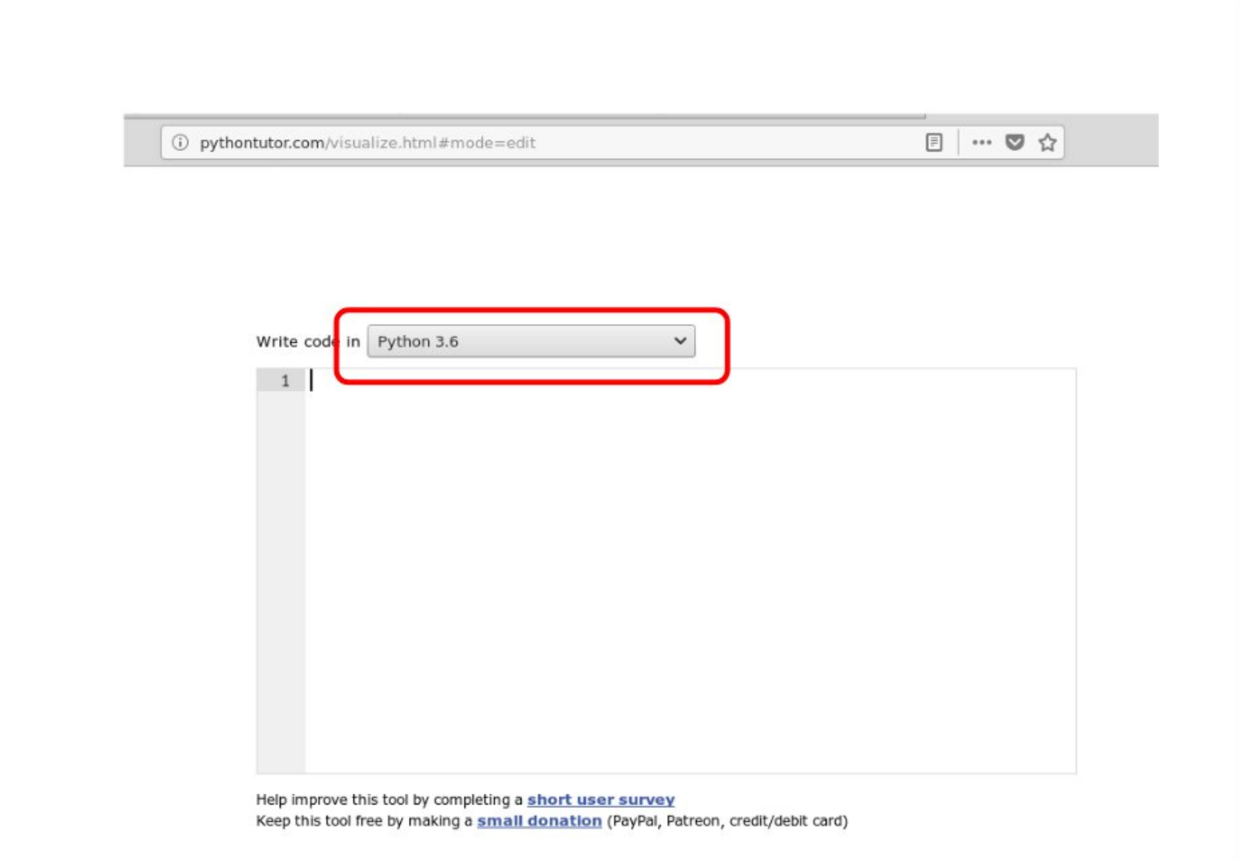
[158] http://pythontutor.com/

Go to pythontutor.com[159] and select *Python 3*

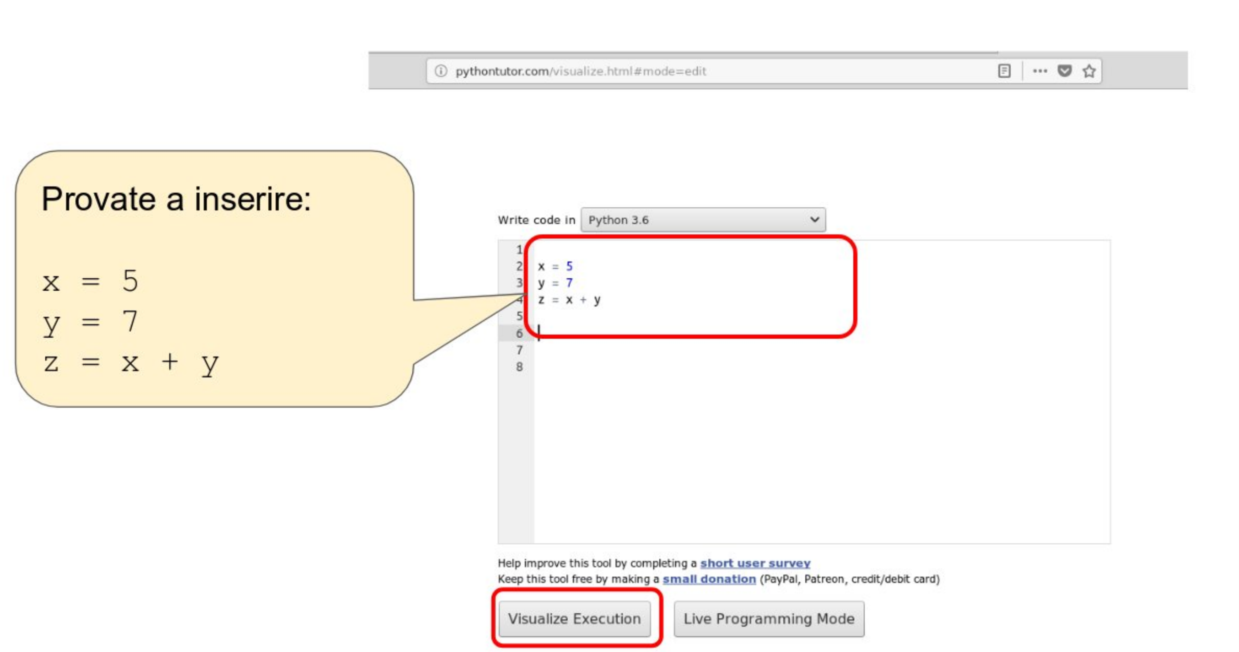


Python tutor 2/4

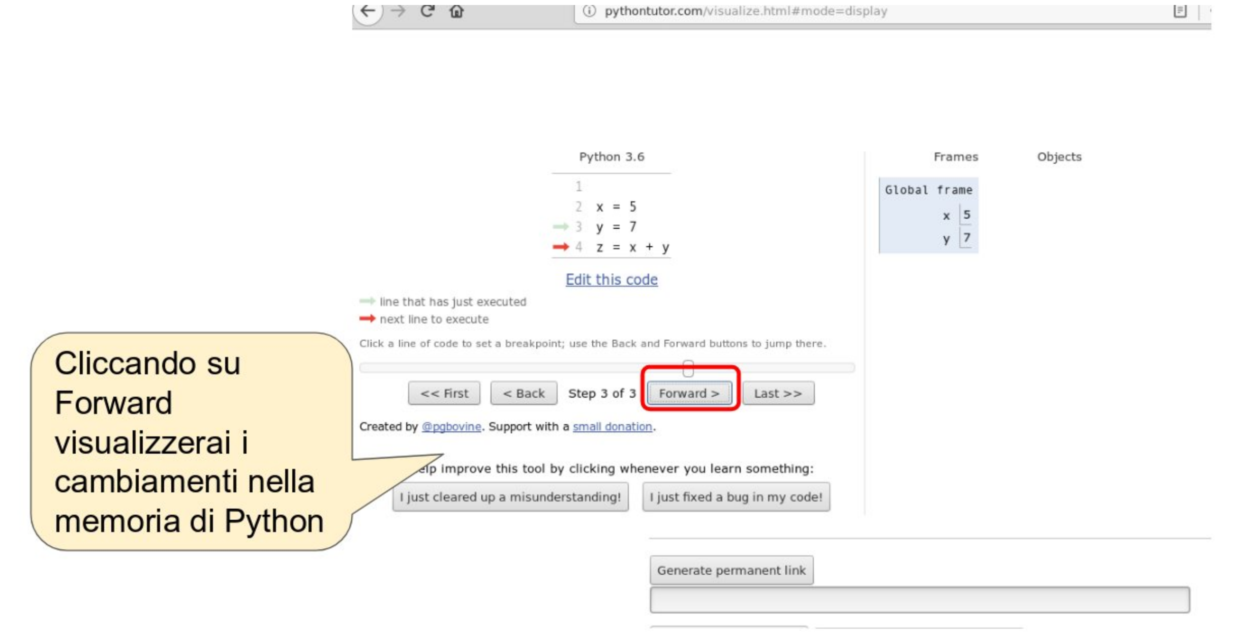
[159] http://pythontutor.com/



Python tutor 3/4



Python tutor 4/4



Debugging code in Jupyter

Python Tutor is fantastic, but when you execute code in Jupyter and it doesn't work, what can you do? To inspect the execution, the editor usually make available a tool called *debugger,* which allows to execute instructions one by one. At present (August 2018), the Jupyter debugger is called pdb[160] and it is extremely limited. To overcome limitations, in this book we invented a custom solution which exploits Python Tutor.

If you insert Python code in a cell, and then **at the cell end** you write the instruction `jupman.pytut()`, the preceding code will be visualized inside Jupyter notebook with Python Tutor, as if by magic.

> **WARNING**: `jupman` is a collection of support functions we invented just for this book.
>
> Whenever you see commands which start with `jupman`, to make them work you need first to execute the cell at the beginning of the document. For convenience we report here that cell. If you already didn't, execute it now.

[13]:
```python
# Remember to execute this cell with Control+Enter
# These commands tell Python where to find the file jupman.py
import jupman;
```

Now we are ready yo try Python Tutor with the magic function `jupman.pytut()`:

[14]:
```python
x = 5
y = 7
z = x + y

jupman.pytut()
```

[14]:
```
<IPython.core.display.HTML object>
```

[160] https://davidhamann.de/2017/04/22/debugging-jupyter-notebooks/

Python Tutor : Limitation 1

Python Tutor is handy, but there are important limitations:

> **ATTENTION**: Python Tutor only looks inside one cell!
>
> Whenever you use Python Tutor inside Jupyter, the only code Python tutors considers is the one inside the cell where the command `jupman.pytut()` is.

So for example in the two following cells, only `print(w)` will appear inside Python tutor without the `w = 3`. If you try clicking *Forward* in Python tutor, you will we warned that `w` was not defined.

[15]:
```python
w = 3
```

[16]:
```python
print(w)

jupman.pytut()
```

```
3
```

```
Traceback (most recent call last):
  File "../jupman.py", line 2305, in _runscript
    self.run(script_str, user_globals, user_globals)
  File "/usr/lib/python3.5/bdb.py", line 431, in run
    exec(cmd, globals, locals)
  File "<string>", line 2, in <module>
NameError: name 'w' is not defined
```

[16]:
```
<IPython.core.display.HTML object>
```

To have it work in Python Tutor you must put ALL the code in the SAME cell:

[17]:
```python
w = 3
print(w)

jupman.pytut()
```

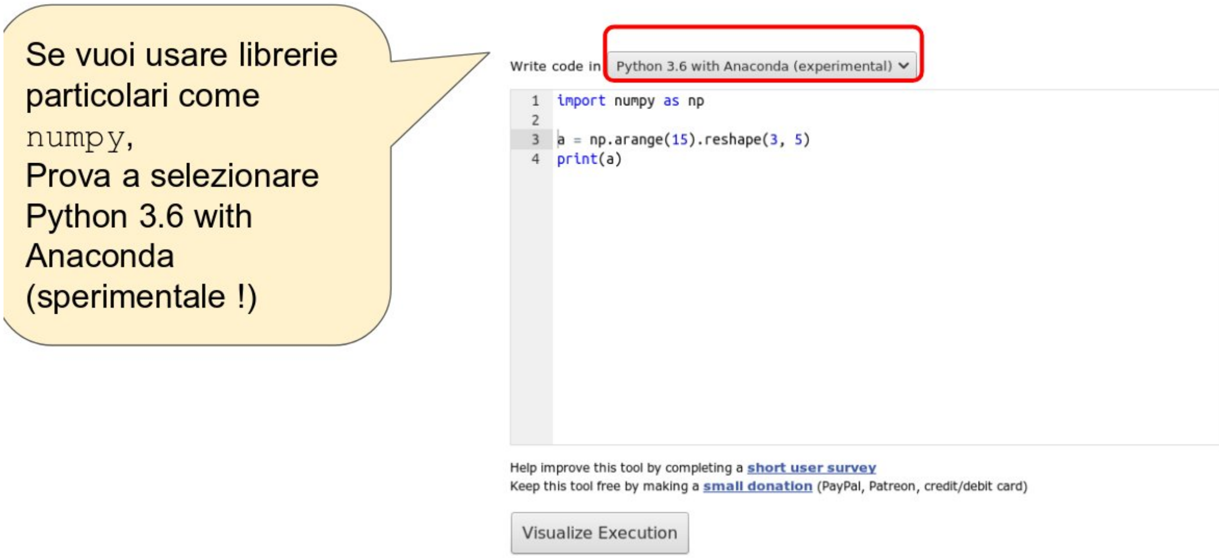
```
3
```

[17]:
```
<IPython.core.display.HTML object>
```

Python Tutor : Limitation 2

> **WARNING: Python Tutor only uses functions from standrd PYthon distribution**
>
> PYthon Tutor is good to inspect simple algorithms with basic Python functions, if you use libraries from third parties it will not work.

If you use some library like `numpy`, you can try **only online** to select `Python 3.6 with anaconda`



Exercise - tavern

Given the variables

```
pirates = 10
each_wants = 5      # mugs of grog
kegs = 4
keg_capacity = 20   # mugs of grog
```

Try writing some code which prints:

```
In the tavern there are 10 pirates, each wants 5 mugs of grog
We have 4 kegs full of grog
From each keg we can take 20 mugs
Tonight the pirates will drink 50 mugs, and 30 will remain for tomorrow
```

- **DO NOT** use numerical constant in your code, instead try using proposed variables
- To keep track of remaining kegs, make a variable `remaining_mugs`
- if you are using Jupyter, try using `jupman.pytut()` at the cell end to visualize the execution

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:

```
pirates = 10
each_wants = 5       # mugs of grog
kegs = 4
keg_capacity = 20   # mugs of grog


# write here
print("In the tavern there are", pirates, "pirates, each wants", each_wants, "mugs of grog")
print("We have", kegs, "kegs full of grog")
print("From each keg we can take", keg_capacity,"mugs")
```

(continues on next page)

(continued from previous page)

```
remaining_mugs = kegs*keg_capacity - pirates*each_wants
print("Tonight the pirates will drink", pirates * each_wants, "mugs, and", remaining_
↪mugs, "will remain for tomorrow")

#jupman.pytut()
```

```
In the tavern there are 10 pirates, each wants 5 mugs of grog
We have 4 kegs full of grog
From each keg we can take 20 mugs
Tonight the pirates will drink 50 mugs, and 30 will remain for tomorrow
```

</div>

[18]:

```
pirates = 10
each_wants = 5      # mugs of grog
kegs = 4
keg_capacity = 20  # mugs of grog


# write here


```

```
In the tavern there are 10 pirates, each wants 5 mugs of grog
We have 4 kegs full of grog
From each keg we can take 20 mugs
Tonight the pirates will drink 50 mugs, and 30 will remain for tomorrow
```

5.1.6 Python Architecture

The following part is not strictly fundamental to understand the book, it's useful to understand what happens under the hood when you execute commands.

Let's go back to Jupyter: the notebook editor Jupyter is a very powerful tool and flexible, allows to execute Python code, not only that, also code written in other programming languages (R, Bash, etc) and formatting languages (HTML, Markdown, Latex, etc).

Se must keep in mind that the Python code we insert in cells of Jupyter notebooks (the files with extension `.ipynb`) is not certainly magically understood by your computer. Under the hood, a lot of transformations are performed so to allow you computer processor to understaned the instructions to be executed. We report here the main transformations which happen, from Jupyter to the processor (CPU):

Python is a high level language

Let's try to understand well what happens when you execute a cell:

1. **source code**: First Jupyter checks if you wrote some Python *source code* in the cell (it could also be other programming languages like R, Bash, or formatting like Markdown …). By default Jupyter assumes your code is Python. Let's suppose there is the following code:

```
x = 3
y = 5
print(x + y)
```

EXERCISE: Without going into code details, try copy/pasting it into the cell below. Making sure to have the cursor in the cell, execute it with `Control + Enter`. When you execute it an `8` should appear as calculation result. The `# write down here` as all rows beginning with a sharp `#` is only a comment which will be ignored by Python

[19]:
```
# write down here
```

If you managed to execute the code, you can congratulate Python! It allowed you to execute a program written in a quite comprehensible language *independently* from your operating system (Windows, Mac Os X, Linux …) and from the processor of your computer (x86, ARM, …)! Not only that, the notebook editor Jupyter also placed the result in your browser.

In detail, what happened? Let's see:

2. **bytecode**: When requesting the execution, Jupyter took the text written in the cell, and sent it to the so-called *Python compiler* which transformed it into *bytecode*. The *bytecode* is a longer sequence of instructions which is less intelligeble for us humans (**this is only an example, there is no need to understand it !!**):

```
2           0 LOAD_CONST               1 (3)
            3 STORE_FAST               0 (x)

3           6 LOAD_CONST               2 (5)
            9 STORE_FAST               1 (y)

4          12 LOAD_GLOBAL              0 (print)
           15 LOAD_FAST                0 (x)
           18 LOAD_FAST                1 (y)
           21 BINARY_ADD
           22 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
           25 POP_TOP
           26 LOAD_CONST               0 (None)
           29 RETURN_VALUE
```

3. **machine code**: The *Python interpreter* took the *bytecode* above one instruction per time, and converted it into *machine code* which can actually be understood by the processor (CPU) of your computer. To us the *machine code* may look even longer and uglier of *bytecode* but the processor is happy and by reading it produces the program results. Example of *machine code* (**it is just an example, you do not need to understand it !!**):

```
mult:
    push rbp
    mov rbp, rsp
    mov eax, 0
mult_loop:
    cmp edi, 0
    je mult_end
    add eax, esi
    sub edi, 1
    jmp mult_loop
mult_end:
    pop rbp
    ret
```

We report in a table what we said above. In the table we explicitly write the file extension ni which we can write the various code formats

- The ones interesting for us are Jupyter notebooks `.ipynb` and Python source code files `.py`
- `.pyc` file smay be generated by the compiler when reading `.py` files, but they are not interesting to us, we will

never need to edit the,

- `.asm` machine code also doesn't matter for us

Tool	Language	File	Example
Jupyter Notebook	Python	.ipynb	
Python Compiler	Python source code	.py	`x = 3 y = 5print(x + y )`
Python Interpreter	Python bytecode	.pyc	`0 LOAD_CONST 1 (3)3 STORE_FAST 0 (x)`
Processor (CPU)	Machine code	.asm	`cmp edi, 0je mult _end`

No that we now have an idea of what happens, we can maybe understand better the statement *Python is a high level language,* that is, it's positioned high in the above table: when we write Python code, we are not interested in the generated *bytecode* or *machine code,* we can **just focus on the program logic**. Besides, the Python code we write is **independent from the pc architecture**: if we have a Python interpreter installed on a computer, it will take care of converting the high-level code into the machine code of that particular architecture, which includes the operating system (Windows / Mac Os X / Linux) and processor (x86, ARM, PowerPC, etc).

Performance

Everything has a price. If we want to write programs focusing only on the *high level logic* without entering into the details of how it gets interpreted by the processor, we typically need to give up on *performance.* Since Python is an *interpreted* language has the downside of being slow. What if we really need efficiency? Luckily, Python can be extended with code written in *C language* which typically is much more performant. Actually, even if you won't notice it, many functions of Python under the hood are directly written in the fast C language. If you really need performance (not in this book!) it might be worth writing first a prototype in Python and, once established it works, compile it into *C language* by using Cython compiler[161] and manually optimize the generated code.

```
[ ]:
```

5.2 Python basics

5.2.1 Download exercises zip

Browse online files[162]

PREREQUISITES:

- **Having installed Python 3 and Jupyter:** if you haven't already, look Installation[163]
- **Having read** Tools and scripts[164]

161 http://cython.org/
162 https://github.com/DavidLeoni/softpython-en/tree/master/basics
163 https://en.softpython.org/installation.html
164 https://en.softpython.org/tools/tools-sol.html

5.2.2 Jupyter

Jupyter is an editor taht allows to work on so called *notebooks,* which are files ending with the extension `.ipynb`. They are documents divided in cells where for each cell you can insert commands and immediately see the respective output. Let's try to open this.

1. Unzip exercises zip in a folder, you should obtain something like this:

```
basics
    basics-sol.ipynb
    basics.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook. Two things should appear, first a console and then a browser. In the browser navigate the files to reach the unzipped folder, and open the notebook `basics.ipynb`

WARNING: DO NOT click Upload button in Jupyer

Just navigate until you reach the file.

WARNING: open the notebook WITHOUT the `-sol` at the end!

Seeing now the solutions is too easy ;-)

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells.Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

WARNING: In this book we use ONLY PYTHON 3

If you obtain weird behaviours, check you are using Python 3 and not 2. If by typing `python` your operating system runs python 2, try executing `python3`

If you don't find Jupyter / something doesn't work: have a look at installation[165]

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[165] https://en.softpython.org/installation.html#Jupyter-Notebook

5.2.3 Objects

In Python everything is an object. Objects have **properties** (fields where to save values) and **methods** (things they can do). For example, an object **car** has the *properties* model, brand, color, numer of doors, etc … and the *methods* turn right, turn left, accelerate, brake, shift gear …

According to Python official documentation:

```
"Objects are Python's abstraction for data. All data in a Python program is
↪represented by objects or by relations between objects."
```

For now it's enough to know that Python objects have an **identifier** (like, their name), a **type** (numbers, text, collections, …) and a **value** (the actual value represented by objects). Once the object has been created the *identifier* and the *type* never change, while the *value* may change (**mutable objects**) or remain constant (**immutable objects**).

Python provides these predefined types (*built-in*):

Type	Meaning	Domain	Mutable?
`bool`	Condition	`True`, `False`	no
`int`	Integer	$\mathbb{Z}$	no
`long`	Integer	$\mathbb{Z}$	no
`float`	Rational	$\mathbb{Q}$ (more or less)	no
`str`	Text	Text	no
`list`	Sequence	Collezione di oggetti	yes
`tuple`	Sequence	Collezione di oggetti	no
`set`	Set	Collezione di oggetti	yes
`dict`	Mapping	Mapping between objects	yes

For now we will consider only the simplest ones, later in the book we will deep dive in each of them.

5.2.4 Variables

Variables are associations among names and objects (we can call them values).

Variables can be associated, or in a more technical term, *assigned* to objects by using the assignment operator =.

The instruction

[2]:
```
diamonds = 4
```

may represent how many precious stones we keed in the safe. What happens when we execute it in Python?

- an object is created
- its type is set to `int` (an integer number)
- its value is set to 4
- a name `diamonds` is create in the environment and assigned to that object

Detect the type of a variable

When you see a variable or costant and you wonder what type it could have, you can use the predefined function `type`:

[3]:
```
type(diamonds)
```

[3]:
```
int
```

[4]:
```
type(4)
```

[4]:
```
int
```

[5]:
```
type(4.0)
```

[5]:
```
float
```

[6]:
```
type("Hello")
```

[6]:
```
str
```

Reassign a variable

Consider now the following code:

[7]:
```
diamonds = 4
print(diamonds)
```

```
4
```

[8]:
```
diamonds = 5
print(diamonds)
```

```
5
```

The value of `diamonds` variable has been changed from 4 to 5, but as reported in the previous table, the `int` type is **immutable**. Luckily, this didn't prevent us from changing the value `diamonds` from 4 to 5. What happend behind the scenes? When we executed the instructions `diamonds = 5`, a new object of type `int` was created (the integer `5`) and made available with the same name `diamonds`

Reusing a variable

When you reassign a variable to another value, to calculate the new value you can freely reuse the old value of the variable you want to change. For example, suppose to have the variable

[9]:
```
flowers = 4
```

and you want to augment the number of `flowers` by one. You can write like this:

[10]:
```
flowers = flowers + 1
```

What happened? When Python encounters a command with =, FIRST it calculates the value of the expression it finds to the right of the =, and THEN assigns that value to the variable to the left of the =.

Given this order, FIRST in the expression on the right the old value is used (in this case `4`) and `1` is summed so to obtain `5` which is THEN assigned to `flowers`.

[11]:
```
flowers
```

[11]:
```
5
```

In a completely equivalent manner, we could rewrite the code like this, using a helper variable `x`. Let's try it in Python Tutor:

[12]:
```
# WARNING: to use the following jupman.pytut() function,
# it is necessary first execute  this cell with Shift+Enter

# it's enough to execute once, you can also find in all notebooks in the first cell.

import jupman
```

[13]:
```

flowers = 4

x = flowers + 1

flowers = x

jupman.pytut()
```

[13]:
```
<IPython.core.display.HTML object>
```

You can execute a sum and do an assignment at the same time with the += notation

[14]:
```
flowers = 4
flowers += 1
print(flowers)
```

```
5
```

This notation is also valid for other arithetic operators:

[15]:
```
flowers = 5
flowers -= 1      # subtraction
print(flowers)
```

```
4
```

[16]:
```
flowers *= 3      # multiplication
print(flowers)
```

```
12
```

[17]:
```
flowers /= 2      # division
print(flowers)
```

```
6.0
```

Assignments - questions

QUESTION: Look at the following questions, and for each try to guess the result it produces (or if it gives an error). Try to verify your guess both in Jupyter and in another editor of `.py` files like Spyder:

1. ```
   x = 1
   x
   x
   ```

2. ```
   x = 1
   x = 2
   print(x)
   ```

3. ```
   x = 1
   x = 2
   x
   ```

4. ```
   x = 1
   print(x)
   x = 2
   print(x)
   ```

5. ```
   print(zam)
   print(zam)
   zam = 1
   zam = 2
   ```

6. ```
   x = 5
   print(x,x)
   ```

7. ```
   x = 5
   print(x)
   print(x)
   ```

8. ```
   x = 3
   print(x,x*x,x**x)
   ```

9. ```
   3 + 5 = x
   print(x)
   ```

10. ```
    3 + x = 1
    print(x)
    ```

11. ```
    x + 3 = 2
    print(x)
    ```

12. ```
    x = 2
    x =+ 1
    print(x)
    ```

13. ```
    x = 2
    x = +1
    print(x)
    ```
    ```

14.
```
x = 2
x += 1
print(x)
```

15.
```
x = 3
x *= 2
print(x)
```

Exercise - exchange

⊛ Given two variables `a` and `b`:

```
a = 5
b = 3
```

write some code that exchanges the two values, so that after your code it must result

```
>>> print(a)
3
>>> print(b)
5
```

- are two variables enough? If they aren't, try to introduce a third one.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```
a = 5
b = 3

# write here
temp = a    # associate 5 to temp variable, so we have a copy
a = b       # reassign a to the value of b, that is 3
b = temp    # reassign b to the value of temp, that is 5
#print(a)
#print(b)
```

</div>

[18]:
```
a = 5
b = 3

# write here
```

Exercise - cycling

⊕ Write a program that given three variables with numebers a,b,c, cycles the values, that is, puts the value of a in b, the value of b in c, and the value of c in a .

So if you begin like this:

```
a = 4
b = 7
c = 9
```

After the code that you will write, by running this:

```
print(a)
print(b)
print(c)
```

You should see:

```
9
4
7
```

There are various ways to do it, try to use **only one** temporary variable and be careful not to lose values !

HINT: to help yourself, try to write down in comments the state of the memory, and think which command to do

`python # a b c t which command do I need? # 4 7 9 # 4 7 9 7 t = b # # #`

[19]:
```
a = 4
b = 7
c = 9

# write code here

print(a)
print(b)
print(c)
```

```
4
7
9
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[20]:
```
# SOLUTION

a = 4
b = 7
c = 9


# a b c t  which command do I need?
# 4 7 9
# 4 7 9 7  t = b
# 4 4 9 7  b = a
# 9 4 9 7  a = c
# 9 4 7 7  c = t
```

(continues on next page)

(continued from previous page)

```
t = b
b = a
a = c
c = t

print(a)
print(b)
print(c)
```

```
9
4
7
```

</div>

[20]:

```
9
4
7
```

Changing type during execution

You can also change the type of a variable duting the program execution but normally it is a **bad habit** because it makes harder to understand the code, and increases the probability to commit errors. Let's make an example:

[21]:
```
diamonds = 4          # integer
```

[22]:
```
diamonds + 2
```

[22]:
```
6
```

[23]:
```
diamonds = "four"   # text
```

Now that `diamonds` became text, if by mistake we try to treat it as if it were a number we will get an error !!

```
diamonds + 2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-9-6124a47997d7> in <module>
----> 1 diamonds + 2

TypeError: can only concatenate str (not "int") to str
```

Multiple commands on the same line

It is possible to put many commands on the same line (non only assignments) by separating them with a semi-colon `;`

```
[24]: a = 10; print('So many!'); b = a + 1;
```

```
So many!
```

```
[25]: print(a,b)
```

```
10 11
```

NOTE: multiple commands on the same line are 'not much pythonic'

Even if sometimes they may be useful and less verbose of explicit definitions, they are a style frowned upon by true Python ninjas.

Multiple initializations

Another thing are multiple initializations, separated by a comma `,` like:

```
[26]: x,y = 5,7
```

```
[27]: print(x)
```

```
5
```

```
[28]: print(y)
```

```
7
```

Differently from multiple commands, multiple assignments are a more acceptable style.

Exercise - exchange like a ninja

⊛ Try now to exchange the value of the two variables `a` and `b` in one row with multiple initialization

```
a,b = 5,3
```

After your code, it must result

```
>>> print(a)
3
>>> print(b)
5
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[29]: a,b = 5,3

# write here
a,b = b,a
#print(a)
#print(b)
```

</div>

[29]:
```python
a,b = 5,3

# write here
```

Names of variables

IMPORTANT NOTE:

You can chose the name that you like for your variables (we advise to pick something reminding their meaning), but you need to adhere to some simple rules:

1. Names can only contain upper/lower case digits (`A-Z`, `a-z`), numbers (`0-9`) or underscores _;
2. Names cannot start with a number;
3. Variable names should start with a lowercase letter
4. Names cannot be equal to reserved keywords:

Reserved words:

`and`	`as`	`assert`	`break`	`class`	`continue`
`def`	`del`	`elif`	`else`	`except`	`exec`
`finally`	`for`	`from`	`global`	`if`	`import`
`in`	`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`	`with`
`yield`	`True`	`False`	`None`		

system functions: beyond reserved words (which are impossible to redefine), Python also offers several predefined system function:

- `bool`, `int`,`float`,`tuple`,`str`,`list`,`set`,`dict`
- `max`, `min`, `sum`
- `next`, `iter`
- `id`, `dir`, `vars`,`help`

Sadly, Python allows careless people to redefine them, but we **do not**:

V COMMANDMENT[166]: **You shall never ever redefine system functions**

Never declare variables with such names !

[166] https://en.softpython.org/commandments.html#V-COMMANDMENT

Names of variables - questions

For each of the following names, try to guess if it is a valid *variable name* or not, then try to assign it in following cell

1. `my-variable`
2. `my_variable`
3. `theCount`
4. `the count`
5. `some@var`
6. `MacDonald`
7. `7channel`
8. `channel7`
9. `stand.by`
10. `channel45`
11. `maybe3maybe`
12. `"ciao"`
13. `'hello'`
14. `as` PLEASE: DO UNDERSTAND THE *VERY IMPORTANT DIFFERENCE* BETWEEN THIS AND FOLLOWING TWOs !!!
15. `asino`
16. `As`
17. `lista` PLEASE: DO UNDERSTAND THE *VERY IMPORTANT DIFFERENCE* BETWEEN THIS AND FOLLOWING TWOs !!!
18. `list` DO NOT EVEN TRY TO ASSIGN THIS ONE IN THE INTERPRETER (like `list = 5`), IF YOU DO YOU WILL BASICALLY BREAK PYTHON
19. `List`
20. `black&decker`
21. `black & decker`
22. `glab()`
23. `caffè` (notice the accented `è` !)
24. `):-]`
25. `€zone` (notice the euro sign)
26. `some:pasta`
27. `aren'tyouboredyet`
28. `<angular>`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[30]: # write here
```

</div>

```
[30]: # write here
```

5.2.5 Numerical types

We already mentioned that numbers are **immutable objects**. Python provides different numerical types:

integers (`int`), reals (`float`), booleans, fractions and complex numbers.

It is possible to make arithmetic operations with the following operators, in precedence order:

Operator	Description
`**`	power
`+ -`	Unary plus and minus
`* / // %`	Multiplication, division, integer division, module
`+ -`	Addition and subtraction

There are also several predefined functions:

Function	Description
`min(x,y, ...)`	the minimum among given numbers
`max(x,y, ...)`	the maximum among given numbers
`abs(x)`	the absolute value

Others are available in the math[167] module (remember that in order to use them you must first import the module `math` by typing `import math`):

Function	Description
`math.floor(x)`	round `x` to inferior integer
`math.ceil(x)`	round `x` to superior integer
`math.sqrt(x)`	the square root
`math.log(x)`	the natural logarithm of `n`
`math.log(x,b)`	the logarithm of `n` in base `b`

… plus many others we don't report here.

5.2.6 Integer numbers

The range of values that integer can have is only limited by available memory. To work with numbers, Python also provides these operators:

```
[31]: 7 + 4
```

```
[31]: 11
```

```
[32]: 7 - 4
```

[167] https://docs.python.org/3/library/math.html

[32]:
```
3
```

[33]:
```
7 // 4
```

[33]:
```
1
```

NOTE: the following division among integers produces a **float** result, which uses a **dot** as separator for the decimals (we will see more details later):

[34]:
```
7 / 4
```

[34]:
```
1.75
```

[35]:
```
type(7 / 4)
```

[35]:
```
float
```

[36]:
```
7 * 4
```

[36]:
```
28
```

NOTE: in many programming languages the power operation is denoted with the cap `^`, but in Python it is denoted with double asterisk `**`:

[37]:
```
7 ** 4    # power
```

[37]:
```
2401
```

Exercise - deadline 1

⊛ You are given a very important deadline in:

[38]:
```
days = 4
hours = 13
minutes = 52
```

Write some code that prints the total minutes. By executing it, it should result:

```
In total there are 6592 minutes left.
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[39]:
```
days = 4
hours = 13
minutes = 52

# write here
print("In total there are", days*24*60 + hours*60 + minutes, "minutes left")
```

```
In total there are 6592 minutes left
```

</div>

[39]:
```
days = 4
hours = 13
```

(continues on next page)

(continued from previous page)

```
minutes = 52

# write here
```

```
In total there are 6592 minutes left
```

Modulo operator

To find the reminder of a division among integers, we can use the modulo operator which is denoted with `%`:

```
[40]: 5 % 3   # 5 divided by 3 gives 2 as reminder
```

```
[40]: 2
```

```
[41]: 5 % 4
```

```
[41]: 1
```

```
[42]: 5 % 5
```

```
[42]: 0
```

```
[43]: 5 % 6
```

```
[43]: 5
```

```
[44]: 5 % 7
```

```
[44]: 5
```

Exercise - deadline 2

⊛ For another super important deadline there are left:

```
tot_minutes = 5000
```

Write some code that prints:

```
There are left:
   3 days
   11 hours
   20 minutes
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[45]: tot_minutes = 5000

# write here
print('There are left:')
print('  ', tot_minutes // (60*24), 'days')
print('  ', (tot_minutes % (60*24)) // 60, 'hours')
print('  ', (tot_minutes % (60*24)) % 60, 'minutes')
```

```
There are left:
   3 days
   11 hours
   20 minutes
```

</div>

[45]:
```python
tot_minutes = 5000

# write here
```

```
There are left:
   3 days
   11 hours
   20 minutes
```

min and max

The minimum among two numbers can be calculated with the function `min`:

[46]:
```python
min(7,3)
```

[46]:
```
3
```

and the maximum with the function `max`:

[47]:
```python
max(2,6)
```

[47]:
```
6
```

To `min` and `max` we can pass an arbitrary number of parameters, even negatives:

[48]:
```python
min(2,9,-3,5)
```

[48]:
```
-3
```

[49]:
```python
max(2,9,-3,5)
```

[49]:
```
9
```

V COMMANDMENT[168]: **You shall never ever redefine system functions like `min` and `max`**

If you use `min` and `max` like they were variables, the corresponding functions will *literally* stop to work!

```python
min = 4    # NOOOO !
max = 7    # DON'T DO IT !
```

QUESTION: given two numbers `a` and `b`, which of the following expressions are equivalent?

```
1. max(a,b)
2. max(min(a,b),b)
3. -min(-a,-b)
4. -max(-a,-b)
```

[168] https://en.softpython.org/commandments.html#V-COMMANDMENT

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1. and 3. are equivalent

</div>

Exercise - transportation

⊛⊛ A company has a truck that every day delivers products to its best client. The truck can at most transport 10 tons of material. Unfortunately, the roads it can drive through have bridges that limit the maximum weight a vehicle can have to pass. These limits are provided in 5 variables:

```
b1,b2,b3,b4,b5 = 7,2,4,3,6
```

The truck must always go through the bridge `b1`, then along the journey there are three possible itineraries available:

- In the first itinerary, the truck also drives through bridge `b2`
- In the second itinerary, the truck also drives through bridges `b3` and `b4`
- In the third itinerary, the truck also drives though bridge `b5

The company wants to know which are the maximum tons it can drive to destination in a sngle journey. Write some code to print this number.

NOTE: we do not want to know which is the best itinerary, we only need to find the greatest number of tons to ship.

Example - given:

```
b1,b2,b3,b4,b5 = 7,2,4,6,3
```

your code must print:

```
In a single journey we can transport at most 4 tons.
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```
b1,b2,b3,b4,b5 = 7,2,4,6,3   # 4
#b1,b2,b3,b4,b5 = 2,6,2,4,5  # 2
#b1,b2,b3,b4,b5 = 8,6,2,9,5  # 6
#b1,b2,b3,b4,b5 = 8,9,9,4,7  # 8


# write here

print('In a single journey we can transport at most',
       max(min(b1,b2), min(b1,b3,b4),min(b1,b5)),
      'tons')
```

```
In a single journey we can transport at most 4 tons
```

</div>

[50]:
```
b1,b2,b3,b4,b5 = 7,2,4,6,3   # 4
#b1,b2,b3,b4,b5 = 2,6,2,4,5  # 2
#b1,b2,b3,b4,b5 = 8,6,2,9,5  # 6
#b1,b2,b3,b4,b5 = 8,9,9,4,7  # 8
```

(continues on next page)

(continued from previous page)

```
# write here
```

```
In a single journey we can transport at most 4 tons
```

Exercise - armchairs

⚝⚝ The tycoon De Industrionis owns to factories of armchairs, one in Belluno city and one in Rovigo. To make an armchair three main components are needed: a mattress, a seatback and a cover. Each factory produces all required components, taking a certain time to produce each component:

[51]:
```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24
```

Belluno takes 23h to produce a mattress, 54h the seatcback and 12h the cover. Rovigo, respectively, takes 13, 37 and 24 hours. When the 3 components are ready, assembling them in the finished armchair requires one hour.

Sometimes peculiar requests are made by filthy rich nobles, that pretends to be shipped in a few hours armchairs with extravagant like seatback in solid platinum and other nonsense.

If the two factories start producting the components at the same time, De Industrionis wants to know in how much time the first armchair will be produced. Write some code to calculate that number.

- **NOTE 1**: we are not interested in which factory will produce the armchair, we just want to know the shortest time in which we will get an armchair
- **NOTE 2**: suppose both factories **don't** have components in store
- **NOTE 3**: the two factories **do not** exchange components

Example 1 - given:

```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24
```

your code must print:

```
The first armchair will be produced in 38 hours.
```

Example 2 - given:

```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 81,37,32,54,36,91
```

your code must print:

```
The first armchair will be produced in 82 hours.
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[52]:
```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24   # 38
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 81,37,32,54,36,91  # 82
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 21,39,47,54,36,91  # 48

# write here
```

(continues on next page)

(continued from previous page)

```
t = min(max(b_mat, b_bac, b_cov) + 1, max(r_mat, r_bac, r_cov) + 1)

print('The first armchair will be produced in', t,'hours.')
```

```
The first armchair will be produced in 38 hours.
```

</div>

[52]:

```
b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 23,54,12,13,37,24    # 38
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 81,37,32,54,36,91   # 82
#b_mat, b_bac, b_cov, r_mat, r_bac, r_cov = 21,39,47,54,36,91   # 48

# write here
```

```
The first armchair will be produced in 38 hours.
```

5.2.7 Booleans

Values of truth in Python are represented with the keywords `True` and `False`. A boolean object can only have the values `True` or `False`. These objects are used in boolean algebra and have the type `bool`.

[53]:

```
x = True
```

[54]:

```
x
```

[54]:

```
True
```

[55]:

```
type(x)
```

[55]:

```
bool
```

[56]:

```
y = False
```

[57]:

```
type(y)
```

[57]:

```
bool
```

Boolean operators

We can operate on boolean values with the operators `not`, `and`, `or`:

[58]:

```
# Expression      Result
not True          # False
not False         # True

False and False   # False
False and True    # False
True  and False   # False
True  and True    # True
```

(continues on next page)

(continued from previous page)

```
False or False    # False
False or True     # True
True  or False    # True
True  or True     # True
```

[58]:
```
True
```

Booleans - Questions with costants

QUESTION: For each of the following boolean expressions, try to guess the result (*before* guess, and *then* try them !):

1. `not (True and False)`
2. `(not True) or (not (True or False))`
3. `not (not True)`
4. `not (True and (False or True))`
5. `not (not (not False))`
6. `True and (not (not((not False) and True)))`
7. `False or (False or ((True and True) and (True and False)))`

Booleans - Questions with variables

QUESTION: For each of these expressions, for which values of `x` and `y` they give `True`? Try to think an answer before trying!

NOTE: there can be many combinations that produce `True`, find them all

1. `x or (not x)`
2. `(not x) and (not y)`
3. `x and (y or y)`
4. `x and (not y)`
5. `(not x) or  y`
6. `y or not (y and x)`
7. `x and ((not x) or not(y))`
8. `(not (not x)) and not (x and y)`

9. `x and (x or (not(x) or not(not(x or not (x)))))`

QUESTION: For each of these expressions, for which values of `x` and `y` they give `False`?

NOTE: there can be many combinations that produce `False`, find them all

1. `x or ((not y) or z)`
2. `x or (not y) or (not z)`
3. `not (x and y and (not z))`
4. `not (x and (not y) and (x or z))`
5. `y or ((x or y) and (not z))`

Booleans - De Morgan

There are a couple of laws that sometimes are useful:

Formula	Equivalent to
`x or y`	`not(not x and not y)`
`x and y`	`not(not x or not y)`

QUESTION: Look at following expressions, and try to rewrite them in equivalent ones by using De Morgan laws, simplifying the result wherever possible. Then verify the translation produces the same result as the original for all possible values of `x` and `y`.

1. `(not x) or y`
2. `(not x) and (not y)`
3. `(not x) and (not (x or y))`

Example:

```
x,y = False, False
#x,y = False, True
#x,y = True, False
#x,y = True, True

orig = x or y
trans = not((not x) and (not y))
print('orig=',orig)
print('trans=',trans)
```

[59]:
```
# write here
```

Booleans - Conversion

We can convert booleans into intergers with the predefined function `int`. Each integer can be converted into a boolean (and vice versa) with `bool`:

[60]:
```python
bool(1)
```

[60]:
```
True
```

[61]:
```python
bool(0)
```

[61]:
```
False
```

[62]:
```python
bool(72)
```

[62]:
```
True
```

[63]:
```python
bool(-5)
```

[63]:
```
True
```

[64]:
```python
int(True)
```

[64]:
```
1
```

[65]:
```python
int(False)
```

[65]:
```
0
```

Each integer is valued to `True` except `0`. Note that truth values `True` and `False` behave respectively like integers `1` and `0`.

Booleans - Questions - what is a boolean?

QUESTION: For each of these expressions, which results it produces?

1. `bool(True)`
2. `bool(False)`
3. `bool(2 + 4)`
4. `bool(4-3-1)`
5. `int(4-3-1)`
6. `True + True`
7. `True + False`
8. `True - True`
9. `True * True`

Booleans - Evaluation order

For efficiency reasons, during the evaluation of a boolean expression if Python discovers the possible result can only be one, it then avoids to calculate further expressions. For example, in this expression:

```
False and x
```

by reading from left to right, in the moment we encounter `False` we already know that the result of `and` operation will always be `False` independetly from the value of `x` (convince yourself).

Instead, if while reading from left to right Python finds first `True`, it will continue the evaluation of following expressions and as result of the whole `and` will return the evaluation of the **last** expression. If we are using booleans, we will not notice the differences, but by exchanging types we might get surprises:

[66]:
```
True and 5
```

[66]:
```
5
```

[67]:
```
5 and True
```

[67]:
```
True
```

[68]:
```
False and 5
```

[68]:
```
False
```

[69]:
```
5 and False
```

[69]:
```
False
```

Let's think which order of evaluation Python might use for the `or` operator. Have a look at the expression:

```
True or x
```

By reading from left to right, as soon as we find the `True` we mich conclude that the result of the whole `or` must be `True` independently from the value of `x` (convince yourself).

Instead, if the first value is `False`, Python will continue in the evaluation until it finds a logical value `True`, when this happens that value will be the result of the whole expression. We can notice it if we use different costants from `True` and `False`:

[70]:
```
False or 5
```

[70]:
```
5
```

[71]:
```
7 or False
```

[71]:
```
7
```

[72]:
```
3 or True
```

[72]:
```
3
```

The numbers you see have always a logical result coherent with the operations we did, that is, if you see `0` the expression result is intended to have logical value `False` and if you see a number different from `0` the result is intended to be `True` (convince yourself).

QUESTION: Have a look at the following expressions, and for each of them try to guess which result it produces (or if it gives an error):

1. `0 and True`

2. `1 and 0`

3. `True and -1`

4. `0 and False`

5. `0 or False`

6. `0 or 1`

7. `False or -6`

8. `0 or True`

Booleans - evaluation errors

What happens if a boolean expression contains some code that would generate an error? According to intuition, the program should terminate, but it's not always like this.

Let's try to generate an error on purpose. During math lessons they surely told you many times that dividing a number by zero is an error because the result is not defined. So if we try to ask Python what the result of `1/0` is we will (predictably) get complaints:

```
print(1/0)
print('after')

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-51-9e1622b385b6> in <module>()
----> 1 1/0

ZeroDivisionError: division by zero
```

Notice that `'after'` *is not* printed because the progam gets first interrupted.

What if we try to write like this?

[73]:
```
False and 1/0
```

[73]:
```
False
```

Python produces a result without complaining ! Why? Evaluating form left to right it found a `False` and so it concluded before hand that the expression result must be `False`. Many times you will not be aware of these potential problems but it is good to understand them because there are indeed situations in which you can event exploit the execution order to prevent errors (for example in `if` and `while` instructions we will see later in the book).

QUESTION: Look at the following expression, and for each of them try to guess which result it produces (or if it gives on error):

1. `True and 1/0`

2. `1/0 and 1/0`
3. `False or 1/0`
4. `True or 1/0`
5. `1/0 or True`
6. `1/0 or 1/0`
7. `True or (1/0 and True)`
8. `(not False) or not 1/0`
9. `True and 1/0 and True`
10. `(not True) or 1/0 or True`
11. `True and (not True) and 1/0`

Comparison operators

Comparison operators allow to build *expressions* which return a boolean value:

Comparator	Description
`a == b`	`True` if and only if $a = b$
`a != b`	`True` if and only if $a \neq b$
`a < b`	`True` if and only if $a < b$
`a > b`	`True` if and only if $a > b$
`a <= b`	`True` if and only if $a \leq b$
`a >= b`	`True` if and only if $a \geq b$

[74]:
```python
3 == 3
```
[74]:
```
True
```

[75]:
```python
3 == 5
```
[75]:
```
False
```

[76]:
```python
a,b = 3,5
```

[77]:
```python
a == a
```
[77]:
```
True
```

[78]:
```python
a == b
```
[78]:
```
False
```

[79]:
```
a == b - 2
```

[79]:
```
True
```

[80]:
```
3 != 5  # 3 is different from 5 ?
```

[80]:
```
True
```

[81]:
```
3 != 3  # 3 is different from 3 ?
```

[81]:
```
False
```

[82]:
```
3 < 5
```

[82]:
```
True
```

[83]:
```
5 < 5
```

[83]:
```
False
```

[84]:
```
5 <= 5
```

[84]:
```
True
```

[85]:
```
8 > 5
```

[85]:
```
True
```

[86]:
```
8 > 8
```

[86]:
```
False
```

[87]:
```
8 >= 8
```

[87]:
```
True
```

Since the comparison are expressions which produce booleans, we can also assign the result to a variable:

[88]:
```
x = 5 > 3
```

[89]:
```
print(x)
```

```
True
```

QUESTION: Look at the following expression, and for each of them try to guess which result it produces (or if it gives on error):

1.
```
x = 3 == 4
print(x)
```

2.
```
x = False or True
print(x)
```

3.
```
True or False = x or False
print(x)
```

4.
```
x,y = 9,10
z = x < y and x == 3**2
print(z)
```

5.
```
a,b = 7,6
a = b
x = a >= b + 1
print(x)
```

6.
```
x = 3^2
y = 9
print(x == y)
```

Booleans - References

- Think Python, Chapter 5, Conditional instructions and recursion[169] , in particular Sctions 5.2 and 5.3, Boolean expressions[170] You can skip recursion

5.2.8 Real numbers

Python saves the real numbers (floating point numbers) in 64 bit of information divided by sign, expnonent and mantissa (also called significand). Let's see an example:

[90]:
```
3.14
```

[90]:
```
3.14
```

[91]:
```
type(3.14)
```

[91]:
```
float
```

> **WARNING: you must use the dot instead of comma!**
>
> So you will write 3.14 instead of 3,14
>
> Be very careful whenever you copy numbers from documents in latin languages, they might contain very insidious commas!

Scientifical notation

Whenever numbers are very big or very small, to avoid having to write too many zeros it is convenient to use scientifical notation with the *e* like xen which multiplies the number *x* by 10^n

With this notation, in memory are only put the most significative digits (the *mantissa*) and the exponent, thus avoiding to waste space.

[92]:
```
75e1
```

[92]:
```
750.0
```

[169] http://greenteapress.com/thinkpython2/html/thinkpython2006.html
[170] http://greenteapress.com/thinkpython2/html/thinkpython2006.html#sec59

[93]:
```
75e2
```

[93]:
```
7500.0
```

[94]:
```
75e3
```

[94]:
```
75000.0
```

[95]:
```
75e123
```

[95]:
```
7.5e+124
```

[96]:
```
75e0
```

[96]:
```
75.0
```

[97]:
```
75e-1
```

[97]:
```
7.5
```

[98]:
```
75e-2
```

[98]:
```
0.75
```

[99]:
```
75e-123
```

[99]:
```
7.5e-122
```

QUESTION: Look at the following expressions, and try to find which result they produce (or if they give and error):

1. `print(1.000.000)`
2. `print(3,000,000.000)`
3. `print(2000000.000)`
4. `print(2000000.0)`
5. `print(0.000.123)`
6. `print(0.123)`
7. `print(0.-123)`
8. `print(3e0)`
9. `print(3.0e0)`
10. `print(7e-1)`
11. `print(3.0e2)`
12. `print(3.0e-2)`

13.
```
print(3.0-e2)
```

14.
```
print(4e2-4e1)
```

Too big or too small numbers

Sometimes calculations on very big or extra small numbers may give as a result `math.nan` (Not a Number) or `math.inf`. For the moment we just mention them, you can find a detailed description in the Numpy page[171]

Exercise - circle

⊛ Calculate the area of a circle at the center of a soccer ball (radius = 9.1m), remember that $area = pi * r^2$

Your code should print as result `263.02199094102605`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[100]:
```
# SOLUTION

r = 9.15
pi = 3.1415926536
area = pi*(r**2)
print(area)
```

```
263.02199094102605
```

</div>

[100]:
```
```

```
263.02199094102605
```

Note that the parenthesis around the squared `r` are not necessary because the power operator has the precedence, but htey may help in augmenting the code readability.

We recall here the operator precedence:

Operatore	Descrizione
`**`	Power (maximum precedence)
`+ -`	unary plus and minus
`* / // %`	Multiplication, division, integer division, modulo
`+ -`	Addition and subtraction
`<= < > >=`	comparison operators
`== !=`	equality operators
`not or and`	Logical operators (minimum precedence)

[171] https://en.softpython.org/matrices-numpy/matrices-numpy-sol.html#NaN-e-infinities

Exercise - fractioning

⊛ Write some code to calculate the value of the following formula for `x = 0.000003`, you should obtain `2.753278226511882`

$$-\frac{\sqrt{x+3}}{\frac{(x+2)^3}{\log x}}$$

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[101]: x = 0.000003

# write here
import math
- math.sqrt(x+3) / (((x+2)**3)/math.log(x))
```

```
[101]: 2.753278226511882
```

</div>

```
[101]: x = 0.000003

# write here
```

```
[101]: 2.753278226511882
```

Exercise - summation

Write some code to calculate the value of the following expression (don't use cycles, write down all calculations), you should obtain `20.53333333333333`

$$\sum_{j=1}^{3} \frac{j^4}{j+2}$$

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[102]: # write here
((1**4) / (1+2)) + ((2**4) / (2+2)) + ((3**4) / (3+2))
```

```
[102]: 20.53333333333333
```

</div>

```
[102]: # write here
```

```
[102]: 20.53333333333333
```

Reals - conversion

If we want to convert a real to an integer, several ways are available:

Function	Description	Mathematical symbol	Result
`math.floor(x)`	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
`int(x)`	round x to inferior integer	$\lfloor 8.7 \rfloor$	8
`math.ceil(x)`	round x to superior integer	$\lceil 5.3 \rceil$	6
`round(x)`	round x to closest integer	$\lfloor 2.5 \rceil$	2
		$\lfloor 2.51 \rceil$	3

QUESTION: Look at the following expressions, and for each of them try to guess which result it produces (or if it gives an error).

1. `math.floor(2.3)`
2. `math.floor(-2.3)`
3. `round(3.49)`
4. `round(3.5)`
5. `round(3.51)`
6. `round(-3.49)`
7. `round(-3.5)`
8. `round(-3.51)`
9. `math.ceil(8.1)`

10. `math.ceil(-8.1)`

QUESTION: Given a float `x`, the following formula is:

```
math.floor(math.ceil(x)) == math.ceil(math.floor(x))
```

1. always `True`
2. always `False`
3. sometimes `True` and sometimes `False` (give examples)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: for integers like `x=2.0` it is `True`, in other cases like `x=2.3` it is `False`

</div>

QUESTION: Given a float `x`, the following formula is:

```
math.floor(x) == -math.ceil(-x)
```

1. always `True`
2. always `False`
3. sometimes `True` and sometimes `False` (give examples)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1.

</div>

Exercise - Invigorate

⊛ Excessive studies lead you search on internet recipes of energetic drinks. Luckily, a guru of nutrition just posted on her Instagram channel @DrinkSoYouGetHealthy this recipe of a miracle drink:

> Pour in a mixer 2 decilitres of kiwi juice, 4 decilitres of soy sauce, and 3 decilitres of shampoo of karitè bio. Mix vigorously and then pour half drink into a glass. Fill the glass until the superior deciliter. Swallow in one shot.

You run to shop the ingredients, and get ready for mixing them. You have a measuring cup with which you transfer the precious fluids, one by one. While transfering, you always pour a little bit more than necessary (but never more than 1 decilitre), and for each ingredient you then remove the excess.

- **DO NOT** use subtractions, try using only rounding operators

Example - given:

```
kiwi = 2.4
soia = 4.8
shampoo = 3.1
measuring_cup = 0.0
mixer = 0
glass = 0.0
```

Your code must print:

```
I pour into the measuring cup 2.4 dl of kiwi juice, then I remove excess until␣
↪keeping 2 dl
I transfer into the mixer, now it contains 2.0 dl
I pour into the measuring cup 4.8 dl of soia, then I remove excess until keeping 4 dl
I transfer into the mixer, now it contains 6.0 dl
I pour into the measuring cup 3.1 dl of shampoo, then I remove excess until keeping 3␣
↪dl
I transfer into the mixer, now it contains 9.0 dl
I pour half of the mix ( 4.5 dl ) into the glass
I fill the glass until superior deciliter, now it contains: 5 dl
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[103]:
```
kiwi = 2.4
soy = 4.8
shampoo = 3.1
measuring_cup = 0.0
mixer = 0.0
glass = 0.0


# write here
print('I pour into the measuring cup', kiwi, 'dl of kiwi juice, then I remove excess␣
↪until keeping', int(kiwi), 'dl')
mixer += int(kiwi)
print('I transfer into the mixer, now it contains', mixer, 'dl')
print('I pour into the measuring cup', soy, 'dl of soia, then I remove excess until␣
↪keeping', int(soy), 'dl')
mixer += int(soy)
print('I transfer into the mixer, now it contains', mixer, 'dl')
print('I pour into the measuring cup', shampoo, 'dl of shampoo, then I remove excess␣
↪until keeping', int(shampoo), 'dl')
mixer += int(shampoo)
print('I transfer into the mixer, now it contains', mixer, 'dl')
bicchiere = mixer/2
print('I pour half of the mix (', glass, 'dl ) into the glass')
print('I fill the glass until superior deciliter, now it contains:', math.ceil(glass),
↪ 'dl')
```

```
I pour into the measuring cup 2.4 dl of kiwi juice, then I remove excess until␣
↪keeping 2 dl
I transfer into the mixer, now it contains 2.0 dl
I pour into the measuring cup 4.8 dl of soia, then I remove excess until keeping 4 dl
I transfer into the mixer, now it contains 6.0 dl
I pour into the measuring cup 3.1 dl of shampoo, then I remove excess until keeping 3␣
↪dl
I transfer into the mixer, now it contains 9.0 dl
I pour half of the mix ( 0.0 dl ) into the glass
I fill the glass until superior deciliter, now it contains: 0 dl
```

</div>

[103]:
```
kiwi = 2.4
soy = 4.8
shampoo = 3.1
measuring_cup = 0.0
```

(continues on next page)

(continued from previous page)

```
mixer = 0.0
glass = 0.0


# write here
```

```
I pour into the measuring cup 2.4 dl of kiwi juice, then I remove excess until␣
↪keeping 2 dl
I transfer into the mixer, now it contains 2.0 dl
I pour into the measuring cup 4.8 dl of soia, then I remove excess until keeping 4 dl
I transfer into the mixer, now it contains 6.0 dl
I pour into the measuring cup 3.1 dl of shampoo, then I remove excess until keeping 3␣
↪dl
I transfer into the mixer, now it contains 9.0 dl
I pour half of the mix ( 0.0 dl ) into the glass
I fill the glass until superior deciliter, now it contains: 0 dl
```

Exercise - roundminder

⊛ Write some code to calculate the value of the following formula for `x = -5.50`, you should obtain `41`

$$|\lceil x \rceil| + \lfloor x \rceil^2$$

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[104]:
```
x = -5.50    # 41
#x = -5.49   # 30

# write here
abs(math.ceil(x)) + round(x)**2
```

[104]:
```
41
```

</div>

[104]:
```
x = -5.50    # 41
#x = -5.49   # 30

# write here
```

[104]:
```
41
```

Reals - equality

> **WARNING: what follows is valid for *all* programming languages!**
>
> Some results will look weird but this is the way most processors (CPU) operates, independently from Python.

When floating point calculations are performed, the processor may introduce rounding errors due to limits of internal representation. Under the hood the numbers like floats are memorized in a sequence of binary code of 64 bits, according to *IEEE-754 floating point arithmetic* standard: this imposes a physical limit to the precision of numbers, and sometimes we ight get surprises due to conversion from decimal to binary. For example, let's try printing `4.1`:

[105]:
```
print(4.1)
```

```
4.1
```

For our convenience Python is showing us `4.1`, but in reality in the processor memory ended up a different number! Which one? To discover what it hides, with `format` function we can explicitly format the number to, for example 55 digits of precision by using the `f` format specifier:

[106]:
```
format(4.1, '.55f')
```

[106]:
```
'4.0999999999999996447286321199499070644378662109375000000'
```

We can then wonder what the result of this calculus might be:

[107]:
```
print(7.9 - 3.8)
```

```
4.1000000000000005
```

We note the result is still different from the expected one! By investigating further, we notice Python is not even showing all the digits:

[108]:
```
format(7.9 - 3.8, '.55f')
```

[108]:
```
'4.1000000000000005329070518200751394033432006835937500000'
```

[ ]:
```

```

What if wanted to know if the two calculations with float produce the 'same' result?

> **WARNING: AVOID THE == WITH FLOATS!**

To understand if the result between the two calculations with the flots is the same, **YOU CANNOT** use the == operator !

[109]:
```
7.9 - 3.8 == 4.1     # TROUBLE AHEAD!
```

[109]:
```
False
```

Instead, you should prefer alternative that evaluate if a float number is *close* to anoter, like for example the handy function math.isclose[172]:

[110]:
```
import math

math.isclose(7.9 - 3.8, 4.1)    # MUCH BETTER
```

[172] https://docs.python.org/3/library/math.html#math.isclose

[110]:
```
True
```

By default `math.isclose` uses a precision of `1e-09`, but, if needed, you can also pass a tolerance limit in which the difference of the numbers must be so to be considered equal:

[111]:
```
math.isclose(7.9 - 3.8, 4.1, abs_tol=0.000001)
```

[111]:
```
True
```

QUESTION: Can we perfectly represnt the number $\sqrt{2}$ as a `float`?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: $\sqrt{2}$ is irrational so there's no hope of a perfect representation, any calculation will always have a certain degree of imprecision.

</div>

QUESTION: Which of these expressions give the same result?

```
import math
print('a)', math.sqrt(3)**2 == 3.0)
print('b)', abs(math.sqrt(3)**2 - 3.0) < 0.0000001)
print('c)', math.isclose(math.sqrt(3)**2, 3.0, abs_tol=0.0000001))
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: b) and c) give `True`. a) gives `False`, because durting floating point calculations rounding errors are made.

</div>

Exercise - quadratic

⊛ Write some code to calculate the zeroes of the equation $ax^2 - b = 0$

- Show numbers with **20 digits** of precision
- At the end check that by substituting the value obtained `x` into the equation you actually obtain zero.

Example - given:

```
a = 11.0
b = 3.3
```

after your code it must print:

```
11.0 * x**2 - 3.3 = 0  per x1 = 0.54772255750516607442
11.0 * x**2 - 3.3 = 0  per x2 = -0.54772255750516607442
0.5477225575051661 is a solution? True
-0.5477225575051661 is a solution? True
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[112]:
```
a = 11.0
b = 3.3
```

(continues on next page)

(continued from previous page)

```
# write here

import math

x1 = math.sqrt(b/a)
x2 = -math.sqrt(b/a)

print(a, "* x**2 -",b,"= 0  per x1 =", format(x1, '.20f'))
print(a, "* x**2 -",b,"= 0  per x2 =", format(x2, '.20f'))

# we need to change the default tolerance value
print(format(x1, '.20f'), "is a solution?", math.isclose(a*(x1**2) - b,0, abs_tol=0.
↪00001))
print(format(x2, '.20f'), "is a solution?", math.isclose(a*((x2)**2) - b,0, abs_tol=0.
↪00001))
```

```
11.0 * x**2 - 3.3 = 0  per x1 = 0.54772255750516607442
11.0 * x**2 - 3.3 = 0  per x2 = -0.54772255750516607442
0.54772255750516607442 is a solution? True
-0.54772255750516607442 is a solution? True
```

</div>

[112]:
```
a = 11.0
b = 3.3

# write here
```

```
11.0 * x**2 - 3.3 = 0  per x1 = 0.54772255750516607442
11.0 * x**2 - 3.3 = 0  per x2 = -0.54772255750516607442
0.54772255750516607442 is a solution? True
-0.54772255750516607442 is a solution? True
```

Exercise - trendy

⊛⊛ You are already thinking about next vacations, but there is a big problem: where do you go, if you don't have a *selfie-stick*. You cannot leave with this serious anxiety: to uniform yourself to this mass phenomena you must buy the stick which is most similar to others. You then conduct a rigourous statistical survey among turists obssessed by selfie sticks with the goal to find the most frequent brands of sticks, in other words, the *mode* of the frequencies. You obtain these results:

[113]:
```
b1,b2,b3,b4,b5 = 'TooManyLikes', 'Boombasticks', 'Timewasters Inc', 'Vanity 3.0',
↪'TrashTrend' # brand
f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.3   # frequencies (as percentages)
```

We deduce that masses love selfie-sticks of the brand `'Boombasticks'` and `TrashTrend`, both in a tie with 30% turists each. Write some code which prints this result:

```
TooManyLikes is the most frequent? False ( 25.0 % )
Boombasticks is the most frequent? True ( 30.0 % )
Timewasters Inc is the most frequent? False ( 10.0 % )
Vanity 3.0 is the most frequent? False ( 5.0 % )
TrashTrend is the most frequent? True ( 30.0 % )
```

- **WARNING**: your code must work with **ANY** series of variables !!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[114]:
```
b1,b2,b3,b4,b5 = 'TooManyLikes', 'Boombasticks', 'Timewasters Inc', 'Vanity 3.0',
↪'TrashTrend'    # brand

f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.3  # frequencies (as percentages)  False␣
↪True False False True
# CAREFUL, they look the same but it must work also with these!
#f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.1 + 0.2  #  False True False False True

# write here
mx = max(f1,f2,f3,f4,f5)
print(b1, 'is the most frequent?', math.isclose(f1,mx), '(', format(f1*100.0, '.1f'),'
↪% )')
print(b2, 'is the most frequent?', math.isclose(f2,mx), '(', format(f2*100.0, '.1f'),'
↪% )')
print(b3, 'is the most frequent?', math.isclose(f3,mx), '(', format(f3*100.0, '.1f'),'
↪% )')
print(b4, 'is the most frequent?', math.isclose(f4,mx), '(', format(f4*100.0, '.1f'),'
↪% )')
print(b5, 'is the most frequent?', math.isclose(f5,mx), '(', format(f5*100.0, '.1f'),'
↪% )')
```

```
TooManyLikes is the most frequent? False ( 25.0 % )
Boombasticks is the most frequent? True ( 30.0 % )
Timewasters Inc is the most frequent? False ( 10.0 % )
Vanity 3.0 is the most frequent? False ( 5.0 % )
TrashTrend is the most frequent? True ( 30.0 % )
```

</div>

[114]:
```
b1,b2,b3,b4,b5 = 'TooManyLikes', 'Boombasticks', 'Timewasters Inc', 'Vanity 3.0',
↪'TrashTrend'    # brand

f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.3  # frequencies (as percentages)  False␣
↪True False False True
# CAREFUL, they look the same but it must work also with these!
#f1,f2,f3,f4,f5 = 0.25, 0.3, 0.1, 0.05, 0.1 + 0.2  #  False True False False True

# write here
```

```
TooManyLikes is the most frequent? False ( 25.0 % )
Boombasticks is the most frequent? True ( 30.0 % )
Timewasters Inc is the most frequent? False ( 10.0 % )
Vanity 3.0 is the most frequent? False ( 5.0 % )
TrashTrend is the most frequent? True ( 30.0 % )
```

5.2.9 Decimal numbers

For most applications float numbers are sufficient, if you are conscius of their limits of representation and equality. If you really need more precision and/or preditability, Python offers a dedicated numeric type called `Decimal`, which allows arbitrary precision. To use it, you must first import `decimal` library:

[115]:
```
from decimal import Decimal
```

You can create a `Decimal` from a string:

[116]:
```
Decimal('4.1')
```

[116]:
```
Decimal('4.1')
```

WARNING: if you create a Decimal from a costant, use a string!

If you pass a `float` you risk losing the utility of Decimals:

[117]:
```
Decimal(4.1)  # this way I keep the problems of floats ...
```

[117]:
```
Decimal('4.0999999999999996447286321199499070644378662109375')
```

Operations between `Decimals` produce other `Decimals`:

[118]:
```
Decimal('7.9') - Decimal('3.8')
```

[118]:
```
Decimal('4.1')
```

This time, we can freely use the equality operator and obtain the same result:

[119]:
```
Decimal('4.1') == Decimal('7.9') - Decimal('3.8')
```

[119]:
```
True
```

Some mathematical functions are also supported, and often they behave more predictably (note we are **not** using `math.sqrt`):

[120]:
```
Decimal('2').sqrt()
```

[120]:
```
Decimal('1.414213562373095048801688724')
```

Remember: computer memory is still finite!

Decimals can't be solve all problems in the universe: for example, $\sqrt{2}$ will never fit the memory of any computer! We can verify the limitations by squaring it:

[121]:
```
Decimal('2').sqrt()**Decimal('2')
```

[121]:
```
Decimal('1.999999999999999999999999999')
```

The only thing we can have more with Decimals is more digits to represent numbers, which if we want we can increase at will[173] until we fill our pc memory. In this book we won't talk anymore about Decimals because typically they are meant only for specific applications, for example, if you need to perform fincancial calculations you will probably want very exact digits!

[173] https://docs.python.org/3/library/decimal.html

5.2.10 Challenges

We now propose some (very easy) exercises without solutions.

Try to execute them both in Jupyter and a text editor such as Spyder or Visual Studio Code to get familiar with both environments.

Challenge - which booleans 1?

⊕ Find the row with values such that the final print prints `True`. Is there only one combination or many?

[122]:

```python
x = False; y = False
#x = False; y = True
#x = True; y = False
#x = True; y = True

print(x and y)
```

```
False
```

Challenge - which booleans 2?

⊕ Find the row in which by assigning values to `x` and `y` it prints `True`. Is there only one combinatin or many?

[123]:

```python
x = False; y = False; z = False
#x = False; y = True; z = False
#x = True; y = False; z = False
#x = True; y = True; z = False
#x = False; y = False; z = True
#x = False; y = True; z = True
#x = True; y = False; z = True
#x = True; y =True; z =True

print((x or y) and (not x and z))
```

```
False
```

Challenge - Triangle area

⊕ Compute the area of a triangle having base 120 units (`b`) and height 33 (`h`). Assign the result to a variable named `area` and print it. Your code should show `Triangle area is: 120.0`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[124]:

```python
# write here
```

</div>

[124]:

```python
# write here
```

Challenge - square area

⊕ Compute the area of a square having side (`s`) equal to 145 units. Assign the result to a variable named `area` and print it, it should show `Square area is: 21025`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[125]: # write here
```

</div>

```
[125]: # write here
```

Challange - area from input

⊕ Modify the program at previous point. to acquire the side `s` from the user at runtime.

Hint: use the input[174] function and remember to convert the acquired value into an `int`). NOTE: from our experimentations, `input` tends to have problems in Jupyter so you'd better try in some other editor.

Try also to put the two previous scripts in two separate files (e.g. `triangle_area.py` and `square_area.py` and execute them from the terminal)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[126]: # write here
```

</div>

```
[126]: # write here
```

Challenge - trapezoid

⊕ Write a small script (`trapezoid.py`) that computes the area of a trapezoid having major base (`mj`) equal to 30 units, minor base (mn) equal to 12 and height (`h`) equal to 17. Print the resulting area. Try executing the script from a text editor like Spyder or Visual Studio Code and from the terminal.

It should print `Trapezoid area is: 357.0`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[127]: # write here
```

</div>

[174] https://www.geeksforgeeks.org/taking-input-in-python/

[127]:
```python
# write here
```

Challenge - first n numbers

⊛ Rewrite the example of the sum of the first 1200 integers by using the following equation:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Then modify the program to make it acquire the number of integers to sum N from the user at runtime

It should show `Sum of first 1200 integers: 720600.0`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

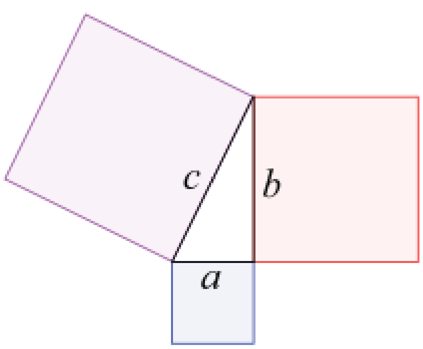
[128]:
```python
# write here
```

</div>

[128]:
```python
# write here
```

challenge - hypotenuse

Write a small script to compute the length of the hypotenuse (`c`) of a right triangle having sides `a`=133 and `b`=72 units (see picture below). It should print `Hypotenuse: 151.23822268196622`



Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[129]:
```python
# write here
```

</div>

[129]:
```python
# write here
```

Challenge - which integers 1?

⊛ Assign numerical values to x y e z to have the expression print True

[130]:
```
x = 0   # ?
y = 0   # ?
print(max(min(x,y), x + 20) > 20)
```

```
False
```

Challenge - which integers 2?

⊛ Assign to x and y values such that True is printed

[131]:
```
x = 0   # ?
y = 0   # ?

print(x > 10 and x < 23 and ((x+y) == 16 or (x + y > 20)))
```

```
False
```

Challenge - which integers 3?

⊛ Assign to z and w values such that True is printed.

[132]:
```
z = 0 # ?
w = 1 # ?
(z < 40 or w < 90) and (z % w > 2)
```

[132]:
```
False
```

Challenge - airport

⊛⊛ You finally decide to take a vacation and go to the airport, expecting to spend some time in several queues. Luckily, you only have carry-on bag, so you directly go to security checks, where you can choose among three rows of people sec1, sec2, sec3. Each person an average takes 4 *minutes* to be examinated, you included, and obviously you choose the shortest queue. Afterwards you go to the gate, where you find two queues of ga1 and ga2 people, and you know that each person you included an average takes 20 *seconds* to pass: again you choose the shortes queue. Luckily the aircraft is next to the gate so you can directly choose whether to board at the queue at the head of the aircraft with bo1 people or at the queue at the tail of the plane with bo2 people. Each passenger you included takes an average 30 *seconds,* and you choose the shortest queue.

Write some code to calculate how much time you take in total to enter the plane, showing it in minutes and seconds.

Example - given:

```
sec1,sec2,sec3, ga1,ga2, bo1,bo2 = 4,5,8, 5,2, 7,6
```

your code must print:

```
24 minutes e 30 seconds
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[133]:
```
sec1,sec2,sec3, ga1,ga2, bo1,bo2 = 4,5,8, 5,2, 7,6   # 24 minutes e 30 seconds
#sec1,sec2,sec3, ga1,ga2, bo1,bo2 = 9,7,1, 3,5, 2,9  # 10 minutes e 50 seconds

# write here
```

</div>

[133]:

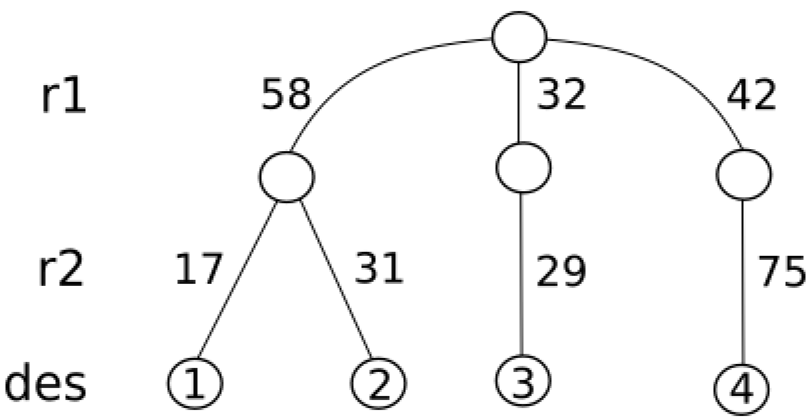
```
sec1,sec2,sec3, ga1,ga2, bo1,bo2 = 4,5,8, 5,2, 7,6   # 24 minutes e 30 seconds
#sec1,sec2,sec3, ga1,ga2, bo1,bo2 = 9,7,1, 3,5, 2,9  # 10 minutes e 50 seconds

# write here
```


Challenge - Holiday trip

⊛⊛ While an holiday you are traveling by car, and in a particular day you want to visit one among 4 destinations. Each location requires to go through two roads `r1` and `r2`. Roads are numbered with two digits numbers, for example to reach destination `1` you need to go to road `58` and road `17`.

Write some code that given `r1` and `r2` roads shows the number of the destination.

- If the car goes to a road it shouldn't (i.e. road 666), put `False` in `destination`
- **DO NOT** use summations
- **IMPORTANT: DO NOT use `if` commands** (it's possible, think about it ;-)



Example 1 - given:

```
r1,r2 = 58,31
```

After your code it must print:

```
The destination is 2
```

Example 2 - given:

```
r1,r2 = 666,31
```

After your code it must print:

```
The destination is False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[134]:
```
r1,r2 = 58,17    # 1
r1,r2 = 58,31    # 2
r1,r2 = 32,29    # 3
r1,r2 = 42,75    # 4
r1,r2 = 666,31   # False
r1,r2 = 58,666   # False
r1,r2 = 32,999   # False

# write here
```

</div>

[134]:
```
r1,r2 = 58,17    # 1
r1,r2 = 58,31    # 2
r1,r2 = 32,29    # 3
r1,r2 = 42,75    # 4
r1,r2 = 666,31   # False
r1,r2 = 58,666   # False
r1,r2 = 32,999   # False

# write here
```

5.2.11 References

- Think Python - Chapter 1[175]: The way of the program
- Think Python - Chapter 2[176]: Variables, expressions and statements

[ ]:
```
```

5.3 Strings 1 - introduction

5.3.1 Download exercises zip

Browse files online[177]

Strings are *immutable* character sequences, and one of the basic Python types. In this notebook we will see how to manipulate them.

[175] http://greenteapress.com/thinkpython2/html/thinkpython2002.html
[176] http://greenteapress.com/thinkpython2/html/thinkpython2003.html
[177] https://github.com/DavidLeoni/softpython-it/tree/master/strings

5.3.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
strings
    strings1.ipynb
    strings1-sol.ipynb
    strings2.ipynb
    strings2-sol.ipynb
    strings3.ipynb
    strings3-sol.ipynb
    strings4.ipynb
    strings4-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `strings1.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.3.3 Creating strings

There are several ways to define a string.

Double quotes, in one line

[2]:
```
a = "my first string, in double quotes"
```

[3]:
```
print(a)
```

```
my first string, in double quotes
```

Single quotes, in one line

This way is equivalent to previous one.

[4]:
```
b = 'my second string, in single quotes'
```

[5]:
```
print(b)
```

```
my second string, in single quotes
```

Between double quotes, on many lines

[6]:
```python
c = """my third string
in triple double quotes
so I can put it

on many rows"""
```

[7]:
```python
print(c)
```

```
my third string
in triple double quotes
so I can put it

on many rows
```

Three single quotes, many lines

[8]:
```python
d = '''my fourth string,
in triple single quotes
also can be put

on many lines
'''
```

[9]:
```python
print(d)
```

```
my fourth string,
in triple single quotes
also can be put

on many lines

```

5.3.4 Printing - the cells

To print a string we can use the function `print`:

[10]:
```python
print('hello')
```

```
hello
```

Note that apices are *not* reported in printed output.

If we write the string without the `print`, we will see the apices indeed:

[11]:
```python
'hello'
```

[11]:
```
'hello'
```

What happens if we write the string with double quotes?

[12]:
```python
"hello"
```

[12]:
```
'hello'
```

Notice that by default Jupyter shows single apices.

The same applies if we assign a string to a variable:

[13]:
```
x = 'hello'
```

[14]:
```
print(x)
```

```
hello
```

[15]:
```
x
```

[15]:
```
'hello'
```

[16]:
```
y = "hello"
```

[17]:
```
print(y)
```

```
hello
```

[18]:
```
y
```

[18]:
```
'hello'
```

5.3.5 The empty string

The string of zero length is represented with two double quotes `""` or two single apices `''`

Note that even if write two double quotes, Jupter shows a string beginning and ending with single apices:

[19]:
```
""
```

[19]:
```
''
```

The same applies if we associate an empty string to a variable:

[20]:
```
x = ""
```

[21]:
```
x
```

[21]:
```
''
```

Note that even if we ask Jupyter to use `print`, we won't see anything:

[22]:
```
print("")
```

```

```

[23]:
```
print('')
```

```

```

5.3.6 Printing many strings

For printing many strings on a single line there are different ways, let's start from the most simple with `print`:

[24]:
```
x = "hello"
y = "Python"

print(x,y)    # note that in the printed characters Python inserted a space:
```

```
hello Python
```

We can add to `print` as many parameters we want, which can also be mixed with other types like numbers:

[25]:
```
x = "hello"
y = "Python"
z = 3

print(x,y,z)
```

```
hello Python 3
```

5.3.7 Length of a string

To obtain the length of a string (or any sequence in general), we can use the function `len`:

[26]:
```
len("ciao")
```

[26]:
```
4
```

[27]:
```
len("")    # empty string
```

[27]:
```
0
```

[28]:
```
len('')    # empty string
```

[28]:
```
0
```

QUESTION: Can we write something like this?

```
"len"("hello")
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: no, `"len"` between quotes will be interpreted as a string, not as a function, so Python will complain telling us we cannot apply a string to another string. Try to see which error appears by rewriting the expression below:

</div>

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[29]:
```
# write here

#"len"("hello")
```

</div>

```
[29]: # write here
```

QUESTION: can we write something like this? What does it produce? an error? a number? which one?

```
len("len('hello')")
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: it returns the number `12`: by putting the Python code `len('hello')` among double quotes, it became a string like any other. So by writing `len("len('hello')")` we count how long the string `"len('hello')"` is.

</div>

QUESTION: What do we obtain if we write like this?

```
len((((("ciao")))))
```

1. an error
2. the length of the string
3. something else

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The second: `"ciao"` is an expression, as such we can enclose it in as many parenthesis as we want.

</div>

Counting escape sequences: Note that some particular sequences called *escape sequences* like for example `\t` occupy less space of what it seems (with `len` they count as 1), but if we print them they will occupy even more than 2 !!

Let's see an example (in the next paragraph we will delve into the details):

```
[30]: len('a\tb')
```

```
[30]: 3
```

```
[31]: print('a\tb')
```

```
a	b
```

5.3.8 Printing - escape sequences

Some characters sequences called *escape sequences* are special because instead of showing characters, they force the printing to do particular things like line feed or inserting extra spaces. These sequences are always preceded by the *backslash* character `\`:

Description	*Escape* sequence
Linefeed	`\n`
Tabulation *(ASCII tab)*	`\t`

Esempio - line feed

```
[32]: print("hello\nworld")
```

```
hello
world
```

Note the line feed happens only when we use `print`, if instead we directly put the string into the cell we will see it verbatim:

```
[33]: "ciao\nmondo"
```

```
[33]: 'ciao\nmondo'
```

In a string you can put as many escape sequences as you like:

```
[34]: print("Today is\na great day\nisn't it?")
```

```
Today is
a great day
isn't it?
```

Example - tabulation

```
[35]: print("hello\tworld")
```

```
hello   world
```

```
[36]: print("hello\tworld\twith\tmany\ttabs")
```

```
hello   world   with    many    tabs
```

EXERCISE: Since *escape sequences* are special, we might ask ourselves how long they are. Use the function `len` to print the string length. Do you notice anything strange?

- `'ab\ncd'`
- `'ab\tcd'`

```
[37]: # write here
```

EXERCISE: Try selecting the character sequence printed in the previous cell with the mouse. What do you obtain? A space sequence, or a single tabulation character? Note this can vary according to the program that actually printed the string.

EXERCISE: find a SINGLE string which printed with `print` is shown as follows:

```
This    is
an

apparently  simple      challenge
```

- USE ONLY combinations of `\t` and `\n`
- DON'T use spaces
- start and end the string with a single apex

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[38]:
```
# write here

print('This\tis\nan\n\napparently\tsimple\t\tchallenge')
```

```
This	is
an

apparently	simple		challenge
```

</div>

[38]:
```
# write here
```

```
This	is
an

apparently	simple		challenge
```

EXERCISE: try to find a string which printed with `print` is shown as follows:

```
At	te
n

	t	ion
	please!
```

- USE ONLY combinations of `\t` and `\n`
- DON'T use any space
- DON'T use triple quotes

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[39]:
```
# write here
print("At\tte\nn\n\n\tt\tion\n\tplease!")
```

```
At	te
n

	t	ion
	please!
```

</div>

[39]:
```
# write here
```

```
At	te
n

	t	ion
	please!
```

Special character: if we want special characters like the single apex `'` or double quotes `"` inside a string, we must create a so-called *escape sequence*, that is, we must first write the *backslash* character `\` and then follow it with the special character we're interested in:

Description	Escape sequence	Printed result
Single apex	`\'`	`'`
Double quote	`\"`	`"`
Backslash	`\\`	`\`

Example:

Let's print a string containing a single apex `'` and a double quote `"`:

```
[40]: my_string = "This way I put \'apices\' e \"double quotes\" in strings"
```

```
[41]: print(my_string)
```

```
This way I put 'apices' e "double quotes" in strings
```

If a string begins with double quotes, inside we can freely use single apices, even without *backslash* `\`:

```
[42]: print("There's no problem")
```

```
There's no problem
```

If the string begins with single apices, we can freely use double quotes even without the *backslash* `\`:

```
[43]: print('It Is So "If You Think So"')
```

```
It Is So "If You Think So"
```

EXERCISE: Find a string to print with `print` which shows the following sequence:

- the string MUST start and finish with single apices `'`

```
This "genius" of strings wants to /\\/ trick me \//\ with atrocious exercises O_o'
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[44]: # write here

print('This "genius" of strings wants to /\\\\/ trick me \\//\\ with atrocious␣
↪exercises O_o\'')
```

```
This "genius" of strings wants to /\\/ trick me \//\ with atrocious exercises O_o'
```

</div>

```
[44]: # write here
```

```
This "genius" of strings wants to /\\/ trick me \//\ with atrocious exercises O_o'
```

5.3.9 Encodings

ASCII characters

When using strings in your daily programs you typically don't need to care much how characters are physically represented as bits in memory, but sometimes it does matter. The representation is called *encoding* and must be taken into account in particular when you read stuff from external sources such as files and websites.

The most famous and used character encoding is ASCII[178] (American Standard Code for Information Interchange), which offers 127 slots made by basic printable characters from English alphabet (`a-z`, `A-Z`, punctuation like `.;,!` and characters like `(`, `@` ...) and control sequences (like `\t`, `\n`)

- See Printable characters[179] (Wikipedia)
- ASCII Control codes[180] (Wikipedia)

Original ASCII table lacks support for non-English languages (for example, it lacks Italian accented letters like `è`,`à`, ...), so many extensions were made to support other languages, for examples see Extended ASCII[181] page on Wikipedia.

Unicode characters

Whenever we need particular characters like ✪ which are not available on the keyboard, we can look at Unicode characters. There are a lot[182], and we can often use them in Python 3 by simple copy-pasting. For example, if you go to this page[183] we can copy-paste the character ✪. In other cases it might be so special it can't even be correctly visualized, so in these cases you can use a more complex sequence in the format `\uxxxx` like this:

Description	Escape sequence	Printed result
Example star in a circle in format `\uxxxx`	`\u272A`	✪

EXERCISE: Search Google for *Unicode heart* and try to print a hear in Python, both by directly copy-pasting the character and by using the notation `\uxxxx`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[45]: # write here

print("I ♥ Python, with copy-paste")
print("I \u2665 Python, also in format \\uxxxx")
```

```
I ♥ Python, with copy-paste
I ♥ Python, also in format \uxxxx
```

</div>

```
[45]: # write here
```

178 https://en.wikipedia.org/wiki/ASCII
179 https://en.wikipedia.org/wiki/ASCII#Printable_characters
180 https://en.wikipedia.org/wiki/C0_and_C1_control_codes#Basic_ASCII_control_codes
181 https://en.wikipedia.org/wiki/Extended_ASCII
182 http://www.fileformat.info/info/unicode/char/a.htm
183 https://www.fileformat.info/info/unicode/char/272a/index.htm

```
I ♥ Python, with copy-paste
I ♥ Python, also in format \uxxxx
```

Unicode references: Unicode can be a complex topic we just mentioned, if you ever need to deal with complex character sets like japanese or heterogenous text encodings here a couple of references you should read:

- first part on Unicode encoding from Strings chapter from book Dive into Python 3[184]
- Python 3 Unicode[185] documentation

5.3.10 Strings are immutable

Strings are *immutable* objects, so once they are created you cannot change them anymore. This might appear retrictive, but it's not so tragic, because we still have available these alternatives:

- generate a new string composed from other strings
- if we have a variable to which we assigned a string, we can assign another string to that variable

Let's generate a new string starting from previous ones, for example by joining two of them with the operator +

[46]:
```
x = 'hello'
```

[47]:
```
y = x + 'world'
```

[48]:
```
x
```

[48]:
```
'hello'
```

[49]:
```
y
```

[49]:
```
'helloworld'
```

The + operation, when executed among strings, it joins them by creating a NEW string. This means that the association to `x` it didn't change at all, the only modification we can observe will be the variable `y` which is now associated to the string `'helloworld`. Try making sure of this in Python Tutor by repeatdly clicking on *Next* button:

[50]:
```
# WARNING: before using the function jupman.pytut() which follows,
# it is necessary to first execute this cell with Shift+Enter

# it's sufficient to execute it only once, you find it also in all other notebooks in␣
↪the first cell

import jupman
```

[51]:
```
x = 'hello'
y = x + 'world'

print(x)
print(y)

jupman.pytut()
```

```
hello
helloworld
```

[184] https://diveintopython3.net/strings.html
[185] https://docs.python.org/3/howto/unicode.html

```
[51]: <IPython.core.display.HTML object>
```

Reassign variables

Other variations to memory state can be obtained by reassigning the variables, for example:

```
[52]: x = 'hello'
```

```
[53]: y = 'world'
```

```
[54]: x = y         # we assign to x the same string contained in y
```

```
[55]: x
```

```
[55]: 'world'
```

```
[56]: y
```

```
[56]: 'world'
```

If a string is created and at some point no variables point to it, Python automatically takes care to eliminate it from the memory. In the case above, the string `hello` is never actually changed: at some point no variable is associated with it anymore and so Python eliminates the string from the memory. Have a look at what happens in Python Tutor:

```
[57]: x = 'hello'
      y = 'world'
      x = y

      jupman.pytut()
```

```
[57]: <IPython.core.display.HTML object>
```

Reassign a variable to itself

We may ask ourselves what happens when we write something like this:

```
[58]: x = 'hello'

      x = x
```

```
[59]: print(x)
```

```
hello
```

No big changes, the assignment of `x` remained the same without alterations.

But what happens if to the right of the = we put a more complex formula?

```
[60]: x = 'hello'

      x = x + 'world'

      print(x)
```

```
helloworld
```

Let's try to carefully understand what happened.

In the first line, Python generated the string `'hello'` and assigned it to the variable `x`. So far, nothing extraordinary.

Then, in the second line, Python did two things:

1. it calculated the result of the expression `x + 'world'`, by generating a NEW string `helloworld`
2. it assigned the generated string `helloworld` to the variable `x`

It is fundamental to understand that whenever a reassignment is performed both passages occurs, so it's worth repeating them:

- FIRST the result of the expression to the right of = is calculated (so when the old value of `x` is still available)
- THEN the result is associated to the variable to the left of = symbol

If we check out what happens in Python Tutor, this double passage is executed in a single shot:

[61]:
```
x = 'hello'
x = x + 'world'

jupman.pytut()
```

[61]:
```
<IPython.core.display.HTML object>
```

EXERCISE: Write some code that changes memory state in such a way so that in the end the following is printed:

```
z =  This
w =  was
x =  a problem
y =  was
s =  This was a problem
```

- to write the code, USE ONLY the symbols =,+,`z`,`w`,`x`,`y`,`s` AND NOTHING ELSE
- you can freely use as many lines of code as you deem necessary
- you can freely use any symbol as many times you deem necessary

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[62]:
```
# these variables are given

z = "This"
w = 'is'
x = 'a problem'
y = 'was'
s = ' '

# write here the code

w = y

s = z + s + y + s + x
```

</div>

[62]:
```
# these variables are given

z = "This"
w = 'is'
x = 'a problem'
y = 'was'
s = ' '

# write here the code
```

[63]:
```
print("z = ", z)
print("w = ", w)
print("x = ", x)
print("y = ", y)
print("s = ", s)
```

```
z =  This
w =  was
x =  a problem
y =  was
s =  This was a problem
```

5.3.11 Strings and numbers

Python strings have the type `str`:

[64]:
```
type("hello world")
```

[64]:
```
str
```

In strings we can insert characters which represent digits:

[65]:
```
print("The character 5 represents the digit five, the character 3 represents the␣
↪digit three")
```

```
The character 5 represents the digit five, the character 3 represents the digit three
```

Obviously, we can also substitute a sequence of digits, to obtain something which looks like a number:

[66]:
```
print("The sequence of characters 7583 represents the number seven thousand five␣
↪hundred eighty-three")
```

```
The sequence of characters 7583 represents the number seven thousand five hundred␣
↪eighty-three
```

Having said that, we can ask ourselves how Python behaves when we have a *string* which contains *only* a sequence of characters which represents a number, like for example `'254'`

Can we use `254` (which we wrote like it were a string) also as if it were a number? For example, can we sum `3` to it?

```
'254' + 3

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-d39aa62a7e3d> in <module>
```

(continues on next page)

(continued from previous page)

```
----> 1 "254" + 3

TypeError: can only concatenate str (not "int") to str
```

As you see, Python immediately complains, because we are trying to mix different types.

SO:

- by writing `'254'` between apices we create a *string* of type `str`
- by writing `254` we create a *number* of type `int`

```
[67]: type('254')
```

```
[67]: str
```

```
[68]: type(254)
```

```
[68]: int
```

> **BEWARE OF `print` !!**
>
> If you try to print a string which only contains digits, Python will show it without apices, and this might mislead you about its true nature !!

```
[69]: print('254')
```

```
254
```

```
[70]: print(254)
```

```
254
```

Only in Jupyter, to show constants, variables or results of calculations, as `print` alternative you can directly insert a formula in the cell. In this case we are simply showing a constant, and whenever it is a string you will see apices:

```
[71]: '254'
```

```
[71]: '254'
```

```
[72]: 254
```

```
[72]: 254
```

The same reasoning applies also to variables:

```
[73]: x = '254'
```

```
[74]: x
```

```
[74]: '254'
```

```
[75]: y = 254
```

```
[76]: y
```

[76]:
```
254
```

So, *only in Jupyter,* when you need to show a constant, a variable or a calculation often it's more convenient to directly write it in the cell without using `print`.

5.3.12 Conversions - from string to number

Let's go back to the problem of summing `'254' + 3`. The first one is a string, the second a number. If they were both numbers the sum would surely work:

[77]:
```python
254 + 3
```

[77]:
```
257
```

So we can try to convert the string `'254'` into an authentic integer. To do it, we can use `int` as if it were a function, and pass as argument the string to be converted:

[78]:
```python
int('254') + 3
```

[78]:
```
257
```

WARNING: strings and numbers are immutable !!

This means that by writing `int('254')'` a *new* number is generated without minimally affecting the string `'254'` from where we started from. Let's see am example:

[79]:
```python
x = '254'      # assign to variable x the string '254'
```

[80]:
```python
y = int(x)     # assign to variable y the number obtained by converting '254' in int
```

[81]:
```python
x              # variable x is now assigned to string '254'
```

[81]:
```
'254'
```

[82]:
```python
y              # in y now there is a number instead (note we don't have apices here)
```

[82]:
```
254
```

It might be useful to see again the example in Python Tutor:

[83]:
```python
x = "254"

y = int(x)

print(y + 3)


jupman.pytut()
```

```
257
```

[83]:
```
<IPython.core.display.HTML object>
```

EXERCISE: Try to convert a string which represents an ill-formed number (for example a number with inside a character: `'43K12'`) into an `int`. What happens?

[84]:
```
# write here
```

5.3.13 Conversions - from number to string

Any object can be converted to string by using `str` as if it were a function and by passing the object to convert. Let's try then to convert a number into a string.

[85]:
```
str(5)
```

[85]:
```
'5'
```

note the apices in the result, which show we actually obtained a string.

If by chance we want to obtain a string which is the concatenation of objects of different types we need to be careful:

```
x = 5
s = 'Workdays in a week are ' + x
print(s)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-154-5951bd3aa528> in <module>
      1 x = 5
----> 2 s = 'Workdays in a week are ' + x
      3 print(s)

TypeError: can only concatenate str (not "int") to str
```

A way to circumvent the problem (even if not the most convenient) is to convert into string each of the objects we're using in the concatenation:

[86]:
```
x = 3
y = 1.6
s = "This week I've been jogging " + str(x) + " times running at an average speed of
↪" + str(y) + " km/h"
print(s)
```

```
This week I've been jogging 3 times running at an average speed of 1.6 km/h
```

QUESTION: Having said that, after executing the code in previous cell, variable `x` is going to be associated to a *number* or a *string* ?

If you have doubts, use Python Tutor.

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: numbers, like strings, are immutable. So by calling the function `str(x)` it is impossible for the number `5` associated to `x` to be changed in any way. `str(x)` will simply generate a NEW string `'5'` which will then be used in the concatenation.

</div>

5.3.14 Formatting strings

Concatenating strings with plus sign like above is cumbersome and error prone. There are several better solutions, for a thorough review we refer to Real Python[186] website. In particular, check out the most handy which is f-strings[187] and available for Python >= 3.6

Formatting with %

Here we now see how to format strings with the `%` operator. This solution is not the best one, but it's widely used and supported in all Python versions, so we adopted it throughout the book:

[87]:
```python
x = 3
"I jumped %s times" % x
```

[87]:
```
'I jumped 3 times'
```

Notice we put a so-called *place-holder* `%s` inside the string, which tells Python to replace it with a variable. To feed Python the variable, *after* the string we have to put a `%` symbol followed by the variable, in this case `x`.

If we want to place more than one variable, we just add more `%s` place-holders and after the external `%` we place the required variables in round parenthesis, separating them with commas:

[88]:
```python
x = 3
y = 5
"I jumped %s times and did %s sprints" % (x,y)
```

[88]:
```
'I jumped 3 times and did 5 sprints'
```

We can put as many variables as we want, also non-numerical ones:

[89]:
```python
x = 3
y = 5
prize = 'Best Athlet in Town'
"I jumped %s times, did %s sprints and won the prize '%s'" % (x,y,prize)
```

[89]:
```
"I jumped 3 times, did 5 sprints and won the prize 'Best Athlet in Town'"
```

Exercise - supercars

You've got some money, so you decide to buy two models of supercars. Since you already know accidents are on the way, for each model you will buy as many cars as there are characters in each model name.

Write some code which stores in the string `s` the number of cars you will buy.

Example - given:

```python
car1 = 'Jaguar'
car2 = 'Ferrari'
```

After your code, it should show:

```
>>> s
'I will buy 6 Jaguar and 7 Ferrari supercars'
```

- **USE** `%s` placeholders

[186] https://realpython.com/python-formatted-output/
[187] https://realpython.com/python-formatted-output/#the-python-formatted-string-literal-f-string

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[90]:
```
car1, car2 = 'Jaguar','Ferrari'         # I will buy 6 Jaguar and 7 Ferrari supercars
#car1, car2 = 'Porsche','Lamborghini'   # I will buy 7 Porsche and 11 Lamborghini␣
↪supercars

# write here

s = 'I will buy %s %s and %s %s supercars' % (len(car1), car1, len(car2), car2)
print(s)
```

```
I will buy 6 Jaguar and 7 Ferrari supercars
```

</div>

[90]:
```
car1, car2 = 'Jaguar','Ferrari'         # I will buy 6 Jaguar and 7 Ferrari supercars
#car1, car2 = 'Porsche','Lamborghini'   # I will buy 7 Porsche and 11 Lamborghini␣
↪supercars

# write here
```

```
I will buy 6 Jaguar and 7 Ferrari supercars
```

5.3.15 References

- Think Python, Chapter 8, Strings[188]
- Think Python, Chapter 9, Word play[189]
- Some extra for people wanting to do *text mining:*, have a look at NLTK library[190]

5.3.16 Continue

Go on reading notebook Strings 2 - operators[191]

[ ]:
```
```

5.4 Strings 2 - operators

5.4.1 Download exercises zip

Browse files online[192]

Python offers several operators to work with strings:

[188] http://greenteapress.com/thinkpython2/html/thinkpython2009.html
[189] http://greenteapress.com/thinkpython2/html/thinkpython2010.html
[190] https://www.nltk.org/
[191] https://en.softpython.org/strings/strings2-sol.html
[192] https://github.com/DavidLeoni/softpython-en/tree/master/strings

Operator	Use	Result	Meaning
len	`len`(str)	int	Returns the length of the string
concatenation	str + str	str	Concatenate two strings
inclusion _	str `in` str	bool	Checks whether a string is contained inside another one
indexing	str`[`int`]`	str	Reads the character at the specified index
slice	str`[`int`:`int`]`	str	Extracts a sub-string
equality	`==`,`!=`	`bool`	Checks whether strings are equal or different
comparisons	`<`,`<=`,`>`, `>=`	`bool`	Performs lexicographic comparison
ord	`ord`(str)	`int`	Returns the order of a character
chr	`chr`(int)	`str`	Given an order, returns the corresponding character
replication	str `*` int	str	Replicate the string

5.4.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
strings
    strings1.ipynb
    strings1-sol.ipynb
    strings2.ipynb
    strings2-sol.ipynb
    strings3.ipynb
    strings3-sol.ipynb
    strings4.ipynb
    strings4-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `strings2.ipynb`

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.4.3 Reading characters

A string is a sequence of characters, and often we might want to access a single character by specifying the position of the character we are interested in.

It's important to remember that the position of characters in strings start from `0`. For reading a character in a certain position, we need to write the string followed by square parenthesis and spcify the position inside. Examples:

[2]:
```
'park'[0]
```

[2]:
```
'p'
```

[3]:
```
'park'[1]
```

[3]:
```
'a'
```

[4]:
```
#0123
'park'[2]
```

[4]:
```
'r'
```

[5]:
```
#0123
'park'[3]
```

[5]:
```
'k'
```

If we try to go beyond the last character, we will get an error:

```
#0123
'park'[4]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-106-b8f1f689f0c7> in <module>
      1 #0123
----> 2 'park'[4]

IndexError: string index out of range
```

Before we used a string by specifying it as a literal, but we can also use variables:

[6]:
```
    #01234
x = 'cloud'
```

[7]:
```
x[0]
```

[7]:
```
'c'
```

[8]:
```
x[2]
```

[8]:
```
'o'
```

How is represented the character we've just read? If you noticed, it is between quotes like if it were a string. Let's check:

[9]:
```
type(x[0])
```

[9]:
```
str
```

It's really a string. To somebody this might come as a surprise, also from a philosophical standpoint: Python strings are made of… strings! Other programming languages may use a specific type for the single character, but Python uses strings to be able to better manage complex alphabets as, for example, japanese.

QUESTION: Let's suppose `x` is *any* string. If we try to execute this code:

```
x[0]
```

we will get:

1. always a character
2. always an error
3. sometimes a character, sometimes an error according to the string

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: we might obtain an error with the empty string (try it)

</div>

QUESTION: Let's suppose `x` is an empty string. If we try to execute this code:

```
x[len(x)]
```

we will get:

1. always a character
2. always an error
3. sometimes a character, sometimes an error according to the string at hand

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: since indexing starts from 0, `len` always gives us a number which is the biggest usable index plus one.

</div>

Exercise - alternate

Given two strings both of length `3`, print a string which alternates characters from both strings. You code must work with any string of this length

Example - given:

```
x="say"
y="hi!"
```

it should print:

```
shaiy!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:
```
# write here

x="say"
y="hi!"
print(x[0] + y[0] + x[1] + y[1] + x[2] + y[2])
```

```
shaiy!
```

</div>

```
[10]: # write here
```

```
shaiy!
```

Negative indexes

In Python we can also use negative indexes, which instead to start from *the beginning* they start *from the end:*

```
[11]: #4321
"park"[-1]
```

```
[11]: 'k'
```

```
[12]: #4321
"park"[-2]
```

```
[12]: 'r'
```

```
[13]: #4321
"park"[-3]
```

```
[13]: 'a'
```

```
[14]: #4321
"park"[-4]
```

```
[14]: 'p'
```

If we go one step beyond, we get an error:

```
#4321
"park"[-5]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-126-668d8a13a324> in <module>
----> 1 "park"[-5]

IndexError: string index out of range
```

QUESTION: Suppose `x` is a NON-empty string. What do we get with the following expression?

```
x[-len(x)]
```

1. always a character
2. always an error
3. sometimes a character, sometime an error according to the string

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1. (we supposed the string is never empty)

</div>

QUESTION: Suppose `x` is a some string (possibly empty), the expressions

```
x[len(x) - 1]
```

and

```
x[-len(x)]
```

are equivalent ? What do they do ?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: the expressions are equivalent: if the string is empty both produce an error, if it is full both give the last character

</div>

QUESTION: If `x` is a non-empty string, what does the following expression produce? Can we simplify it to a shorter one?

```
(x + x)[len(x)]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: it's the same as `x[0]`

</div>

QUESTION: If `x` is a non-empty string, what does the following expression produce? An error? Something else? Can we simplify it?

```
'park'[0][0]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: We know that `'park'[0]` produces a character, but we also know that in Python characters extracted from strings are also strings of length 1. So, if after the expression `"park"[0]` which produces the string `'p'` we add another `[0]` it's like we were writing `'p'[0]`, which returns the zeroth character found in the string in the string `'p'`, that is `'p'` itself.

</div>

QUESTION: If `x` is a non-empty string, what does the following expression produce? An error? Something else? Can we simplify it?

```
(x[0])[0]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `x[0]` is an expression which produces the first character of the string `x`. In Python, we can place expressions among parenthesis whenever we want. So in this case the parenthesis don't produce any effect, and the expression becomes equivalent to `x[0][0]` which as we've seen before it's the same as writing `x[0]`

</div>

5.4.4 Substitute characters

We said strings in Python are immutable. Suppose we have a string like this:

[15]:
```
    #01234
x = 'port'
```

and, for example, we want to change the character at position `2` (in this case, the `r`) into an `s`. What do we do?

We might be tempted to write like the following, but Python would punish us with an error:

```
x[2] = 's'

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-113-e5847c6fa4bf> in <module>
----> 1 x[2] = 's'

TypeError: 'str' object does not support item assignment
```

The correct solution is assigning a completely new string to `x`, obtained by taking pieces from the previous one:

[16]:
```
x = x[0] + x[1] + 's' + x[3]
```

[17]:
```
x
```

[17]:
```
'post'
```

If seeing `x` to the right of equal sign baffles you, we can decompose the code like this and it will work the same way:

[18]:
```
x = "port"
y = x
x = y[0] + y[1] + 's' + y[3]
```

Try it in Python Tutor:

[19]:
```
x = "port"
y = x
x = y[0] + y[1] + 's' + y[3]

jupman.pytut()
```

[19]:
```
<IPython.core.display.HTML object>
```

5.4.5 Slices

We might want to read only a subsequence which starts from a position and ends up in another one. For example, suppose we have:

[20]:
```
    #0123456789
x = 'mercantile'
```

and we want to extract the string `'canti'`, which starts at index 3 **included**. We might extract the single characters and concatenate them with `+` sign, but we would write a lot of code. A better option is to use the so-called slices[193]: simply write the string followed by square parenthesis containing only start index (**included**), a colon, and finally end index (**excluded**):

[193] http://greenteapress.com/thinkpython2/html/thinkpython2009.html#sec95

```
[21]:     #0123456789
      x = 'mercantile'

      x[3:8]    # note the : inside start and end indexes
```

```
[21]: 'canti'
```

WARNING: Extracting with slices DOES NOT modify the original string !!

Let's see an example:

```
[22]:     #0123456789
      x = 'mercantile'

      print('               x is', x)
      print('The slice x[3:8] is', x[3:8])
      print('               x is', x)       # note x continues to point to old string!
```

```
               x is mercantile
The slice x[3:8] is canti
               x is mercantile
```

QUESTION: if `x` is any string of length at least `5`, what does this code produce? An error? It works? Can we shorten it?

```
x[3:4]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: If the string has length at least `5`, can might have a situation like this:

```
    #01234
x = 'abcde'
```

The slice `x[3:4]` will extract from position `3` **included** until position 4 **excluded**, so as a matter of fact it will extract only one character from position 3. So the code is equivalent to `x[3]`

</div>

Exercise - garalampog

Write some code to extract and print `alam` from the string `"garalampog"`. Try guessing the correct indexes.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[23]: x = "garalampog"

      # write here

      #      0123456789
      print(x[3:7])
```

```
alam
```

</div>

```
[23]: x = "garalampog"

# write here
```

```
alam
```

Exercise - ifEweEfav lkSD lkWe

Write some code to extract and print `kD` from the string `"ifE\te\nfav  lkD  lkWe"`. Be careful of spaces and special characters (before you might want to print `x`). Try guessing correct indexes.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[24]: x = "ifE\te\nfav  lkD lkWe"

# write here

#     0123 45 67890123456789
#x = "ifE\te\nfav  lkD lkWe"

print(x[12:14])
```

```
kD
```

</div>

```
[24]: x = "ifE\te\nfav  lkD lkWe"

# write here
```

```
kD
```

Slices - limits

Whenever we use slice we must be careful with index limits. Let's see how they behave:

```
[25]: #012345
"chair"[0:3]   # from index 0 *included* to 3 *excluded*
```

```
[25]: 'cha'
```

```
[26]: #012345
"chair"[0:4]   # from index 0 *included* to 4 *excluded*
```

```
[26]: 'chai'
```

```
[27]: #012345
"chair"[0:5]   # from index 0 *included* to 5 *excluded*
```

```
[27]: 'chair'
```

[28]:
```python
#012345
"sedia"[0:6]    # if we go beyond string length Python doesn't complain
```

[28]:
```
'sedia'
```

QUESTION: if x is any string (also empty), what does this expression do? Can it give an error? Does it return something useful?

```python
x[0:len(x)]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: It always returns a NEW copy of the whole string, because it starts from index `0` *included* and ends at index `len(x)` *excluded*. It also works with the empty string, as `''[0:len('')]` is equivalent to `''[0:0]` that is a substring from `0` *included* to `0` *excluded*, so we don't take any character and we do not go beyond string limits. Actually, even if we went beyond, we wouldn't upset Python (try writing `''[0:100]`

</div>

Slice - Omitting limits

If we want, it's possible to omit the starting index, in this case Python will suppose it's a `0`:

[29]:
```python
#0123456789
"catamaran"[:3]
```

[29]:
```
'cat'
```

It's also possible to omit the ending index, in that case Python will extract until the end of the string:

[30]:
```python
#0123456789
"catamaran"[3:]
```

[30]:
```
'amaran'
```

By omitting both indexes we obtain the full string:

[31]:
```python
"catamaran"[:]
```

[31]:
```
'catamaran'
```

Exercise - ysterymyster

Write some code that given a string `x` prints the string composed with all the characters of `x` except the first one, followed by all characters of `x` except the last one.

- your code must work with any string

Example 1 - given:

```python
x = "mystery"
```

must print:

```
ysterymyster
```

Example 2 - given:

```
x = "talking"
```

must print:

```
alkingtalkin
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[32]:
```
x = "mystery"
#x = "talking"

# write here

print(x[1:] + x[0:len(x)-1])
```

```
ysterymyster
```

</div>

[32]:
```
x = "mystery"
#x = "talking"

# write here
```

```
ysterymyster
```

Slice - negative limits

If we want, it's also possible to set negative limits, although it's not always intuitive:

[33]:
```
#0123456

"vegetal"[3:0]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

[33]:
```
''
```

[34]:
```
#0123456
"vegetal"[3:1]    # from index 3 to positive indexes <= 3 doesn't produce anything
```

[34]:
```
''
```

[35]:
```
#0123456
"vegetal"[3:2]   # from index 3 to positive indexes <= 3 doesn't produce anything
```

[35]:
```
''
```

[36]:
```
#0123456
"vegetal"[3:3]   # from index 3 to positive indexes <= 3 doesn't produce anything
```

[36]:
```
''
```

Let's see what happens with negative indexes:

[37]:
```
#0123456   positive indexes
#7654321   negative indexes
"vegetal"[3:-1]
```

[37]:
```
'eta'
```

[38]:
```
#0123456   positive indexes
#7654321   negative indexes
"vegetal"[3:-2]
```

[38]:
```
'et'
```

[39]:
```
#0123456   positive indexes
#7654321   negative indexes
"vegetal"[3:-3]
```

[39]:
```
'e'
```

[40]:
```
#0123456   positive indexes
#7654321   negative indexes
"vegetal"[3:-4]
```

[40]:
```
''
```

[41]:
```
#0123456   positive indexes
#7654321   negative indexes
"vegetal"[3:-5]
```

[41]:
```
''
```

Exercise - javarnanda

Given a string `x`, write some code to extract and print its last 3 characters joined to the to first 3.

- Your code should work for any string of length equal or greater than 3

Example 1 - given:

```
x = "javarnanda"
```

it should print:

```
javnda
```

Example 2 - given:

```
x = "abcd"
```

it should print:

```
abcbcd
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[42]:
```
x = "javarnanda"
#x = "abcd"

# write here

print(x[:3] + x[-3:])
```

```
javnda
```

</div>

[42]:
```
x = "javarnanda"
#x = "abcd"

# write here
```

```
javnda
```

Slice - modifying

Suppose to have the string

[43]:
```
    #0123456789
s = "the table is placed in the center of the room"
```

and we want to change `s` assignment so it becomes associated to the string:

```
#0123456789
"the chair is placed in the center of the room"
```

Since both strings are similar, we might be tempted to only redefine the character sequence which corresponds to the word `"table"`, which goes from index `4` included to index `9` excluded:

```
s[4:9] = "chair"   # WARNING! WRONG!

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-57-0de7363c6882> in <module>
----> 1 s[4:9] = "chair"   # WARNING! WRONG!

TypeError: 'str' object does not support item assignment
```

Sadly, we would receive an error, because as repeated many times strings are IMMUTABLE, so we cannot select a chunk of a particular string and try to change the original string. What we can do instead is to build a NEW string from pieces of the original string, concatenates the desired characters and associates the result to the variabile of which we want to modify the assignment:

[44]:
```
    #0123456789
s = "the table is placed in the center of the room"
s = s[0:4] + "chair" + s[9:]
print(s)
```

```
the chair is placed in the center of the room
```

When Python finds the line

```
s = s[0:4] + "chair" + s[9:]
```

FIRST it calculates the result on the right of the =, and THEN associates the result to the variable on the left. In the expression on the right only NEW strings are generated, which once built can be assigned to variable `s`

Exercise - the run

Write some code such that when given the string `s`

```
s = 'The Gold Rush has begun.'
```

and some variables

```
what = 'Atom'
happened = 'is over'
```

substitues the substring `'Gold'` with the string in the variable `what` and substitues the substring `'has begun'` with the string in the variable `happened`.

After exectuing your code, the string associated to `s` should be

```
>>> print(s)
"The Atom Rush is over."
```

- **DON'T** use constant characters in your code, i.e. dots `'.'` aren't allowed !

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[45]:
```
    #01234567890123456789012345678
s = 'The Gold Rush has begun.'
what = 'Atom'
happened = 'is over'

# write here

s = s[0:4] + what + s[8:14] + happened + s[23:]
print(s)
```

```
The Atom Rush is over.
```

</div>

[45]:
```
    #01234567890123456789012345678
s = 'The Gold Rush has begun.'
what = 'Atom'
happened = 'is over'

# write here

```

```
The Atom Rush is over.
```

5.4.6 `in` operator

To check if a string is contained in another one, we use the the `in` operator.

Note the result of this expression is a boolean:

[46]:
```
'the' in 'Singing in the rain'
```

[46]:
```
True
```

[47]:
```
'si' in 'Singing in the rain'   # in operator is case-sensitive
```

[47]:
```
False
```

[48]:
```
'Si' in 'Singing in the rain'
```

[48]:
```
True
```

Exercise - contained 1

You are given two strings `x` and `y`, and a third `z`. Write some code which prints `True` if `x` and `y` are both contained in `z`.

Example 1 - given:

```
x = 'cad'
y = 'ra'
z = 'abracadabra'
```

it should print:

```
True
```

Example 2 - given:

```
x = 'zam'
y = 'ra'
z = 'abracadabra'
```

it should print:

```
False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[49]:
```
x,y,z = 'cad','ra','abracadabra'    # True
#x,y,z = 'zam','ra','abracadabra'   # False

# write here

print((x in z) and (y in z))
```

```
True
```

</div>

[49]:
```
x,y,z = 'cad','ra','abracadabra'   # True
#x,y,z = 'zam','ra','abracadabra'  # False

# write here
```

```
True
```

Exercise - contained 2

Given three strings `x`, `y`, `z`, write some code which prints `True` if the string `x` is contained in at least one of the strings `y` or `z`, otherwise prints `False`

- your code should work with any set of strings

Example 1 - given:

```
x = "ope"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should print:

```
True
```

Example 2 - given:

```
x = "nope"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should print:

```
False
```

Example 3 - given:

```
x = "cle"
y = "honesty makes for long friendships"
z = "I hope it's clear enough"
```

it should show:

```
True
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```
x,y,z = "ope","honesty makes for long friendships","I hope it's clear enough"   # True
#x,y,z = "nope","honesty makes for long friendships","I hope it's clear enough"  #
↪False
#x,y,z = "cle","honesty makes for long friendships","I hope it's clear enough"  # True

# write here

print((x in y) or (x  in z))
```

```
True
```

</div>

```
[50]: x,y,z = "ope","honesty makes for long friendships","I hope it's clear enough"   # True
#x,y,z = "nope","honesty makes for long friendships","I hope it's clear enough"  #␣
↪False
#x,y,z = "cle","honesty makes for long friendships","I hope it's clear enough"  # True

# write here
```

```
True
```

5.4.7 Comparisons

Python offers us the possibility to perform a *lexicographic comparison* among strings, like we would when placing names in an address book. Although sorting names is something intuitive we often do, we must be careful about special cases.

First, let's determine when two strings are equal.

Equality operators

To check whether two strings are equal, you can use te operator == which as result produces the boolean `True` or `False`

WARNING: == is written with TWO equal signs !!!

```
[51]: "dog" == "dog"
```

```
[51]: True
```

```
[52]: "dog" == "wolf"
```

```
[52]: False
```

Equality operator is case-sensitive:

```
[53]: "dog" == "DOG"
```

```
[53]: False
```

To check whether two strings are NOT equal, we can use the operator !=, which we can expect to behave exactly as the opposite of ==:

```
[54]: "dog" != "dog"
```

```
[54]: False
```

```
[55]: "dog" != "wolf"
```

```
[55]: True
```

```
[56]: "dog" != "DOG"
```

```
[56]: True
```

As an alternative, we might use the operator `not`:

[57]:
```python
not "dog" == "dog"
```

[57]:
```
False
```

[58]:
```python
not "wolf" == "dog"
```

[58]:
```
True
```

[59]:
```python
not "dog" == "DOG"
```

[59]:
```
True
```

QUESTION: what does the following code print?

```python
x = "river" == "river"
print(x)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: When Python encounters `x = "river" == "river"` it sees an assignment, and associates the result of the expression `"river" == "river"` to the variable `x`. So FIRST it calculates the expression `"river" == "river"` which produces the boolean `True`, and THEN associates the value `True` to the variable `x`. Finally `True` is printed.

</div>

QUESTION: for each of the following expressions, try to guess whether it produces `True` or `False`

1. `'hat' != 'Hat'`
2. `'hat' == 'HAT'`
3. `'choralism'[2:5] == 'contemporary'[7:10]`
4. `'AlAbAmA'[4:] == 'aLaBaMa'`
5. `'bright'[9:20] == 'dark'[10:15]`
6. `'optical'[-1] == 'crystal'[-1]`
7. `('hat' != 'jacket') == ('trousers' != 'bow')`
8. `('stra' in 'stradivarius') == ('div' in 'digital divide')`
9. `len('note') in '5436'`
10. `str(len('note') in '5436'`
11. `len('posters') in '5436'`
12. `str(len('posters')) in '5436'`

Exercise - statist

Write some code which prints `True` if a `word` begins with the same two characters it ends with.

- Your code should work for any `word`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[60]:
```python
word = 'statist'   # True
#word = 'baobab'   # False
#word = 'maxima'   # True
#word = 'karma'    # False

# write here

print(word[:2] == word[-2:len(word)])
```

```
True
```

</div>

[60]:
```python
word = 'statist'   # True
#word = 'baobab'   # False
#word = 'maxima'   # True
#word = 'karma'    # False

# write here
```

```
True
```

Comparing characters

Characters have an inherent order we can exploit. Let's see an example:

[61]:
```python
'a' < 'g'
```

[61]:
```
True
```

another one:

[62]:
```python
'm' > 'c'
```

[62]:
```
True
```

They sound reasonable comparisons! But what about this (notice capital `'Z'`)?

[63]:
```python
'a' < 'Z'
```

[63]:
```
False
```

Maybe this doesn't look so obvious. And what if we get creative and compare with symbols such as square bracket or Unicode[194] hearts ??

[194] https://en.softpython.org/strings/strings1-sol.html#Unicode-characters

[64]:
```python
'a' > '♥'
```

[64]:
```
False
```

To determine how to deal with this special cases, we must remember ASCII[195] assignes a position number to each character, defining as a matter of fact *an ordering* between all its characters.

If we want to know the corresponding number of a character, we can use the function `ord`:

[65]:
```python
ord('a')
```

[65]:
```
97
```

[66]:
```python
ord('b')
```

[66]:
```
98
```

[67]:
```python
ord('z')
```

[67]:
```
122
```

If we want to go the other way, given a position number we can obtain the corresponding character with `chr` function:

[68]:
```python
chr(97)
```

[68]:
```
'a'
```

Uppercase characters have different positions:

[69]:
```python
ord('A')
```

[69]:
```
65
```

[70]:
```python
ord('Z')
```

[70]:
```
90
```

EXERCISE: Using the functions above, try to find which characters are *between* capital `Z` and lowercase `a`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[1]:
```python
# write here
#print(chr(91),chr(92), chr(93),chr(94), chr(95),chr(96))
```

</div>

[1]:
```python
# write here
```

The ordering allows us to perform *lexicographic comparisons* between single characters:

[72]:
```python
'a' < 'b'
```

[72]:
```
True
```

[195] https://en.softpython.org/strings/strings1-sol.html#ASCII-characters

```
[73]: 'g' >= 'm'
```

```
[73]: False
```

EXERCISE: Write some code that:

1. prints the `ord` values of `'A'`, `'Z'` and a given `char`
2. prints `True` if `char` is uppercase, and `False` otherwise

- Would your code also work with accented capitalized characters such as `'Á'`?
- **NOTE**: the possibile character sets are way too many, so the proper solution would be to use the method isupper[196] we will see in the next tutorial.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[15]: char = 'G'   # True
      #char = 'g'  # False
      #char = 'Á'  # True ??
      # write here
      print('A:', ord('A'), ' Z:', ord('Z'))
      print(char+':',ord(char))
      ord(c) >= ord('A') and ord(c) <= ord('Z')   #  only checks simple English alphabet␣
      ↪cases
```

```
A: 65  Z: 90
G: 71
```

```
[15]: False
```

</div>

```
[15]: char = 'G'   # True
      #char = 'g'  # False
      #char = 'Á'  # True ??
      # write here
```

```
A: 65  Z: 90
G: 71
```

```
[15]: False
```

Also, since Unicode character set *includes* ASCII, the ordering of ASCII characters can be used to safely compare them against unicode characters, so comparing characters or their `ord` should be always equivalent:

```
[74]: ord('a')   # ascii
```

```
[74]: 97
```

```
[75]: ord('♥')   # unicode
```

```
[75]: 9829
```

```
[76]: 'a' > '♥'
```

[196] https://en.softpython.org/strings/strings3-sol.html#isupper-and-islower-methods

[76]:
```
False
```

[77]:
```
ord('a') > ord('♥')
```

[77]:
```
False
```

Python also offers lexicographic comparisons on strings with more than one character. To understand what the expected result should be, we must distinguish among several cases, though:

- strings of equal / different length
- strings with same / mixed case

Let's begin with same length strings:

[78]:
```
'mario' > 'luigi'
```

[78]:
```
True
```

[79]:
```
'mario' > 'wario'
```

[79]:
```
False
```

[80]:
```
'Mario' > 'Wario'
```

[80]:
```
False
```

[81]:
```
'mario' > 'Wario'   # capital case is *before* lowercase in ASCII
```

[81]:
```
True
```

Comparing different lengths

Short strings which are included in longer ones come first in the ordering:

[82]:
```
'troll' < 'trolley'
```

[82]:
```
True
```

If they only share a prefix with a longer string, Python compares characters after the common prefix, in this case it detects that `s` is greater than corresponding `e`:

[83]:
```
'trolls' < 'trolley'
```

[83]:
```
False
```

Exercise - Character intervals

You are given a couple of strings `i1` and `i2` of two characters each.

We suppose they represent character intervals: the first character of an interval always has order number lower or equal than the second.

There are five possibilities: either the first interval 'is contained in', or 'contains', or 'overlaps', or 'is before' or 'is after' the second interval. Write some code which tells which containment relation we have.

Example 1 - given:

```
i1 = 'gm'
i2 = 'cp'
```

Your program should print:

```
gm is contained in cp
```

To see why, you can look at this little representation (you **don't** need to print this!):

```
  c   g     m  p
abcdefghijklmnopqrstuvwxyz
```

Example 2 - given:

```
i1 = 'mr'
i2 = 'pt'
```

Your program should print:

```
mr overlaps pt
```

because `mr` is not contained nor contains nor completely precedes nor completely follows `pt` (you **don't** need to print this!):

```
            m  p r t
abcdefghijklmnopqrstuvwxyz
```

- if `i1` interval coincides with `i2`, it is consideraded as containing `i2`
- **DO NOT** use cycles nor `if`
- **HINT**: to satisfy above constraint, think about booleans evaluation order[197], for example the expression

```
'g' >= 'c' and 'm' <= 'p' and 'is contained in'
```

produces as result the string `'is contained in'`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[84]:
```
i1,i2 = 'gm','cp'   # gm is contained in cp
#i1,i2 = 'dh','dh'  # gm is contained in cp  #(special case)
#i1,i2 = 'bw','dq'  # bw contains dq
#i1,i2 = 'ac','bd'  # ac overlaps bd
#i1,i2 = 'mr','pt'  # mr overlaps pt
#i1,i2 = 'fm','su'  # fm is before su
#i1,i2 = 'xz','pq'  # xz is after pq


# write here
res = (i1[0] >= i2[0] and i1[1] <= i2[1] and 'is contained in')  \
      or (i1[0] < i2[0] and i1[1] >i2[1] and 'contains')          \
      or (i1[0] >= i2[0] and i1[0] <= i2[1] and 'overlaps')       \
      or (i1[1] >= i2[0] and i1[1] <= i2[1] and 'overlaps')       \
      or (i1[1] < i2[0] and 'is before')                          \
      or (i1[0] > i2[1] and 'is after')
```

(continues on next page)

[197] https://en.softpython.org/basics/basics-sol.html#Booleans---Evaluation-order

(continued from previous page)

```
#print(i1, res, i2)
```

</div>

[84]:
```
i1,i2 = 'gm','cp'   # gm is contained in cp
#i1,i2 = 'dh','dh'  # gm is contained in cp  #(special case)
#i1,i2 = 'bw','dq'  # bw contains dq
#i1,i2 = 'ac','bd'  # ac overlaps bd
#i1,i2 = 'mr','pt'  # mr overlaps pt
#i1,i2 = 'fm','su'  # fm is before su
#i1,i2 = 'xz','pq'  # xz is after pq


# write here
```

5.4.8 Replication operator

With the operator `*` you can replicate a string n times, for example:

[85]:
```
'beer' * 4
```

[85]:
```
'beerbeerbeerbeer'
```

Note a NEW string is created, without tarnishing the original:

[86]:
```
drink = "beer"
```

[87]:
```
print(drink * 4)
```

```
beerbeerbeerbeer
```

[88]:
```
drink
```

[88]:
```
'beer'
```

Exercise - za za za

Given a `syllable` and a `phrase` which terminates with a character `n` as a digit, write some code which prints a string with the `syllable` repeated `n` times, separated by spaces.

- Your code must work with any string assigned to `syllable` and `phrase`

Example - given:

```
phrase = 'the number 7'
syllable = 'za'
```

after you code, ti should print:

```
za za za za za za za
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[89]:
```python
phrase = 'the number 7'
syllable = 'za'          # za za za za za za za
#phrase = 'Give me 5'    # za za za za za


# write here

print((syllable +' ') * (int(phrase[-1])))
```

```
za za za za za za za
```

</div>

[89]:
```python
phrase = 'the number 7'
syllable = 'za'          # za za za za za za za
#phrase = 'Give me 5'    # za za za za za


# write here

```

```
za za za za za za za
```

5.4.9 Continue

Go on reading notebook Strings 3 - methods[198]

[ ]:
```python

```

5.5 Strings 3 - methods

5.5.1 Download exercises zip

Browse files online[199]

Every data type has associated particular methods for that type, let's see those associated to type string (`str`)

> **WARNING: ALL string methods ALWAYS generate a NEW string**
>
> The original string object is NEVER changed (because strings are immutable).

[198] https://en.softpython.org/strings/strings3-sol.html
[199] https://github.com/DavidLeoni/softpython-en/tree/master/strings

Result	Method	Meaning
str	*str.upper()*	Return the string with all characters uppercase
str	*str.lower()*	Return the string with all characters lowercase
str	*str.capitalize()*	Return the string with the first uppercase character
str	*str.strip(str)*	Remove strings from the sides
str	*str.lstrip(str)*	Remove strings from left side
str	*str.rstrip(str)*	Remove strings from right side
str	*str.replace(str, str)*	Substitute substrings
bool	*str.startswith(str)*	Check if the string begins with another one
bool	*str.endswith(str)*	Check whether the string ends with another one
int	*str.find(str)*	Return the first position of a substring starting from the left
int	*str.rfind(str)*	Return the first position of a substring starting from the right
int	*str.count(str)*	Count the number of occurrences of a substring
bool	*str.isalpha(str)*	Check if all characters are alhpabetic
bool	*str.isdigit(str)*	Check if all characters are digits
bool	*str.isupper*	Check if all characters are uppercase
bool	*str.islower*	Check if all characters are lowercase

Note: the list is not exhaustive, here we report only the ones we use in the book. For the full list see Python documentation[200]

5.5.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
strings
    strings1.ipynb
    strings1-sol.ipynb
    strings2.ipynb
    strings2-sol.ipynb
    strings3.ipynb
    strings3-sol.ipynb
    strings4.ipynb
    strings4-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `strings3.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[200] https://docs.python.org/3/library/stdtypes.html#string-methods

5.5.3 Example - `upper`

A method is a function of an object that takes as input the object to which is it is applied and does some calculation.

The type of the string (`str`) has predefined methods like `str.upper()` which can be applied to other string objects (i.e.: `'hello'` is a string object)

The method `str.upper()` takes the string to which it is applied, and creates a NEW string in which all the characters are in uppercase. To apply a method like `str.upper()` to the particular string object `'hello'`, we must write:

```
'hello'.upper()
```

Frst we write the object on which apply the method (`'hello'`), then a dot `.`, and afterwards the method name followed by round parenthesis. The brackets can also contain further parameters according to the method.

Examples:

[2]:
```
'hello'.upper()
```

[2]:
```
'HELLO'
```

[3]:
```
"I'm important".upper()
```

[3]:
```
"I'M IMPORTANT"
```

WARNING: like ALL string methods, the original string object on which the method is called does NOT get modified.

Example:

[4]:
```
x = "hello"
y = x.upper()    # generates a NEW string and associates it to the variables y
```

[5]:
```
x                # x variable is still associated to the old string
```

[5]:
```
'hello'
```

[6]:
```
y                #  y variable is associated to the new string
```

[6]:
```
'HELLO'
```

Have a look now at the same example in Python Tutor:

[7]:
```
x = "hello"
y = x.upper()
print(x)
print(y)

jupman.pytut()
```

```
hello
HELLO
```

[7]:
```
<IPython.core.display.HTML object>
```

5.5.4 Exercise - walking

Write some code which given a string `x` (i.e.: `x='walking'`) prints twice the row:

```
walking WALKING walking WALKING
walking WALKING walking WALKING
```

- **DO NOT** create new variables
- your code must work with any string

[8]:
```python
x = 'walking'

print(x, x.upper(), x, x.upper())
print(x, x.upper(), x, x.upper())
```

```
walking WALKING walking WALKING
walking WALKING walking WALKING
```

Help: If you are not sure about a method (for example, `strip`), you can ask Python for help this way:

WARNING: when using help, DON'T put parenthesis after the method name !!

[9]:
```python
help("hello".strip)
```

```
Help on built-in function strip:

strip(…) method of builtins.str instance
    S.strip([chars]) -> str

    Return a copy of the string S with leading and trailing
    whitespace removed.
    If chars is given and not None, remove characters in chars instead.
```

5.5.5 `lower` method

Return the string with all lowercase characters

[10]:
```python
my_string = "HEllo WorLd"

another_string = my_string.lower()

print(another_string)
```

```
hello world
```

[11]:
```python
print(my_string)   # didn't change
```

```
HEllo WorLd
```

Exercise - lowermid

Write some code that given any string `x` of odd length, prints a new string like `x` having the mid-character as lowercase.

- your code must work with any string !
- **HINT**: to calculate the position of the mid-character, use integer division with the operator `//`

Example 1 - given:

```
x = 'ADORATION'
```

it should print:

```
ADORaTION
```

Example 2 - given:

```
x = 'LEADINg'
```

it should print:

```
LEAdINg
```

[12]:

```
    #012345678
x = 'ADORATION'
#x = 'LEADINg'
k = len(x) // 2

print(x[:k] + x[k].lower() + x[k+1:])
```

```
ADORaTION
```

5.5.6 `capitalize` method

`capitalize()` creates a NEW string having only the FIRST character as uppercase:

[13]:

```
"artisan".capitalize()
```

[13]:

```
'Artisan'
```

[14]:

```
"premium".capitalize()
```

[14]:

```
'Premium'
```

[15]:

```
x = 'goat'
y = 'goat'.capitalize()
```

[16]:

```
x      #  x remains associate to the old value
```

[16]:

```
'goat'
```

[17]:

```
y      #  y is associated to the new string
```

[17]:

```
'Goat'
```

Exercise - Your Excellence

Write some code which given two strings `x` and `y` returns the two strings concatenated, separating them with a space and both as lowercase except the first two characters which must be uppercase

Example 1 - given:

```
x = 'yoUR'
y = 'exCelLenCE'
```

it must print:

```
Your Excellence
```

Example 2 - given:

```
x = 'hEr'
y = 'maJEsty'
```

it must print:

```
Her Majesty
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```
x,y = 'yoUR','exCelLenCE'
#x,y = 'hEr','maJEsty'

# write here

print(x.capitalize() + " " + y.capitalize())
```

```
Your Excellence
```

</div>

[18]:
```
x,y = 'yoUR','exCelLenCE'
#x,y = 'hEr','maJEsty'

# write here


```

```
Your Excellence
```

5.5.7 `strip` method

Eliminates white spaces, tabs and linefeeds from the *sides* of the string. In general, this set of characters is called *blanks.*

NOTE: it does NOT removes *blanks* inside string words! It only looks on the sides.

[19]:
```
x = ' \t\n\n\t carpe diem \t  '    # we put white space, tab and line feeds at the␣
↪sides
```

[20]:
```
x
```

```
[20]: ' \t\n\n\t carpe diem \t  '
```

```
[21]: print(x)
```

```


         carpe diem
```

```
[22]: len(x)   # remember that special characters like \t and \n occupy 1 character
```

```
[22]: 20
```

```
[23]: y = x.strip()
```

```
[24]: y
```

```
[24]: 'carpe diem'
```

```
[25]: print(y)
```

```
carpe diem
```

```
[26]: len(y)
```

```
[26]: 10
```

```
[27]: x      # IMPORTANT: x is still associated to the old string !
```

```
[27]: ' \t\n\n\t carpe diem \t  '
```

Specificying character to strip

If you only want Python to remove some specific character, you can specify them in parenthesis. Let's try to specify only one:

```
[28]: 'salsa'.strip('s')    #  not internal `s` is not stripped
```

```
[28]: 'alsa'
```

If we specify two or more, Python removes all the characters it can find from the sides

Note the order in which you specify the characters does **not** matter:

```
[29]: 'caustic'.strip('aci')
```

```
[29]: 'ust'
```

WARNING: If you specify characters, Python doesn't try anymore to remove blanks!

```
[30]: 'bouquet  '.strip('b')    # it won't strip right spaces !
```

```
[30]: 'ouquet  '
```

```
[31]: '\tbouquet  '.strip('b')     # ... nor strip left blanks such as tab
```

```
[31]: '\tbouquet  '
```

According to the same principle, if you specify a space `' '`, then Python will **only** remove spaces and won't look for other blanks!!

```
[32]: '  careful! \t'.strip(' ')    # strips only on the left!
```

```
[32]: 'careful! \t'
```

QUESTION: for each of the following expressions, try to guess which result it produces (or if it gives an error):

1. `'\ttumultuous\n'.strip()`
2. `' a b c '.strip()`
3. `'\ta\tb\t'.strip()`
4. `'\\tMmm'.strip()`
5. `'sky diving'.strip('sky')`
6. `'anacondas'.strip('sad')`
7. `'\nno way '.strip(' ')`
8. `'\nno way '.strip('\\n')`
9. `'\nno way '.strip('\n')`
10. `'salsa'.strip('as')`
11. `'\t ACE '.strip('\t')`
12. `' so what? '.strip("")`
13. `str(-3+1).strip("+"+"-")`

5.5.8 `lstrip` method

Eliminates white spaces, tab and line feeds from *left side* of the string.

NOTE: does NOT remove *blanks* between words of the string! Only those on left side.

```
[33]: x = '\n \t the street \t '
```

```
[34]: x
```

```
[34]: '\n \t the street \t '
```

```
[35]: len(x)
```
```
[35]: 17
```
```
[36]: y = x.lstrip()
```
```
[37]: y
```
```
[37]: 'the street \t '
```
```
[38]: len(y)
```
```
[38]: 13
```
```
[39]: x       # IMPORTANT: x is still associated to the old string !
```
```
[39]: '\n \t the street \t '
```

5.5.9 `rstrip` method

Eliminates white spaces, tab and line feeds from *left side* of the string.

NOTE: does NOT remove *blanks* between words of the string! Only those on right side.

```
[40]: x = '\n \t the lighthouse \t '
```
```
[41]: x
```
```
[41]: '\n \t the lighthouse \t '
```
```
[42]: len(x)
```
```
[42]: 21
```
```
[43]: y = x.rstrip()
```
```
[44]: y
```
```
[44]: '\n \t the lighthouse'
```
```
[45]: len(y)
```
```
[45]: 18
```
```
[46]: x       # IMPORTANT: x è is still associated to the old string !
```
```
[46]: '\n \t the lighthouse \t '
```

Exercise - hatespace

Given a string `x` which may contain some *blanks* (spaces, control characters like `\t` and `\n`, ...) from begin to end, write some code which prints the string without *blanks* and the strings `'START'` and `'END'` at the sides

Example - given:

```
x = ' \t  \n \n hatespace\n   \t \n'
```

prints

```
STARThatespaceEND
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[47]: # write here

x = ' \t  \n \n hatespace\n   \t \n'

print('START' + x.strip() + 'END')
```

```
STARThatespaceEND
```

</div>

```
[47]: # write here
```

```
STARThatespaceEND
```

Exercise - Bad to the bone

You have an uppercase string `s` which contains at the sides some stuff you want to remove: `punctuation` , a lowercase `char` and some blanks. Write some code to perform the removal

Example - given:

```
char = 'b'
punctuation = '!?.;,'
s = ' \t\n...bbbbbBAD TO THE BONE\n!'
```

your code should show:

```
'BAD TO THE BONE'
```

- use only `strip` (or `lstrip` and `rstrip`) methods (if necessary, you can do repeated calls)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[48]: char = 'b'
punctuation = '!?.;,'
s = ' \t\n...bbbbbBAD TO THE BONE\n!'

# write here
s.strip().strip(char + punctuation).strip()
```

```
[48]: 'BAD TO THE BONE'
```

</div>

```
[48]: char = 'b'
      punctuation = '!?.;,'
      s = ' \t\n...bbbbbBAD TO THE BONE\n!'

      # write here
```

```
[48]: 'BAD TO THE BONE'
```

5.5.10 `replace` method

`str.replace` takes two strings and looks in the string on which the method is called for occurrences of the first string parameter, which are substituted with the second parameter. Note it gives back a NEW string with all substitutions performed.

Example:

```
[49]: "the train runs off the tracks".replace('tra', 'ra')
```

```
[49]: 'the rain runs off the racks'
```

```
[50]: "little beetle".replace('tle', '')
```

```
[50]: 'lit bee'
```

```
[51]: "talking and joking".replace('ING', 'ed')  # it's case sensitive
```

```
[51]: 'talking and joking'
```

```
[52]: "TALKING AND JOKING".replace('ING', 'ED')  # here they are
```

```
[52]: 'TALKED AND JOKED'
```

As always with strings, `replace` DOES NOT modify the string on which it is called:

```
[53]: x = "On the bench"
```

```
[54]: y = x.replace('bench', 'bench the goat is alive')
```

```
[55]: y
```

```
[55]: 'On the bench the goat is alive'
```

```
[56]: x  # IMPORTANT: x is still associated to the old string !
```

```
[56]: 'On the bench'
```

If you give an optional third argument `count`, only the first `count` occurrences will be replaced:

```
[57]: "TALKING AND JOKING AND LAUGHING".replace('ING', 'ED', 2)  # replaces only first 2␣
      ↪occurrences
```

[57]:
```
'TALKED AND JOKED AND LAUGHING'
```

QUESTION: for each of the following expressions, try to guess which result it produces (or if it gives an error)

1. ```
   '$£ciao£$'.replace('£','').replace('$','')
   ```

2. ```
   '$£ciao£$'.strip('£').strip('$')
   ```

Exercise - substitute

Given a string `x`, write some code to print a string like `x` but with all occurrences of `bab` substituted by `dada`

Example - given:

```
x = 'kljsfsdbabòkkrbabej'
```

it should print

```
kljsfsddadaòkkrdadaej
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[58]:
```
# write here

x = 'kljsfsdbabòkkrbabej'
print(x.replace('bab', 'dada'))
```

```
kljsfsddadaòkkrdadaej
```

</div>

[58]:
```
# write here
```

```
kljsfsddadaòkkrdadaej
```

5.5.11 `startswith` method

`str.startswith` takes as parameter a string and returns `True` if the string before the dot begins with the string passed as parameter. Example:

[59]:
```
"the dog is barking in the road".startswith('the dog')
```

[59]:
```
True
```

[60]:
```
"the dog is barking in the road".startswith('is barking')
```

[60]:
```
False
```

[61]:
```
"the dog is barking in the road".startswith('THE DOG')   # uppercase is different from␣
↪lowercase
```

[61]:
```
False
```

[62]:
```
"THE DOG BARKS IN THE ROAD".startswith('THE DOG')      # uppercase is different from␣
↪lowercase
```

[62]:
```
True
```

Exercise - by Jove

Write some code which given any three strings `x`, `y` and `z`, prints `True` if both `x` and `y` start with string `z`, otherwise prints `False`

Example 1 - given:

```
x = 'by Jove'
y = 'by Zeus'
z = 'by'
```

it should print:

```
True
```

Example 2 - given:

```
x = 'by Jove'
y = 'by Zeus'
z = 'from'
```

it should print:

```
False
```

Example 3 - given:

```
x = 'from Jove'
y = 'by Zeus'
z = 'by'
```

it should print:

```
False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[63]:
```
x,y,z = 'by Jove','by Zeus','by'     # True
#x,y,z = 'by Jove','by Zeus','from'  # False
#x,y,z = 'from Jove','by Zeus','by'  # False

# write here

print(x.startswith(z) and y.startswith(z))
```

```
True
```

</div>

[63]:
```
x,y,z = 'by Jove','by Zeus','by'      # True
#x,y,z = 'by Jove','by Zeus','from'   # False
#x,y,z = 'from Jove','by Zeus','by'   # False

# write here
```

```
True
```

5.5.12 `endswith` method

`str.endswith` takes as parameter a string and returns `True` if the string before the dot ends with the string passed as parameter. Example:

[64]:
```
"My best wishes".endswith('st wishes')
```

[64]:
```
True
```

[65]:
```
"My best wishes".endswith('best')
```

[65]:
```
False
```

[66]:
```
"My best wishes".endswith('WISHES')     # uppercase is different from lowercase
```

[66]:
```
False
```

[67]:
```
"MY BEST WISHES".endswith('WISHES')     # uppercase is different from lowercase
```

[67]:
```
True
```

Exercise - Snobbonis

Given couple names `husband` and `wife`, write some code which prints `True` if they share the surname, `False` otherwise.

- assume the surname is always at position `9`
- your code must work for any couple `husband` and `wife`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[68]:
```
                 #0123456789                #0123456789
husband, wife  = 'Antonio  Snobbonis',      'Carolina Snobbonis'    # True
#husband, wife = 'Camillo  De Spaparanzi',  'Matilda  Degli Agi'    # False

# write here

print(wife.endswith(husband[9:]))
```

```
True
```

</div>

[68]:
```
                   #0123456789                 #0123456789
husband, wife  = 'Antonio  Snobbonis',      'Carolina Snobbonis'    # True
#husband, wife = 'Camillo  De Spaparanzi', 'Matilda  Degli Agi'    # False

# write here
```

```
True
```

5.5.13 `count` method

The method `count` takes a substring and counts how many occurrences are there in the string before the dot.

[69]:
```
"astral stars".count('a')
```

[69]:
```
3
```

[70]:
```
"astral stars".count('A')     # it's case sensitive
```

[70]:
```
0
```

[71]:
```
"astral stars".count('st')
```

[71]:
```
2
```

Optionally, you can pass a two other parameters to indicate an index to start counting from (included) and where to end (excluded):

[72]:
```
#012345678901
"astral stars".count('a',4)
```

[72]:
```
2
```

[73]:
```
#012345678901
"astral stars".count('a',4,9)
```

[73]:
```
1
```

Exercise - astro money

During 2020 lockdown while looking at the stars above you started feeling… waves. After some thinking, you decided *THEY* wanted to communicate with you so you you set up a dish antenna on your roof to receive messages from aliens. After months of apparent irrelevant noise, one day you finally receive a message you think you can translate. Aliens are *obviously* trying to tell you the winning numbers of lottery!

A message is a sequence of exactly 3 *different* character repetitions, the number of characters in each repetition is a number you will try at the lottery. You frantically start developing the translator to show these lucky numbers on the terminal.

Example - given:

```
s = '$$$$€€€€€!!'
```

it should print:

```
$ € !
4 5 2
```

- **IMPORTANT: you can assume all sequences have *different* characters**
- **DO NOT** use cycles nor comprehensions
- for simplicity assume each character sequence has at most 9 repetitions

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[74]:
```python
    #01234567890      # $ € !
s = '$$$$€€€€€!!'     # 4 5 2
                      # a b c
#s = 'aaabbbbbbccc'   # 3 6 3
                      # H A L
#s = 'HAL'            # 1 1 1

# write here
p1 = 0
d1 = s.count(s[p1])
p2 = p1 + d1
d2 = s.count(s[p2])
p3 = p2 + d2
d3 = s.count(s[p3])

print(s[p1],s[p2],s[p3])
print(d1,d2,d3)
```

```
$ € !
4 5 2
```

</div>

[74]:
```python
    #01234567890      # $ € !
s = '$$$$€€€€€!!'     # 4 5 2
                      # a b c
#s = 'aaabbbbbbccc'   # 3 6 3
                      # H A L
#s = 'HAL'            # 1 1 1

# write here
```

```
$ € !
4 5 2
```

5.5.14 `find` method

`find` returns the index of the *first* occurrence of some given substring:

[75]:
```
#0123456789012345
'bingo bongo bong'.find('ong')
```

[75]:
```
7
```

If no occurrence is found, it returns `-1`:

[76]:
```
#0123456789012345
'bingo bongo bong'.find('bang')
```

[76]:
```
-1
```

[77]:
```
#0123456789012345
'bingo bongo bong'.find('Bong')    #  case-sensitive
```

[77]:
```
-1
```

Optionally, you can specify an index from where to start searching (included):

[78]:
```
#0123456789012345
'bingo bongo bong'.find('ong',10)
```

[78]:
```
13
```

And also where to end (excluded):

[79]:
```
#0123456789012345
'bingo bongo bong'.find('g',4, 9)
```

[79]:
```
-1
```

Exercise - bananas

While exploring a remote tropical region, an ethologist discovers a population of monkeys which appear to have some concept of numbers. They collect bananas in the hundreds which are then traded with coconuts collected by another group. To comunicate the quantities of up to 999 bananas, they use a series of exactly three guttural sounds. The ethologist writes down the sequencies and formulates the following theory: each sound is comprised by a sequence of the same character, repeated a number of times. The number of characters in the first sequence is the first digit (the hundreds), the number of characters in the second sequence is the second digit (the decines), while the last sequence represents units.

Write some code which puts in variable `bananas` **an integer** representing the number.

For example - given:

```
s = 'bb bbbbb aaaa'
```

your code should print:

```
>>> bananas
254
>>> type(bananas)
int
```

- **IMPORTANT 1: different sequences may use the *same* character!**

- **IMPORTANT 2: you cannot assume which characters monkeys will use**: you just know each digit is represented by a repetition of the same character
- **DO NOT** use cycles nor comprehensions
- the monkeys have no concept of zero

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[80]:
```python
    #0123456789012
s = 'bb bbbbb aaaa'     # 254
#s = 'ccc cc ccc'       # 323
#s = 'vvv rrrr ww'      # 342
#s = 'cccc h jjj'       # 413
#s = '□□□ □□□□□□ □□□□'  # 364  (you could get *any* weird character, also unicode ...)

# write here
p1 = s.find(' ')
bananas = len(s[:p1])*100
p2 = s.find(' ',p1+1)
bananas += len(s[p1+1:p2])*10
bananas += len(s[p2+1:])*1

#print('The bananas are',bananas)
#type(bananas)
```

</div>

[80]:
```python
    #0123456789012
s = 'bb bbbbb aaaa'     # 254
#s = 'ccc cc ccc'       # 323
#s = 'vvv rrrr ww'      # 342
#s = 'cccc h jjj'       # 413
#s = '□□□ □□□□□□ □□□□'  # 364  (you could get *any* weird character, also unicode ...)

# write here
```

5.5.15 `rfind` method

Like *find method*, but search starts from the right.

5.5.16 `isalpha` method

The method `isalpha` returns `True` if all characters in the string are alphabetic:

[81]:
```python
'CoralReel'.isalpha()
```

[81]:
```
True
```

Numbers are not considered alphabetic:

[82]:
```python
'Route 666'.isalpha()
```

[82]:
```
False
```

Also, blanks are *not* alphabetic:

[83]:
```
'Coral Reel'.isalpha()
```

[83]:
```
False
```

… nor punctuation:

[84]:
```
'!'.isalpha()
```

[84]:
```
False
```

… nor weird Unicode stuff:

[85]:
```
'♥'.isalpha()
```

[85]:
```
False
```

[86]:
```
''.isalpha()
```

[86]:
```
False
```

5.5.17 `isdigit` method

`isdigit` method returns `True` if a string is only composed of digits:

[87]:
```
'391'.isdigit()
```

[87]:
```
True
```

[88]:
```
'400m'.isdigit()
```

[88]:
```
False
```

Floating point and scientific notations are not recognized:

[89]:
```
'3.14'.isdigit()
```

[89]:
```
False
```

[90]:
```
'4e29'.isdigit()
```

[90]:
```
False
```

5.5.18 `isupper` and `islower` methods

We can check wheter a character is uppercase or lowercase with `isupper` and `islower` methods:

[91]:
```
'q'.isupper()
```

[91]:
```
False
```

[92]:
```
'Q'.isupper()
```

[92]:
```
True
```

[93]:
```
'b'.islower()
```

[93]:
```
True
```

[94]:
```
'B'.islower()
```

[94]:
```
False
```

They also work on longer strings, checking if all characters meet the criteria:

[95]:
```
'GREAT'.isupper()
```

[95]:
```
True
```

[96]:
```
'NotSoGREAT'.isupper()
```

[96]:
```
False
```

Note blanks and punctuation are not taken into account:

[97]:
```
'REALLY\nGREAT !'.isupper()
```

[97]:
```
True
```

We could check whether a character is upper/lower case by looking at ASCII code but the best way we covers all alphabets is by using `isupper` and `islower` methods, for example they also work with accented letters:

[98]:
```
'à'.isupper()
```

[98]:
```
False
```

[99]:
```
'Á'.isupper()
```

[99]:
```
True
```

5.5.19 Other exercises

QUESTION: For each following expression, try to find the result

1. `'gUrP'.lower() == 'GuRp'.lower()`
2. `'NaNo'.lower() != 'nAnO'.upper()`
3. `'O' + 'ortaggio'.replace('o','\t \n     ').strip() + 'O'`
4. `'DaDo'.replace('D','b') in 'barbados'`

5.5.20 Continue

Go on reading notebook Strings 4 - other exercises[201]

```
[ ]:
```

5.6 Lists 1 - Introduction

5.6.1 Download exercises zip

Browse files online[202]

A Python list is a **mutable** sequence of heterogeneous elements, in which we can put the objects we want. The order in which we put them is preserved.

5.6.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
lists
    lists1.ipynb
    lists1-sol.ipynb
    lists2.ipynb
    lists2-sol.ipynb
    lists3.ipynb
    lists3-sol.ipynb
    lists4.ipynb
    lists4-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `lists1.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[201] https://en.softpython.org/strings/strings4-sol.html
[202] https://github.com/DavidLeoni/softpython-en/tree/master/lists

5.6.3 Creating lists

We can create a list by specifying the elements it contains between square brackets, separating them with a comma.

For example, in this list we insert the numbers `7`, `4` e `9`:

```
[2]: [7,4,9]
```

```
[2]: [7, 4, 9]
```

Like all Python objects, we can associate them to a variable, in this case we create a new one we call `my_list`:

```
[3]: my_list = [7,4,9]
```

```
[4]: my_list
```

```
[4]: [7, 4, 9]
```

Let's see what happens in memory, and compare strings representation with lists representation:

```
[5]: # WARNING: before using the function jupman.pytut() which follows,
     #          it is necessary to first execute this cell with Shift+Enter
     #          it's sufficient to execute it only once


     import jupman
```

```
[6]: my_string = "prova"

     my_list = [7,4,9]

     jupman.pytut()
```

```
[6]: <IPython.core.display.HTML object>
```

We suddenly note a relevant difference. The string remained in the azure region where associations among variables and values usually stay. From variable `my_list` we see instead an arrow departing to a new yellow memory region, which is created as soon the execution reaches the row where the list is defined.

Later we will analyze more in detail the consequences of this.

In a list the same elements may appear many times:

```
[7]: numbers = [1,2,3,1,3]
```

```
[8]: numbers
```

```
[8]: [1, 2, 3, 1, 3]
```

We can put any element, for example strings:

```
[9]: fruits = ["apple", "pear", "peach", "strawberry", "cherry"]
```

```
[10]: fruits
```

```
[10]: ['apple', 'pear', 'peach', 'strawberry', 'cherry']
```

We can also mix the object types contained in a list, for example we can have integers and strings:

[11]:
```
mix = ["table", 4 ,"chair", 8, 5, 1, "chair"]
```

In Python Tutor it will be shown like this:

[12]:
```
mix = ["table", 5 , 4, "chair", 8, "chair"]

jupman.pytut()
```

[12]:
```
<IPython.core.display.HTML object>
```

For convenience we can also write the list on many rows (the spaces in this case do not count, only remember to terminate rows with commas `,`)

[13]:
```
mix = ["table",
       5 ,
       4,
       "chair",
       8,
       "chair"]
```

EXERCISE: try writing the list above WITHOUT putting a comma after the `5`, which error appears?

[14]:
```
# write here
```

A list can also contain other lists:

[15]:
```
table = [  ['a','b','c'],  ['d','e','f']  ]
```

Typically, whenver we have structures like this, it's convenient to displace them on many rows (it's not mandatory but improves clarity):

[16]:
```
table = [                # start external big list
          ['a','b','c'],    # internal list 1
          ['d','e','f']     # internal list 2
        ]                # end external big list
```

[17]:
```
table
```

[17]:
```
[['a', 'b', 'c'], ['d', 'e', 'f']]
```

Let's see how it is shown in Python Tutor:

[18]:
```
table = [
          ['a','b','c'],
          ['d','e','f']
        ]

jupman.pytut()
```

[18]:
```
<IPython.core.display.HTML object>
```

As we previously said, in a list we can put the elements we want, so we can mix lists with different dimensions, strings, numbers and so on:

[19]:
```
so_much = [
            ['hello',3,'world'],
            'a string',
            [9,5,6,7,3,4],
            8,
          ]
```

[20]:
```
print(so_much)
```

```
[['hello', 3, 'world'], 'a string', [9, 5, 6, 7, 3, 4], 8]
```

Let's see how it appears in Python Tutor:

[21]:
```
so_much = [
             ['hello',3,'world'],
             'a string',
             [9,5,6,7,3,4],
             8,
          ]

jupman.pytut()
```

[21]:
```
<IPython.core.display.HTML object>
```

Empty list

There are two ways to create an empty list.

1) with square brackets:

[22]:
```
my_empty_list = []
```

[23]:
```
my_empty_list
```

[23]:
```
[]
```

2) Or with `list()`:

[24]:
```
another_empty_list = list()
```

[25]:
```
another_empty_list
```

[25]:
```
[]
```

WARNING: When you create an empty list (independently from the used notation), a NEW region in memory is allocated to place the list.

Let's see what this means with Python Tutor:

[26]:
```
a = []
b = []

jupman.pytut()
```

[26]:
```
<IPython.core.display.HTML object>
```

Note two arrows appeared, which point to **different** memory regions. The same would have happend by initializing the lists with some elements:

[27]:
```
la = [8,6,7]
lb = [9,5,6,4]

jupman.pytut()
```

[27]:
```
<IPython.core.display.HTML object>
```

We would have two lists in different memory regions also by placing identical elements inside the lists:

[28]:
```
la = [8,6,7]
lb = [8,6,7]

jupman.pytut()
```

[28]:
```
<IPython.core.display.HTML object>
```

Things get complicated when we start using assignment operations:

[29]:
```
la = [8,6,7]
```

[30]:
```
lb = [9,5,6,4]
```

[31]:
```
lb = la
```

By writing `lb = la`, we told Python to 'forget' the previous assignment of `lb` to `[9,5,6,4]`, and instead to associate `lb` to the same value associated to `la`, that is `[8,6,7]`. Thus, in memory we will see an arrow departing from `lb` and arriving into `[8,6,7]`, and the memory region where the list `[9,5,6,4]` was placed will be removed (won't be associated to any variable anymore). Let's see what happens with Python Tutor:

[32]:
```
la = [8,6,7]
lb = [9,5,6,4]
lb = la

jupman.pytut()
```

[32]:
```
<IPython.core.display.HTML object>
```

EXERCISE: Try swapping the lists associated to variables `la` and `lb` by using only assignments and **without creating new lists**. If you want, you can overwrite a third variable `lc`. Verify what happes with Python Tutor.

- your code must work for any value of `la`, `lb` and `lc`

Example - given:

```
la = [9,6,1]
lb = [2,3,4,3,5]
lc = None
```

After your code, it must result:

```
>>> print(la)
[2,3,4,3,5]
>>> print(lb)
[9,6,1]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[33]:
```
la = [9,6,1]
lb = [2,3,4,3,5]
lc = None

# write here

lc = la
la = lb
lb = lc

#print(la)
#print(lb)
```

</div>

[33]:
```
la = [9,6,1]
lb = [2,3,4,3,5]
lc = None

# write here
```

QUESTION: Have a look at these two pieces of code. For each case, try thinking how they might be represented in memory and then verify with Python Tutor.

- could there be a difference?
- how many memory cells will be allocated in total?
- how many arrows will you see?

```
# first case
lb = [
       [8,6,7],
       [8,6,7],
       [8,6,7],
       [8,6,7],
     ]
```

```
# second case
la = [8,6,7]
lb = [
       la,
       la,
       la,
       la
     ]
```

[34]:
```
# first case
lb = [
       [8,6,7],
       [8,6,7],
       [8,6,7],
       [8,6,7],
     ]
jupman.pytut()
```

[34]:
```
<IPython.core.display.HTML object>
```

[35]:
```
# second case

la = [8,6,7]
lb = [
       la,
       la,
       la,
       la
     ]
jupman.pytut()
```

[35]:
```
<IPython.core.display.HTML object>
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: In the first case, we have a 'big list' associated to the variable `lb` which contains `4` sublists each of 3 elements. Each sublist is created as new, so in total in memory we end up with 4 cells of the big list lb + (4 sublists * 3 cells each) = 16 cells

In the second case we have instead always the 'big list' associated to the variable lb of 4 cells, but inside it contains some pointers to the same identical list la. So the total number of occupied cells is 4 cells of big list lb + (1 sublist * 3 cells) = 7 cells

</div>

Exercise - create lists 1

Given two variables:

```
la = [4,3]
lb = [9,6,7]
```

Write some code which prints the list `[[4, 3], [[9, 6, 7], [4, 3], [9, 6, 7]], [4, 3]]`

- **DO NOT write numbers**, use only lists of variables

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[36]:
```
la = [4,3]
lb = [9,6,7]

# write here
print([
        la,
        [lb,la,lb],
        la
])
```

```
[[4, 3], [[9, 6, 7], [4, 3], [9, 6, 7]], [4, 3]]
```

</div>

[36]:
```
la = [4,3]
lb = [9,6,7]

# write here
```

```
[[4, 3], [[9, 6, 7], [4, 3], [9, 6, 7]], [4, 3]]
```

Exercise - create lists 2

Insert some values in the lists `la`, `lb` such that

```
print([[la,la],[lb,la]])
```

prints

```
[[[8, 4], [8, 4]], [[4, 8, 4], [8, 4]]]
```

- **Insert only NUMBERS**
- Observe in Python Tutor how arrows are represented

[37]:
```
la = []  # insert numbers
lb = []  # insert numbers

print([[la,la],[lb,la]])
```

```
[[[], []], [[], []]]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[38]:
```
# SOLUTION

la = [8,4]
lb = [4,8,4]

#print([[la,la],[lb,la]])
```

</div>

[38]:
```
```

Exercise - create lists 3

Insert some values as elements of the lists `la`, `lb` e `lc` such that

```
print([[lb,lb,[lc,la]],lc])
```

prints

```
[[[8, [7, 7]], [8, [7, 7]], [[8, 7], [8, 5]]], [8, 7]]
```

- **insert only NUMBERS or NEW LISTS OF NUMBERS**

- Observe in Python Tutor are arrows are represented

[39]:
```
la = []  # insert elements  (numbers or lists of numbers)
lb = []  # insert elements  (numbers or lists of numbers)
lc = []  # insert elements  (numbers or lists of numbers)

print([[lb,lb,[lc,la]],lc])
```

```
[[[], [], [[], []]], []]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[40]:
```
# SOLUTION

la = [8,5]
lb = [8,[7,7]]
lc = [8,7]

#print([[lb,lb,[lc,la]],lc])
```

</div>

[40]:
```

```

Exercise - create lists 4

Insert some values in the lists `la`, `lb` such that

```
print([[la,lc,la], lb])
```

prints

```
[[[3, 2], [[3, 2], [8, [3, 2]]], [3, 2]], [8, [3, 2]]]
```

- **insert only NUMBERS or VARIABLES** `la`,`lb` or `lc`
- Observe in Python Tutor how arrows are represented

[41]:
```
la = []  # insert numbers or variables la, lb, lc
lb = []  # insert numbers or variables la, lb, lc
lc = []  # insert numbers or variables la, lb, lc

print([[la,lc,la], lb])
```

```
[[[], [], []], []]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[42]:
```
# SOLUTION
la = [3,2]
lb = [8,la]
lc = [la,lb]

#print([[la,lc,la], lb])
```

</div>

[42]:

```
```

5.6.4 Convert sequences into lists

`list` may also be used to convert any sequence into a NEW list. A sequence type we've alredy seen are strings, so we can check what happens when we use `list` like it were a function and we pass a string as parameter:

[43]:

```python
list("train")
```

[43]:

```
['t', 'r', 'a', 'i', 'n']
```

We obtained a list in which each element is made of a character from the original string.

What happens if we call instead `list` on another list?

[44]:

```python
list( [7,9,5,6] )
```

[44]:

```
[7, 9, 5, 6]
```

Apparently, nothing particular, we obtained a list with the same start elements. But is it really the same list? Let's have a better look with Python Tutor:

[45]:

```python
la = [7,9,5,6]

lb = list( la )

jupman.pytut()
```

[45]:

```
<IPython.core.display.HTML object>
```

We note a NEW memory region was created with the same elements of `la`.

Exercise - gulp

Given a string with mixed uppercase and lowercase characters, write some code which creates a list containing as first element a list with characters from the string lowercased and as second element a list containing all the uppercased characters

- your code must work with any string
- if you don't remember the string methods, look here[203]

Example - given:

```python
s = 'GuLp'
```

your code must print:

```
[['g', 'u', 'l', 'p'], ['G', 'U', 'L', 'P']]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[203] https://en.softpython.org/strings/strings3-sol.html

[46]:
```
s = 'GuLp'

# write here
print([list(s.lower()), list(s.upper())])
```

```
[['g', 'u', 'l', 'p'], ['G', 'U', 'L', 'P']]
```

</div>

[46]:
```
s = 'GuLp'

# write here

```

```
[['g', 'u', 'l', 'p'], ['G', 'U', 'L', 'P']]
```

QUESTION: This code:

- produces an error or assigns something to `x` ?
- After its execution, how many lists remain in memory?
- Can we shorten it?

```
s = "marathon"
x = list(list(list(list(s))))
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The code assigns the list `['m', 'a', 'r', 'a', 't', 'h', 'o', 'n']` to variable `x`. The first time `list(s)` generates a NEW list `['m', 'a', 'r', 'a', 't', 'h', 'o', 'n']`. Successive calls to `list` take as input the just generated list and keep creating NEW lists with the same identical content. Since no produced list except the last one is assigned to a variable, the intermediate ones are eliminated at the end of execution. We can thus safely shorten the code by writing:

```
s = "marathon"
x = list(s)
```

</div>

QUESTION: This code:

- produces an error or assigns something to `x` ?
- After its execution, how many lists remain in memory?

```
s = "chain"
a = list(s)
b = list(a)
c = b
x = list(c)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Only 3 lists remain in memory, each containing 6 cells. This time the lists persist in memory because they are associated to variables `a`, `b` and `c`. We have 3 and not 4 lists because in instruction `c = b` the `c` variable is associated to the same identical memory region associated as variable `b`

</div>

Exercise - garaga

Given

```
sa = "ga"
sb = "ra"
la = ['ga']
lb = list(la)
```

- Assign to `lc` a list built in such a way so that once printed produces:

```
>>> print(lc)
```

```
[['g', 'a', 'r', 'a'], ['ga'], ['ga'], ['r', 'a', 'g', 'a']]
```

- **in Python Tutor, ALL the arrows must point to a different memory region**

[47]:
```
sa = "ga"
sb = "ra"
la = ['ga']
lb = list(la)

# insert come code in the list
lc = []

print(lc)
jupman.pytut()
```

```
[]
```

[47]:
```
<IPython.core.display.HTML object>
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[48]:
```
# SOLUTION

sa = "ga"
sb = "ra"
la = ['ga']
lb = list(la)
lc = [list(sa + sb), list(la), list(lb), list(sb + sa) ]

print(lc)
jupman.pytut()
```

```
[['g', 'a', 'r', 'a'], ['ga'], ['ga'], ['r', 'a', 'g', 'a']]
```

[48]:
```
<IPython.core.display.HTML object>
```

</div>

[48]:
```

```

```
[['g', 'a', 'r', 'a'], ['ga'], ['ga'], ['r', 'a', 'g', 'a']]
```

[48]:
```
<IPython.core.display.HTML object>
```

5.6.5 References

- Think Python, Chapter 10, Lists[204] Exercises 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7

5.6.6 Continue

Go on reading notebook Lists 2 - operators[205]

```
[ ]:
```

5.7 Lists 2 - operators

5.7.1 Download exercises zip

Browse online files[206]

There are several operators to manipulate lists. The following ones behave like the ones we've seen in strings:

Operator	Result	Meaning
`len`(lst)	`int`	Return the list length
list[int]	obj	Reads/writes an element at the specified index
list[int:int]	`list`	Extracts a sublist - return a NEW list
obj `in` list	`bool`	Cheks if the element is contained in the list
list + list	`list`	Concatenates two lists - return a NEW list
`max`(lst)	`int`	Given a list of numbers, return the greatest one
`min`(lst)	`int`	Given a list of numbers, returns the smallest one
`sum`(lst)	`int`	Given a list of numbers, sums all of them
list `*` int	`list`	Replicates the list - return a NEW list
`==`,`!=`	`bool`	Cheks whether lists are equal of different

5.7.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
lists
    lists1.ipynb
    lists1-sol.ipynb
    lists2.ipynb
    lists2-sol.ipynb
    lists3.ipynb
    lists3-sol.ipynb
    lists4.ipynb
    lists4-sol.ipynb
    jupman.py
```

204 http://greenteapress.com/thinkpython2/html/thinkpython2011.html
205 https://en.softpython.org/lists/lists2-sol.html
206 https://github.com/DavidLeoni/softpython-en/tree/master/lists

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `lists2.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.7.3 Length of a list

A list is a sequence, and like any sequence you can use the function `len` to obtain the length:

```
[2]: a = [7,5,8]
```

```
[3]: len(a)
```

```
[3]: 3
```

```
[4]: b = [8,3,6,4,7]
```

```
[5]: len(b)
```

```
[5]: 5
```

If a list contains other lists, they count as single elements:

```
[6]: mixed = [
                [4,5,1],
                [8,6],
                [7,6,0,8],
            ]
```

```
[7]: len(mixed)
```

```
[7]: 3
```

WARNING: YOU CAN'T use `len` as a method

For example, this DOESN'T work: `[3,4,2].len()`

EXERCISE: Try writing `[3,4,2].len()` here, which error appears?

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[8]:
```
# write here
#[3,4,2].len()
```

</div>

[8]:
```
# write here
```

EXERCISE: Try writing `[3,4,2].len` WITHOUT the round parenthesis at the end, which error appears?

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[9]:
```
# write here
#[3,4,2].len
```

</div>

[9]:
```
# write here
```

QUESTION: If `x` is some list, by writing:

`len(len(x))`

what do we get?

1. the length of the list
2. an error
3. something else

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: `len` wants as argument a *sequence* and gives back a *number,* so the internal call to `len(x)` produces a number which is given to the external `len` and at that point Python will complain it received a number instead of a sequence. Verify which error appears by writing `len(len(x))` down here.

</div>

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:
```
# write here
#len(len(x))
```

</div>

[10]:
```
# write here
```

QUESTION: Look at this expression, without executing it. What does it produce?

```
[len([]), len([len(['a','b'])])]
```

1. an error (which one?)
2. a number (which one?)
3. a list (which one?)

Try writing the result by hand, and then compare it with the one obtained by executing the code in a cell.

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: the list `[0, 1]`

</div>

QUESTION: Look at this expression, without executing it. What does it produce?

```
len([[[],[]],[],[[[]]],[[],[]]])
```

1. an error (which one?)
2. a number (which one?)
3. a list (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2. produces the number `4`

</div>

QUESTION: What does the following expression produce?

```
[[((len('ababb')))],len(["argg",('b'),("c")]), len([len("bc")])]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `[[5], 3, 1]`

</div>

5.7.4 Reading an element

Like for strings, we can access an element a list element by putting the index of the position we want to access among square brackets:

[11]:
```
    # 0   1   2   3
la = [77, 69, 95, 57]
```

As for any sequence, the positions start from `0`:

[12]:
```
la[0]
```

[12]:
```
77
```

[13]:
```
la[1]
```

[13]:
```
69
```

[14]:
```
la[2]
```

[14]:
```
95
```

[15]:
```
la[3]
```

[15]:
```
57
```

Like for any string, if we exaggerate with the index we get an error:

```
la[4]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-134-09bfed834fa2> in <module>
----> 1 la[4]

IndexError: list index out of range
```

As in strings, we can obtain last element by using a negative index:

[16]:
```
    # 0   1   2   3
la = [77, 69, 95, 57]
```

[17]:
```
la[-1]
```

[17]:
```
57
```

[18]:
```
la[-2]
```

[18]:
```
95
```

[19]:
```
la[-3]
```

[19]:
```
69
```

[20]:
```
la[-4]
```

[20]:
```
77
```

If we go beyond the list length, we get an error:

```
la[-5]

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-169-f77280923dce> in <module>
----> 1 la[-5]

IndexError: list index out of range
```

QUESTION: if `x` is some list, by writing:

```
x[0]
```

what do we get?

1. the first element of the list
2. always an error
3. sometimes an element, sometimes an error according to the list

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: if the list is empty Python will not find the element and will give us an error. Which one? Try writing in the cell down here `[][0]` and see what happens.

</div>

```
[ ]:
```

QUESTION: if `x` is some list, by writing:

```
x[len(x)]
```

what do we get?

1. an element of the list
2. always an error
3. sometimes an element, sometimes an error according to the list

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2. always an error: `len(x)` will always be a number equal to the last available index + 1

</div>

5.7.5 Writing an element

Since all the lists are MUTABLE, given a list object we can change the content of any cell inside.

For example, suppose you want to change the cell at index `2` of the list `la`, from `6` to `5`:

```
[21]:      #0  1  2  3
      la = [7, 9, 6, 8]
```

We might write like this:

```
[22]: la[2] = 5
```

```
[23]: la
```

```
[23]: [7, 9, 5, 8]
```

Let's see what's happening with Python Tutor:

[24]:
```
# WARNING: FOR PYTHON TUTOR TO WORT, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#           (it's sufficient to execute it only once)


import jupman
```

[25]:
```
#      0  1  2  3
la = [7, 9, 6, 8]
la[2] = 5

jupman.pytut()
```

[25]:
```
<IPython.core.display.HTML object>
```

As you see, no new memory regions are created, it just overwrites an existing cell.

5.7.6 Mutating shared lists

WARNING: 90% OF PROGRAMMING ERRORS ARE CAUSED BY MISUNDERSTANDING THIS TOPIC !!!

READ VERY WELL !!!

What happens when we associate the same identical mutable object to two variables, like for example a list, and then we mutate the object using one of the two variables?

Let's look at an example - first, we associate the list `[7,9,6]` to variable `la`:

[26]:
```
la = [7,9,6]
```

Now we define a new variable `lb`, and we associate the *same value* that was already associated to variable `la`. Note: we are NOT creating new lists !

[27]:
```
lb = la
```

[28]:
```
print(la)   # la is always the same
```

```
[7, 9, 6]
```

[29]:
```
print(lb)   # lb is the *same* list associated to la
```

```
[7, 9, 6]
```

We can now try modifying a cell of `lb`, putting `5` in the cell at index `0`:

[30]:
```
lb[0] = 5
```

If we try printing the variables `la` and `lb`, Python will look at the values associated to each variable. Since the value is the same identical list (which is in the same identical memory region), in both cases you will see the change we just did !

[31]:
```
print(la)
```

```
[5, 9, 6]
```

[32]:
```
print(lb)
```

```
[5, 9, 6]
```

Let's see in detail what happens with Python Tutor:

[33]:
```
la = [7,9,6]
lb = la
lb[0] = 5
print('la è', la)
print('lb è', lb)

jupman.pytut()
```

```
la è [5, 9, 6]
lb è [5, 9, 6]
```

[33]:
```
<IPython.core.display.HTML object>
```

Let's see the difference when we explicitly create a list equal to `la`.

In this case we will have two distinct memory regions and `la` will NOT be modified:

[34]:
```
la = [7,9,6]
lb = [7,9,6]
lb[0] = 5
print('la is', la)
print('lb is', lb)

jupman.pytut()
```

```
la is [7, 9, 6]
lb is [5, 9, 6]
```

[34]:
```
<IPython.core.display.HTML object>
```

QUESTION: After executing this code, what will be printed? How many lists will be present in memory?

Try drawing **ON PAPER** what is supposed to happen in memory, and then compare with Python Tutor!

```
la = [8,7,7]
lb = [9,6,7,5]
lc = lb
la = lb
print('la è', la)
print('lb è', lb)
print('lc è', lc)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: it will print:

```
la è [9, 6, 7, 5]
lb è [9, 6, 7, 5]
lc è [9, 6, 7, 5]
```

because

```
la = [8,7,7]
lb = [9,6,7,5]

lc = lb   # variable lc is associated to the same identical list of lb

la = lb   # variable la is associated to the same identical list of lb
          # the list previously associated to la is lost
```

</div>

[35]:
```
la = [8,7,7]
lb = [9,6,7,5]
lc = lb
la = lb
#print('la è', la)
#print('lb è', lb)
#print('lc è', lc)
jupman.pytut()
```

[35]:
```
<IPython.core.display.HTML object>
```

QUESTION: Look at the following code. After its execution, by printing `la`, `lb` and `lc` what will we get?

Try drawing **ON PAPER** what is happening in memory, then compare the result with Python Tutor!

```
la = [7,8,5]
lb = [6,7]
lc = lb
lb = la
lc[0] = 9
print('la is', la)
print('lb is', lb)
print('lc is', lc)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The print will produce

```
la è [7, 8, 5]
lb è [7, 8, 5]
lc è [9, 7]
```

because :

```
la = [7,8,5]
lb = [6,7]
# the variable lc is assigned to the same list of lb [6,7]
lc = lb
# the variable lb is associated to the same list of la [7,8,5].
# This doesn't change the assignment of lc, which remains associated to [6,7] !
lb = la
# Modifies the first element of the list associated to lc which from [6,7] becomes [9,
↪7]
lc[0] = 9
print('la è', la)
print('lb è', lb)
print('lc è', lc)
```

</div>

```
[36]: la = [7,8,5]
lb = [6,7]
lc = lb
lb = la
lc[0] = 9
#print('la è', la)
#print('lb è', lb)
#print('lc è', lc)

jupman.pytut()
```

```
[36]: <IPython.core.display.HTML object>
```

5.7.7 Slices

We can extract sequences from lists by using *slices*. A slice is produced by placing square brackets after the list with inside the starting index (INCLUDED), followed by a colon `:`, followed by the end index (EXCLUDED). It works exactly as with strings: in that case the slice produces a new string, in this case it produces a NEW list. Let's see an example:

```
[37]:      #0  1  2  3  4  5  6  7  8  9
la = [43,35,82,75,93,12,43,28,54,65]
```

```
[38]: la[3:7]
```

```
[38]: [75, 93, 12, 43]
```

We extracted a NEW list `[75, 93, 12, 43]` from the list `la` starting from index `3` INCLUDED until index `7` EXCLUDED. We can see the original list is preserved:

```
[39]: la
```

```
[39]: [43, 35, 82, 75, 93, 12, 43, 28, 54, 65]
```

Let's verify what happens with Python Tutor, by assigning the new list to a variable `lb`:

```
[40]: #     0  1  2  3  4  5  6  7  8  9
la = [43,35,82,75,93,12,43,28,54,65]
lb = la[3:7]

jupman.pytut()
```

```
[40]: <IPython.core.display.HTML object>
```

You will notice a NEW memory region, associated to variable `lb`.

Slice - limits

When we operate with slices we must be careful about indeces limits. Let's see how they behave:

```
[41]: #0  1  2  3  4
      [58,97,76,87,99][0:3]   # from index 0 *included* to 3 *excluded*
```

```
[41]: [58, 97, 76]
```

```
[42]: #0  1  2  3  4
      [58,97,76,87,99][0:4]   # from index 0 *included* a 4 *excluded*
```

```
[42]: [58, 97, 76, 87]
```

```
[43]: #0  1  2  3  4
      [58,97,76,87,99][0:5]   # from index 0 *included* to 5 *excluded*
```

```
[43]: [58, 97, 76, 87, 99]
```

```
[44]: #0  1  2  3  4
      [58,97,76,87,99][0:6]    # if we go beyond the list length Python does not complain
```

```
[44]: [58, 97, 76, 87, 99]
```

```
[45]: #0  1  2  3  4
      [58,97,76,87,99][8:12] # Python doesn't complain even if we start from non-existing␣
      ↪indeces
```

```
[45]: []
```

QUESTION: This expression:

```
[][3:8]
```

1. produces a result (which one?)
2. produces an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: given an empty list, we are trying yo create a sublist which goes from index `3` INCLUDED to index `8` EXCLUDED. As we've seen before, if we start after the limit and also if we go beyond the limit Python does not complain, and when elements are not found we are simply served with an empty list.

</div>

QUESTION: if `x` is some list (may also empty), what does this expression do? Can it give an error? Does it return something useful?

```
x[0:len(x)]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Always return a NEW copy of the entire list, because it starts from index 0 *INCLUDED* and ends at index `len(x)` *EXCLUDED*.

It also works with the empty list, because `[][0:len([])]` is equivalent to `[][0:0]` that is sublist from 0 *included* to 0 *excluded*, so we are not taking any character and are not going beyod list limits. In fact, as we've seen before, even if we went beyond Python wouldn't complain.

</div>

Slices - omitting limits

If we will, it is possible to omit start index, in which case Python will suppose it's `0`:

```
[46]: #0  1  2  3  4  5  6  7  8  9
      [98,67,85,77,65,99,67,55,79][:3]
```

```
[46]: [98, 67, 85]
```

It is also possible to omit the end index, in this case Python will extract elements until the list end:

```
[47]: #0  1  2  3  4  5  6  7  8  9
      [98,67,85,77,65,99,67,55,79][3:]
```

```
[47]: [77, 65, 99, 67, 55, 79]
```

By omitting both indexes we obtain the full list:

```
[48]: #0  1  2  3  4  5  6  7  8  9
      [98,67,85,77,65,99,67,55,79][:]
```

```
[48]: [98, 67, 85, 77, 65, 99, 67, 55, 79]
```

QUESTION: What is this code going to print? Will `la` get modified or not?

```
la = [7,8,9]
lb = la[:]
lb[0] = 6
print('la =',la)
print('lb =',lb)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `lb = la[:]` creates a NEW list containing all the elements which are in `la`. When we write `lb[0] = 6` we are only modifying the memory region associated to `lb`. If you observe it in Python Tutor, you will see that `la` and `lb` are pointing to different memory regions:

</div>

```
[49]: la = [7,8,9]
      lb = la[:]
      lb[0] = 6
      #print('la =',la)
      #print('lb =',lb)

      jupman.pytut()
```

```
[49]: <IPython.core.display.HTML object>
```

QUESTION: For each of the following expressions, try guessing which value it produces, or if it gives an error.

1. `[9,7,8,6][1:1]`

2. `[9,7,8,6][1:2]`

3. `[9,7,8,6][2:3][0]`
4. `[][]`
5. `[][:]`
6. `[3][:]`
7. `[:][]`

Slices - negative limits

It is also possible to set inverse and negative limits, although it is not always intuitive:

```
[50]: #0  1  2  3  4  5  6
      [73,48,19,57,64,15,92][3:0]   # from index 3 to positive indexes <= 3 produces nothing
```

```
[50]: []
```

```
[51]: #0  1  2  3  4  5  6
      [73,48,19,57,64,15,92][3:1]   # from index 3 to positive indexes <= 3 produces nothing
```

```
[51]: []
```

```
[52]: #0  1  2  3  4  5  6
      [73,48,19,57,64,15,92][3:2]   # from index 3 to positive indexes <= 3 produces nothing
```

```
[52]: []
```

```
[53]: #0  1  2  3  4  5  6
      [73,48,19,57,64,15,92][3:3]   # from index 3 to positive indexes <= 3 produces nothing
```

```
[53]: []
```

Let's see what happens with negative indexes:

```
[54]: # 0  1  2  3  4  5  6
      #-7 -6 -5 -4 -3 -2 -1
      [73,48,19,57,64,15,92][3:-1]
```

```
[54]: [57, 64, 15]
```

```
[55]: # 0  1  2  3  4  5  6
      #-7 -6 -5 -4 -3 -2 -1
      [73,48,19,57,64,15,92][3:-2]
```

```
[55]: [57, 64]
```

```
[56]: # 0  1  2  3  4  5  6
      #-7 -6 -5 -4 -3 -2 -1
      [73,48,19,57,64,15,92][3:-3]
```

```
[56]: [57]
```

[57]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][3:-4]
```

[57]:
```
[]
```

[58]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][3:-5]
```

[58]:
```
[]
```

It is also possible to start from a negative index and arrive to a positive one. As long as the first index marks a position which precedes the second index, something gets returned:

[59]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-7:3]
```

[59]:
```
[73, 48, 19]
```

[60]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-6:3]
```

[60]:
```
[48, 19]
```

[61]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-5:3]
```

[61]:
```
[19]
```

[62]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-4:3]
```

[62]:
```
[]
```

[63]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-3:3]
```

[63]:
```
[]
```

[64]:
```
# 0  1  2  3  4  5  6
#-7 -6 -5 -4 -3 -2 -1
[73,48,19,57,64,15,92][-2:3]
```

[64]:
```
[]
```

QUESTION: For each of the following expressions, try guessing which value is produced, or if it gives an error

1. `[9,7,8,6][0:-2]`

2. `[0:-2][9,7,8,6]`

3. `[5,7,9][1:-1]`

4. `[][-13:-17]`

5. `[9,7,8,6][-4:-1]`

6. `[9,7,8,6][-5:-1]`

7. `[9,7,8,6,10,32][-3:1]`

8. `[9,7,8,6,10,32][-3:5]`

Slices - modifying

Suppose we have the list

[65]:
```
    # 0  1  2  3  4  5  6  7
la = [12,23,35,41,74,65,34,22]
```

and we want to change `la` cells from index `3` INCLUDED to index `6` EXCLUDED in such a way they contain the numbers taken from list `[98,96,97]`. We can do it with this special notation which allows us to write a slice *to the left* of operator =:

[66]:
```
la[3:6] = [98,96,97]
```

[67]:
```
la
```

[67]:
```
[12, 23, 35, 98, 96, 97, 34, 22]
```

In this slightly more complex example we verify in Python Tutor that the original memory region gets actually modifyied:

[68]:
```
#     0  1  2  3  4  5  6  7
la = [12,23,35,41,74,65,34,22]
lb = la
lb[3:6] = [98,96,97]

jupman.pytut()
```

[68]:
```
<IPython.core.display.HTML object>
```

QUESTION: Look at the following code - what does it produce?

```
la = [9,6,5,8,2]
la[1:4] = [4,7,0]
print(la)
```

1. modify `la` (how?)
2. an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1 - MODIFIES `la` like this:

```
# 0  1  2  3  4
[ 9, 4, 7, 0, 2]
```

so from index `1` INCLUDED to index `4` EXCLUDED

</div>

QUESTION: Look at the following code. What does it produce?

```
la = [7,6,8,4,2,4,2,3,1]
i = 3
lb = la[0:i]
la[i:2*i] = lb
print(la)
```

1. modifies `la` (how?)
2. an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1 - modifies `la` by copying first `i` cells into successive ones.

</div>

5.7.8 List of strings

We said we can put any object into a list, for example some strings:

[69]:
```
vegetables = ['tomatoes', 'onions', 'carrots', 'cabbage']
```

Let's try extracting a vegetable by writing this expression:

[70]:
```
vegetables[2]
```

[70]:
```
'carrots'
```

Now, the preceding expression produces the result `'carrots'`, which we know is a string. This suggests we can use the expression exactly like if it were a string.

Suppose we want to obtain the first character of the string `'carrots'`, if we directly have the string we can write like this:

[71]:
```
'carrots'[0]
```

[71]:
```
'c'
```

But if the string is inside the previous list, we could directly do like this:

[72]:
```
vegetables[2][0]
```

[72]:
```
'c'
```

Exercise - province codes

Given a list with exactly 4 province codes in lowercase, write some code which creates a NEW list containing the same codes in uppercase characters.

- your code must work with any list of 4 provinces
- hint: if you don't remember the right method, have a look here[207]

Example 1 - given:

```
provinces = ['tn','mi','to','ro']
```

your code must print:

```
['TN', 'MI', 'TO', 'RO']
```

Example 2 - given:

```
provinces = ['pa','ge','ve', 'aq']
```

Your code must print:

```
['PA', 'GE', 'VE', 'AQ']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[73]:
```
provinces = ['tn','mi','to','ro']
#provinces = ['pa','ge','ve', 'aq']

# write here

#print([provinces[0].upper(), provinces[1].upper(), provinces[2].upper(),␣
↪provinces[3].upper()])
```

</div>

[73]:
```
provinces = ['tn','mi','to','ro']
#provinces = ['pa','ge','ve', 'aq']

# write here
```

Exercise - games

Given a list `games` of exactly 3 strings, write some code which MODIFIES the list so it contains only the first characters of each string.

- Your code must work with any list of exactly 3 strings

Example - given

```
games = ["Monopoly",
         "RISK",
         "Bingo"]
```

[207] https://en.softpython.org/strings/strings3-sol.html

After executing the code, it must result:

```
>>> print(games)
["M","R","B"]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[74]: games = ["Monopoly",
               "RISK",
               "Bingo"]


      # write here

      games = [ games[0][0], games[1][0], games[2][0] ]
      #print(games)
```

</div>

```
[74]: games = ["Monopoly",
               "RISK",
               "Bingo"]


      # write here
```

5.7.9 List of lists

NOTE: We will talk much more in detail of lists of lists in the tutorial Matrices - list of lists[208], this is just a brief introduction.

The consideration we've seen so far about string lists are also valid for a list of lists:

```
[75]: couples = [             # external list
                  [67,95],    # internal list at index 0
                  [60,59],    # internal list at index 1
                  [86,75],    # internal list at index 2
                  [96,90],    # internal list at index 3
                  [88,87],    # internal list at index 4
                ]
```

If we want ot extract the number `90`, we must first extract the sublist from index `3`:

```
[76]: couples[3]   # NOTE: the expression result is a list
```

```
[76]: [96, 90]
```

and so in the extracted sublist (which has only two elements) we can recover the number at index `0`:

```
[77]: couples[3][0]
```

```
[77]: 96
```

[208] https://en.softpython.org/matrices-lists/matrices-lists-sol.html

and at index `1`:

[78]:
```
couples[3][1]
```

[78]:
```
90
```

Exercise - couples

1. Write some code to extract and print the number `86`, `67` and `87`
2. Given a row with index `i` and a column `j`, print the number at row `i` and column `j` multiplied by the number at successive row and same column

After your code, you should see printed

```
point 1: 86 67 87

point 2:  i = 3  j = 1  result = 7830
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[79]:
```
couples = [              # external list
            [67,95],     # internal list at index 0
            [60,59],     # internal list at index 1
            [86,75],     # internal list at index 2
            [96,90],     # internal list at index 3
            [88,87],     # internal list at index 4
          ]

i = 3
j = 1

# write here

print("point 1:", couples[2][0], couples[0][0], couples[4][1])
print()
print("point 2:"," i =",i," j =",j," result =", couples[i][j]*couples[i+1][j])
```

```
point 1: 86 67 87

point 2:  i = 3  j = 1  result = 7830
```

</div>

[79]:
```
couples = [              # external list
            [67,95],     # internal list at index 0
            [60,59],     # internal list at index 1
            [86,75],     # internal list at index 2
            [96,90],     # internal list at index 3
            [88,87],     # internal list at index 4
          ]

i = 3
j = 1

# write here
```

(continues on next page)

(continued from previous page)

```
point 1: 86 67 87

point 2:  i = 3  j = 1  result = 7830
```

Exercise - nonunif

Given a list `nonunif` of sublists of any length, and a row at index `i`, write some code which MODIFIES the sublists of `nonunif` at row `i` and successive one in such a way the last element of both lists becomes `99`.

- your code must work with any `nonunif` and any `i`

Example 1 - given:

```
nonunif = [                    # external list
            [67,95],           # internal at index 0
            [60,23,23,13,59],  # internal at index 1
            [86,75],           # internal at index 2
            [96,90,92],        # internal at index 3
            [88,87],           # internal at index 4
         ]

i = 1
```

after your code, by writing (we use `pprint` because it will print on many lines)

```
from pprint import pprint
pprint(nonunif,width=30)
```

it should print:

```
[[67, 95],
 [60, 23, 23, 13, 99],
 [86, 99],
 [96, 90, 92],
 [88, 87]]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[80]:
```
nonunif = [                    # external list
            [67,95],           # internal list at index 0
            [60,23,23,13,59],  # internal list at index 1
            [86,75],           # internal list at index 2
            [96,90,92],        # internal list at index 3
            [88,87],           # internal list at index 4
         ]

i = 1

# write here

nonunif[i][-1] = 99
nonunif[i+1][-1] = 99
```

(continues on next page)

(continued from previous page)

```
from pprint import pprint
pprint(nonunif,width=30)
```

```
[[67, 95],
 [60, 23, 23, 13, 99],
 [86, 99],
 [96, 90, 92],
 [88, 87]]
```

</div>

[80]:
```
nonunif = [                     # external list
            [67,95],            # internal list at index 0
            [60,23,23,13,59],   # internal list at index 1
            [86,75],            # internal list at index 2
            [96,90,92],         # internal list at index 3
            [88,87],            # internal list at index 4
         ]

i = 1

# write here
```

```
[[67, 95],
 [60, 23, 23, 13, 99],
 [86, 99],
 [96, 90, 92],
 [88, 87]]
```

5.7.10 `in` operator

To verify whether an object is contained in a list, we can use the `in` operator.

Note the result of this expression is a boolean:

[81]:
```
9 in [6,8,9,7]
```

[81]:
```
True
```

[82]:
```
5 in [6,8,9,7]
```

[82]:
```
False
```

[83]:
```
"apple" in ["watermelon","apple","banana"]
```

[83]:
```
True
```

[84]:
```
"carrot" in ["watermelon","apple","banana"]
```

[84]:
```
False
```

QUESTION: What's the result of this expression? `True` or `False`?

```
True in [ 5 in [6,7,5],
          2 in [8,1]
        ]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Gives back `True` because

```
[ 5 in [6,7,5],
  2 in [8,1]
]
```

represents a list of two elements. Each element is an expression with `in` which gets evaluated. In the first case, `5 in [6,7,5]` results `True`. So the final list becomes `[True, False]` and by writing `True in [True, False]` we obtain `True`

</div>

not in

We can write the check of **non** belonging in two ways:

Way 1:

```
[85]: "carrot" not in ["watermelon","banana","apple"]
```

```
[85]: True
```

```
[86]: "watermelon" not in ["watermelon","banana","apple"]
```

```
[86]: False
```

Way 2:

```
[87]: not "carrot" in ["watermelon","banana","apple"]
```

```
[87]: True
```

```
[88]: not "watermelon" in ["watermelon","banana","apple"]
```

```
[88]: False
```

QUESTION: Given any element `x` and list `y`, what does the following expression produce?

```
x in y and not x in y
```

1. `False`
2. `True`
3. `False` or `True` according to the values of `x` and `y`
4. an error

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1. Gives back `False`, because internally Python brackets the expression like this:

```
(x in y) and (not x in y)
```

and one element cannot be both contained in the list and not contained in the same list

</div>

QUESTION: For each of the following expressions, try to guess the result

1. ```
   3 in [3]
   ```
2. ```
   [4,5] in [1,2,3,4,5]
   ```
3. ```
   [4,5] in [[1,2,3],[4,5]]
   ```
4. ```
   [4,5] in [[1,2,3,4],[5,6]]
   ```
5. ```
   'n' in ['alien'[-1]]
   ```
6. ```
   'rts' in 'karts'[1:4]
   ```
7. ```
   [] in [[[]]]
   ```
8. ```
   [] in [[]]
   ```
9. ```
   [] in ["[]"]
   ```

QUESTION: For each of the following expressions, independently from the value of `x`, tell whether it always results `True`:

1. ```
   x in x
   ```
2. ```
   x in [x]
   ```
3. ```
   x not in []
   ```
4. ```
   x in [[x]]
   ```
5. ```
   x in [[x][0]]
   ```
6. ```
   (x and y) in [x,y]
   ```
7. ```
   x in [x,y] and y in [x,y]
   ```

Exercise - vegetables

Given the list `vegetables` of exactly 5 strings and the list of strings `fruits`, MODIFY the variable `vegetables` so that in each cell there is `True` if the vegetable is a fruit or `False` otherwise.

- your code must work with any list of 5 strings `vegetables` and any list `fruits`

Example - given:

```
vegetables = ["carrot",
              "cabbage",
              "apple",
              "aubergine",
              "watermelon"]

fruits = ["watermelon","banana","apple",]
```

after execution your code must print:

```
>>> print(vegetables)
[False, False, True, False, True]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[89]:
```
vegetables = ["carrot",
              "cabbage",
              "apple",
              "aubergine",
              "watermelon"]

fruits = ["watermelon","banana","apple",]

# write here

vegetables = [vegetables[0] in fruits,
              vegetables[1] in fruits,
              vegetables[2] in fruits,
              vegetables[3] in fruits,
              vegetables[4] in fruits,
]

#print(vegetables)
```

</div>

[89]:
```
vegetables = ["carrot",
              "cabbage",
              "apple",
              "aubergine",
              "watermelon"]

fruits = ["watermelon","banana","apple",]

# write here
```

5.7.11 List concatenation with +

Given two lists `la` and `lb`, we can concatenate them with the operator + which produces a NEW list:

[90]:
```python
la = [77,66,88]
lb = [99,55]

la + lb
```

[90]:
```
[77, 66, 88, 99, 55]
```

Note the operator + produces a NEW list, so `la` and `lb` remained unchanged:

[91]:
```python
print(la)
```

```
[77, 66, 88]
```

[92]:
```python
print(lb)
```

```
[99, 55]
```

Let's check with Python Tutor:

[93]:
```python
la = [77,66,88]
lb = [99,55]
lc = la + lb

print(la)
print(lb)
print(lc)

jupman.pytut()
```

```
[77, 66, 88]
[99, 55]
[77, 66, 88, 99, 55]
```

[93]:
```
<IPython.core.display.HTML object>
```

Exercise - concatenation

Write some code which given lists `la` and `lb`, puts into list `lc` the last two elements of `la` and the first two of `lb`

Example - given:

```python
la = [18,26,30,45,55]
lb = [16,26,37,45]
```

after your code it must print:

```
>>> print(la)
[18, 26, 30, 45, 55]
>>> print(lb)
[16, 26, 37, 45]
>>> print(lc)
[45, 55, 37, 45]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[94]: la = [18,26,30,45,55]
lb = [16,26,37,45]

# write here
lc = la[-2:] + lb[2:]
#print(la)
#print(lb)
#print(lc)
```

</div>

```
[94]: la = [18,26,30,45,55]
lb = [16,26,37,45]

# write here
```

QUESTION: For each of the following expressions, try guessing the result

1. `[6,7,8] + [9]`
2. `[6,7,8] + []`
3. `[] + [6,7,8]`
4. `[] + []`
5. `[] + [[]]`
6. `[[]]+[]`
7. `[[]]+[[]]`
8. `([6] + [8])[0]`
9. `([6] + [8])[1]`
10. `([6] + [8])[2:]`
11. `len([4,2,5])+len([3,1,2])`
12. `len([4,2,5] + [3,1,2])`
13. `[5,4,3] + "3,1"`
14. `[5,4,3] + "[3,1]"`
15. `"[5,4,3]" + "[3,1]"`

16.
```
["4","1","7"] + ["3","1"]
```

17.
```
list('coca') + ['c','o','l','a']
```

5.7.12 min and max

A list is a sequence of elements, and as such we can pass it to functions `min` or `max` for finding respectively the minimum or the maximum element of the list.

[95]:
```
min([4,5,3,7,8,6])
```

[95]:
```
3
```

[96]:
```
max([4,5,3,7,8,6])
```

[96]:
```
8
```

> V COMMANDMENT[209] **: You shall never ever use `min` and `max` as variable names.**
>
> (adapted) If you do, you will lose the functions!

Note it's also possible to directly pass to `min` and `max` the elements to compare without including them in a list:

[97]:
```
min(4,5,3,7,8,6)
```

[97]:
```
3
```

[98]:
```
max(4,5,3,7,8,6)
```

[98]:
```
8
```

But if we pass only one, without including it in a list, we will get an error:

```
min(4)
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-156-bb3db472b52e> in <module>
----> 1 min(4)

TypeError: 'int' object is not iterable
```

The error tells us that when we pass only an argument, Python expects a sequence like a list:

[99]:
```
min([4])
```

[99]:
```
4
```

To `min` and `max` we can also pass strings, and we will get the character which is alphabetically lesser or greater:

[100]:
```
min("orchestra")
```

[100]:
```
'a'
```

[209] https://en.softpython.org/commandments.html#V-COMMANDMENT

[101]:
```python
max("orchestra")
```

[101]:
```
't'
```

If we pass a list of strings, we will obtain the lesser or greater string in lexicographical order (i.e. the phonebook order)

[102]:
```python
min(['the', 'sailor', 'walks', 'around', 'the', 'docks'])
```

[102]:
```
'around'
```

[103]:
```python
max(['the', 'sailor', 'walks', 'around', 'the', 'docks'])
```

[103]:
```
'walks'
```

QUESTION: For each of the following expressions, try guessing the result (or if it gives an error)

1. `max(7)`
2. `max([7])`
3. `max([5,4,6,2])`
4. `max([min([7,3])])`
5. `max([])`
6. `max(2,9,3)`
7. `max([3,2,5] + [9,2,3])`
8. `max(max([3,2,5], max([9,2,3]))`
9. `max(min(3,6), min(8,2))`
10. `min(max(3,6), max(8,2))`
11. `max(['a','b','d','c'])`
12. `max(['barca', 'dado', 'aloa', 'cerchio'])`
13. `min(['prova','','z','v'])`
14. `max(['martello'[-1],'cacciavite'[-1],'brugola'[-1]])`
15. `min(['martello'[-1],'cacciavite'[-1],'brugola'[-1]])`

5.7.13 sum

With `sum` we can sum all the elements in a list:

```
[104]: sum([1,2,3])
```

```
[104]: 6
```

```
[105]: sum([1.0, 2.0, 0.14])
```

```
[105]: 3.14
```

> V COMMANDMENT[210] **: You shall never ever use `sum` as a variable name**
>
> (adapted) If you do, you will lose the function!

QUESTION: For each of the following expressions, try guessing the result (or if it gives an error):

1. ```
   sum[3,1,2]
   ```
2. ```
   sum(1,2,3)
   ```
3. ```
   la = [1,2,3]
   sum(la) > max(la)
   ```
4. ```
   la = [1,2,3]
   sum(la) > max(la)*len(la)
   ```
5. ```
   la = [4,2,6,4,7]
   lb = [max(la), min(la), max(la)]
   print(max(lb) != max(la))
   ```

Exercise - balance

Given a list of `n` numbers `balance` with `n` even, write some code which prints `True` if the sum of all first `n/2` numbers is equal to the sum of all successive ones.

- your code must work for *any* number list

Example 1 - given:

```
balance = [4,3,7,1,5,8]
```

after your code, it must print:

```
True
```

Example 2 - given:

```
balance = [4,3,3,1,9,8]
```

after your code, it must print:

[210] https://en.softpython.org/commandments.html#V-COMMANDMENT
```

```
False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[106]: balance = [4,3,7,1,5,8]
       #bilancia = [4,3,3,1,9,8]

       # write here
       n = len(balance)
       sum(balance[:n//2]) == sum(balance[n//2:])
```

```
[106]: True
```

</div>

```
[106]: balance = [4,3,7,1,5,8]
       #bilancia = [4,3,3,1,9,8]

       # write here
```

```
[106]: True
```

5.7.14 Multiplying lists

To replicate the elements of a list, it's possible to use the operator * which produces a NEW list:

```
[107]: [7,6,8] * 2
```

```
[107]: [7, 6, 8, 7, 6, 8]
```

```
[108]: [7,6,8] * 3
```

```
[108]: [7, 6, 8, 7, 6, 8, 7, 6, 8]
```

Note a NEW list is produced, and the original one is not modified:

```
[109]: la = [7,6,8]
```

```
[110]: lb = [7,6,8] * 3
```

```
[111]: la    # original
```

```
[111]: [7, 6, 8]
```

```
[112]: lb    # expression result
```

```
[112]: [7, 6, 8, 7, 6, 8, 7, 6, 8]
```

We can multiply a list of strings:

```
[113]: la = ["a", "world", "of", "words"]
```

[114]:
```
lb = la * 2
```

[115]:
```
print(la)
```

```
['a', 'world', 'of', 'words']
```

[116]:
```
print(lb)
```

```
['a', 'world', 'of', 'words', 'a', 'world', 'of', 'words']
```

As long as we multiply lists which contain immutable elements like numbers or strings, no particular problems arise:

[117]:
```
la = ["a", "world", "of", "words"]
lb = la * 2

jupman.pytut()
```

[117]:
```
<IPython.core.display.HTML object>
```

The matter becomes much more sophisticated when we multiply lists which contain mutable objects like other lists. Let's see an example:

[118]:
```
la = [5,6]
lb = [7,8,9]
lc = [la,lb] * 2
```

[119]:
```
print(la)
```

```
[5, 6]
```

[120]:
```
print(lb)
```

```
[7, 8, 9]
```

[121]:
```
print(lc)
```

```
[[5, 6], [7, 8, 9], [5, 6], [7, 8, 9]]
```

By printing it, we see that the lists `la` and `lb` are represented inside `lc` - but how, exactly? `print` calls may trick you about the effective state of memory - to investigate further it's convenient to use Python Tutor:

[122]:
```
la = [5,6]
lb = [7,8,9]
lc = [la,lb] * 2

jupman.pytut()
```

[122]:
```
<IPython.core.display.HTML object>
```

Arggh ! A jungle of arrows will appear ! This happens because when we write `[la, lb]` we create a list with two *references* to other lists `[5,6]` and `[7,8,9]`, and the operator `*` when duplicating it just copies *references*.

For now we stop here, we will see the implications details later in the tutorial matrices - lists of lists[211]

[211] https://en.softpython.org/matrices-lists/matrices-lists-sol.html

5.7.15 Equality

We can check whether two lists are equal with equality operator ==, which given two lists returns `True` if they contain equal elements or `False` otherwise:

```
[123]: [4,3,6] == [4,3,6]
```

```
[123]: True
```

```
[124]: [4,3,6] == [4,3]
```

```
[124]: False
```

```
[125]: [4,3,6] == [4,3,6, 'ciao']
```

```
[125]: False
```

```
[126]: [4,3,6] == [2,2,8]
```

```
[126]: False
```

We can check equality of lists with heterogenous elements:

```
[127]: ['apples', 3, ['cherries', 2], 6] == ['apples', 3, ['cherries', 2], 6]
```

```
[127]: True
```

```
[128]: ['bananas', 3,['cherries', 2], 6] == ['apples', 3, ['cherries', 2], 6]
```

```
[128]: False
```

To check for inequality, we can use the operatpr `!=`:

```
[129]: [2,2,8] != [2,2,8]
```

```
[129]: False
```

```
[130]: [4,6,0] != [2,2,8]
```

```
[130]: True
```

```
[131]: [4,6,0] != [4,6,0,2]
```

```
[131]: True
```

QUESTION: For each of the following expressions, guess whether it is `True`, `False` or it produces an error:

1. `[2,3,1] != [2,3,1]`
2. `[4,8,12] == [2*2,4*2,6*2]`
3. `[7,8][:] == [7,9-1]`
4. `[7][0] == [[7]][0]`
5. `[9] == [9][0]`

6.
```python
[max(7,9)] == [max([7]),max([9])]
```

7.
```python
['a','b','c'] == ['A','B','C']
```

8.
```python
['a','b'] != ['a','b','c']
```

9.
```python
["ciao"] != ["CIAO".lower()]
```

10.
```python
[True in [True]] != [False]
```

11.
```python
[][:] == []
```

12.
```python
[[]] == [] + []
```

13.
```python
[[],[]] == [] + []
```

14.
```python
[[[]]] == [[[]+[]]]
```

5.7.16 Continue

You can find more exercise in the notebook Lists 3[212]

```
[ ]:
```

5.8 Lists 3 - Methods

5.8.1 Download exercises zip

Browse files online[213]

Lists are objects of type `list` and have several methods for performing operations on them:

Method	Returns	Description
list.append(obj)	`None`	Adds a new element at the end of the list
list.extend(list)	`None`	Adds many elements at the end of the list
list.insert(int,obj)	`None`	Adds a new element into some given position
list.remove(obj)	`None`	Removes the first occurrence of the element
list.pop()	obj	Removes and return the element at last position
list.pop(int)	obj	Given an index, removes and return the element at that position
list.reverse()	`None`	Inverts the order of elements
list.sort()	`None`	Sorts the elements *in-place*
list.index(obj)	`int`	Finds the first occurrence of an element and returns the position
list.count(obj	`int`	Counts the occurrences of an object

[212] https://en.softpython.org/lists/lists3-sol.html

[213] https://github.com/DavidLeoni/softpython-en/tree/master/lists

WARNING 1: LIST METHODS *MODIFY* THE LIST ON WHICH ARE CALLED !

Whenever you call a method of a list (the object to the left of the dot `.`), you MODIFY the list itself (differently from string methods which always generate a new string without changing the original)

WARNING 2: LIST METHODS RETURN NOTHING!

They almost always return the object `None` (differently from strings which always return a new string)

5.8.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
lists
    lists1.ipynb
    lists1-sol.ipynb
    lists2.ipynb
    lists2-sol.ipynb
    lists3.ipynb
    lists3-sol.ipynb
    lists4.ipynb
    lists4-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `lists3.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.8.3 `append` method

We can MODIFY a list adding a single element at a time with the method `append`.

Suppose to start from an empty list:

[2]:
```
la = []
```

If we want to add as element the number `57`, we can write like this:

[3]:
```python
la.append(57)
```

Note the list we initalilly created got MODIFIED:

[4]:
```python
la
```

[4]:
```
[57]
```

> **WARNING:** `la.append(57)` **returned NOTHING !!!!**
>
> Observe carefully the output of cell with instruction `la.append(57)`, you will notice there is absolutely nothing. This happens because the purpose of `append` is to MODIFY the list on which it is called, NOT generating new lists.

We append another number *at the end* of the list:

[5]:
```python
la.append(96)
```

[6]:
```python
la
```

[6]:
```
[57, 96]
```

[7]:
```python
la.append(74)
```

[8]:
```python
la
```

[8]:
```
[57, 96, 74]
```

Let's see what happened in Python Tutor:

[9]:
```python
# WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#          (it's sufficient to execute it only once)

import jupman
```

[10]:
```python
la = []
la.append(57)
la.append(96)
la.append(74)

jupman.pytut()
```

[10]:
```
<IPython.core.display.HTML object>
```

Note there is only one yellow memory region associated to variable `la` which gets expanded as you click on Next.

We said `append` method returns nothing, let's try to add some detail. In the methods table, there is present a column named *Returns*. If you check it, for almost all methods included `append` there is indicated it returns `None`.

`None` is the most boring object in Python, because it literally means nothing. What can you do with nothing? Very few things, so few that whenever Jupyter finds as result the `None` object it doesn't even print it. Try directly inserting `None` in a cell, you will see it won't be reported in cell output:

[11]:
```python
None
```

A way to force the print is by using the command `print`:

[12]:
```
print(None)
```

```
None
```

EXERCISE: What is the type of the object `None`? Discover it by using the funciton `type`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[13]:
```
# write here

#type(None)
```

</div>

[13]:
```
# write here
```

Let's try repeating what happens with `append`. If you call the method `append` on a list, `append` silently MODIFIES the list, and RETURNS the object `None` as call result. Notice that Jupyter considers this object as non-interesting, so it doesn't even print it.

Let's try to get explicit about this misterious `None`. If it's true that `append` produces it as call result, it means we can associate this result to some variable. Let's try to associate it to variable `x`:

[14]:
```
la = []
x = la.append(78)
```

Now, if everything went as we wrote, `append` should have modified the list:

[15]:
```
la
```

[15]:
```
[78]
```

and there should be associated `None` to variable `x`. So, if we ask Jupyter to show the value associated to `x` and that value is `None`, nothing will appear:

[16]:
```
x
```

Note there is no output in the cell, apparently we are really in presence of a `None`. Let's force the `print`:

[17]:
```
print(x)
```

```
None
```

Here it is! Probably you will be a little confused by all of this, so let's check again what happens in Python Tutor:

[18]:
```
la = []
x = la.append(78)
print("la is", la)
print("x is", x)

jupman.pytut()
```

```
la is [78]
x is None
```

[18]:
```
<IPython.core.display.HTML object>
```

What's the final gist?

> **REUSING THE RESULT OF LIST METHODS CALLS IS ALMOST ALWAYS AN ERROR !**
>
> Since calling list methods returns `None`, which is a 'useless' object, trying to reuse it will almost surely produce an error

EXERCISE: Build a list by adding one element at a time with the method `append`. Add the elements `77`, `"test"`, `[60,93]` with three calls to `append`, and finally print the list.

After your code, you should see `[77, 'test', [60, 93]]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[19]:
```python
la = []

# write here
la.append(77)
la.append("test")
la.append([60, 93])

#print(la)
```

</div>

[19]:
```python
la = []

# write here
```

QUESTION: The following code:

```python
la = []
la.append(85,70,94)
```

1. produces an error (which one?)
2. modifies the list (how?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1: `append` accepts only one argument, by passing more than one will produce an error, to see which try to execute the code in a cell.

</div>

QUESTION: The following code

```python
la = []
la.append(87).append(96)
```

1. produces an error
2. appends to `la` the numbers `87` and `96`

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1: produces an error, because we said the call to `la.append(87)` MODIFIES the list `la` on which it is called and return the value `None`. If on `None` we try calling `.append(96)`, since `None` is not a list we will get an error message. Make sure of this using Python Tutor.

</div>

QUESTION: let's briefly go back to strings. Look at the following code (if you don't remember what string methods do see here[214])

```
sa = '    trento    '
sb = sa.strip().capitalize()
print(sb)
```

1. produces an error (which one?)
2. changes `sa` (how?)
3. prints something (what?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: prints `Trento`. Differently from lists, the strings are *immutable* sequences: this means that when you call a method of strings you are sure it will RETURN a NEW string. So the first call to `sa.strip()` RETURNS the string without spaces at beginning and end of `'trento'`, and on this string the method `capitalize()` is called to make the first character uppercase.

If this is not clear to you, try to executing the following code in Python Tutor. It is equivalent to the one in the example but it explicitly shows the passage by assigning the result of calling `sa.strip()` to the extra variable `x`

```
sa = '    trento    '
x = sa.strip()
sb = x.capitalize()
print(sb)
```

</div>

QUESTION: Have a look at this code. Will it print something at the end? Or will it produce an error?

```
la = []
lb = []
la.append(lb)

lb.append(98)
lb.append(77)

print(la)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: It will print `[[98, 77]]`, because we put `lb` inside `la`.

Even if with first `append` we added `lb` as first element of `la`, aftwerwards it is perfectly lecit keeping on modifying `lb` by calling `lb.append(98)`.

Try executing the code in Python Tutor, and see the arrows.

[214] https://en.softpython.org/strings/strings3-sol.html#Methods

</div>

Exercise - augmenting a list 1

Given the list `la` of *fixed dimension 7,* write some code to augment the empty list `lb` so to *only* contain the elements of `la` with even index (0, 2, 4, …).

- Your code should work with *any* list `la` of fixed dimension 7

```
#    0 1 2 3 4 5 6
la=[8,4,3,5,7,3,5]
lb=[]
```

After your code, you should obtain:

```
>>> print(lb)
[8,3,7,5]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[20]:

```
#    0 1 2 3 4 5 6
la=[8,4,3,5,7,3,5]
lb=[]

# write here
lb.append(la[0])
lb.append(la[2])
lb.append(la[4])
lb.append(la[6])
print(lb)
```

```
[8, 3, 7, 5]
```

</div>

[20]:

```
#    0 1 2 3 4 5 6
la=[8,4,3,5,7,3,5]
lb=[]

# write here
```

```
[8, 3, 7, 5]
```

5.8.4 `extend` method

We've seen that with `append` we can augment a list *one element at a time.*

What if we wanted to add in a single shot many elements, maybe taken from another list?

We should use the method `extend`, which MODIFIES the list on which it is called by adding all the elements found in the input sequence.

[21]:
```
la = [78,60,59]
```

[22]:
```
lb = [68,97,67,98]
```

[23]:
```
la.extend(lb)
```

[24]:
```
la
```

[24]:
```
[78, 60, 59, 68, 97, 67, 98]
```

[25]:
```
lb
```

[25]:
```
[68, 97, 67, 98]
```

In the example above, `extend` is called on the variable `la`, and we passed `lb` as parameter

WARNING: `la` is MODIFIED, but the sequence we passed in round parenthesis is not (`lb` in the example)

QUESTION: the execution of method `extend` returns something? What do you see in the output of cell `la.extend(lb)` ?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `extend`, as all list methods, doesn't return anything. To be more explicit, it returns the object `None`, which is not even printed by Jupyter.

</div>

Let's verify what happened with Python Tutor:

[26]:
```
la = [78,60,59]
lb = [68,97,67,98]
la.extend(lb)

jupman.pytut()
```

[26]:
```
<IPython.core.display.HTML object>
```

QUESTION: Look inside this code. Which will be the values associated to variables `la` ,`lb` and `x` after its execution?

```
la = [34,79,54]
lb = [86,45]
x = la.extend(lb)

print('la is ', la)
print('lb is ', lb)
print('x  is ', x)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: It will print this:

```
la is  [34, 79, 54, 86, 45]
lb is  [86, 45]
x  is  None
```

`la` was MODIFIED by adding all the elements of `lb`.

The call to `extend`, like all list methods, returned the object `None` which was associated to variable `x`. Try to understand well what happend by using Python Tutor.

</div>

Extending with sequences

We said that `extend` can take any generic sequence in the round parenthesis, not only lists. This means we can also try to pass a string. For example:

[27]:
```
la = [78,65,87]

s = "hello"

la.extend(s)
```

[28]:
```
la
```

[28]:
```
[78, 65, 87, 'h', 'e', 'l', 'l', 'o']
```

Since the string is a character sequence, `extend` took each of these elements and added them to `la`

QUESTION: was the value associated to variable `s` modified?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: absolutely impossible, because a) `extend` only modifies the list on which it is called and b) strings are immutable anyway.

</div>

QUESTION: The following code:

```
la = [78,65]
la.extend(68,85,87)
```

1. produces un error (which one?)
2. modifies `la` (how?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1: produces an error, because we have to pass a SINGLE parameter to `extend`, which must be a *sequence*. Here instead we are passing many parameters. An alternative might be to build a list like this:

```
la = [78,65]
la.extend([68,85,87])
```

</div>

QUESTION: If this code is executed, what happens?

```
sa = "hello"
sb = "world"
sa.extend(sb)
```

1. `sa` is modified (how?)
2. we get an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: we obtain an error, because `extend` is an exclusive method of lists. It only belongs to lists because MODIFIES the object on which it is called - since strings are immutable objects, it wouldn't make sense to change them.

</div>

QUESTION: If this code is executed, what happens?

```
la = [1,2,3]
lb = [4,5]
lc = [6,7,8]

la.extend(lb).extend(lc)
```

1. `la` becomes `[1,2,3,4,5,6,7,8]`
2. an error (which one?)
3. `la` becomes `[1,2,3,4,5]` and an error (which one?)

QUESTION: 3: `la` becomes `[1,2,3,4,5]` and right after we get an error, because the call to `la.extend(lb)` MODIFIES `la` to `[1,2,3,4,5]` and RETURN the value `None`. At that point, Python tries to call the method `extend` on the object `None`, but since it is not a list, we get an error (**to convince yourself, verify everything with Python Tutor** !!!)

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-45-0a08a154ada4> in <module>
      3 lc = [6,7,8]
      4
----> 5 la.extend(lb).extend(lc)

AttributeError: 'NoneType' object has no attribute 'extend'
```

Exercise: augmenting a list 2

Given two *lists* `la` and `lb` and an element `x`, write some code to MODIFY `la` so that `la` contains at the end the element `x` followed by all other elements of `lb`

- **NOTE 1**: your code should work with any `la` and `lb`
- **NOTE 2**: `id` is a Python function which associates to each memory region a unique identifier. If you try printing `id(la)` *before* modyfing `la` and `id(la)` afterwards, you should obtain *exactly* the same id. If you obtain a different one, it means you generated an entirely new list. In that case, verify how it's working with Python Tutor.

```
la = [5,9,2,4]
lb = [7,1,3]
x = 8
```

You should obtain:

```
>>> print(la)
[5,9,2,4,8,7,1,3]
>>> print(lb)
[7,1,3]
>>> print(x)
8
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[29]:

```
la = [5,9,2,4]
lb = [7,1,3]
x = 8

# write here
la.append(x)
la.extend(lb)
#print(la)
#print(lb)
#print(x)
```

</div>

[29]:

```
la = [5,9,2,4]
lb = [7,1,3]
x = 8

# write here
```

Exercise - zslice

Write some code which given two lists `la` (of at least 3 elements) and `lb`, MODIFIES `lb` in such a way to add 3 elements of `la` followed by the last 3 of `la`.

- your code must work with any list
- use `extends` and slices

```
la = ['a','b','c','d','e','f','g','h','i','l','m','n','o']
lb = ['z']
```

You should obtain:

```
>>> print(la)
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'l', 'm', 'n', 'o']
>>> print(lb)
['z', 'a', 'b', 'c', 'm', 'n', 'o']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[30]:
```
la = ['a','b','c','d','e','f','g','h','i','l','m','n','o']
lb = ['z']

# write here

lb.extend(la[:3])  # a slice generates a list
lb.extend(la[-3:])

#print(la)
#print(lb)
```

</div>

[30]:
```
la = ['a','b','c','d','e','f','g','h','i','l','m','n','o']
lb = ['z']

# write here
```

Exercise - Zebarerun

Write some code which given a list of three strings `words` and an empty list `la`, fills `la` with all the first 3 characters of every string in `words`.

- your code must work with any list of 3 strings
- use slices

Example given:

```
words = ["Zebras", "are", "running"]
la = []
```

Your code must show:

```
>>> print(t)
['Z', 'e', 'b', 'a', 'r', 'e', 'r', 'u', 'n']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[31]:
```
words = ["Zebras", "are", "running"]

la = []

# write here
la.extend(words[0][:3])
la.extend(words[1][:3])
la.extend(words[2][:3])
 #print(la)
```

</div>

[31]:
```python
words = ["Zebras", "are", "running"]

la = []

# write here
```

5.8.5 `join` - build strings from lists

Given a string to use as separator, and a sequence like for example a list `la` which only contains strings, it's possible to concatenate them into a (new) string with `join` method:

[32]:
```python
la = ["When", "the", "sun", "raises"]

'SEPARATOR'.join(la)
```

[32]:
```
'WhenSEPARATORtheSEPARATORsunSEPARATORraises'
```

As separator we can put any character, like a space:

[33]:
```python
' '.join(la)
```

[33]:
```
'When the sun raises'
```

Note the original list is not modified:

[34]:
```python
la
```

[34]:
```
['When', 'the', 'sun', 'raises']
```

QUESTION: What does this code produce?

```python
''.join(['a','b','c']).upper()
```

1. an error (which one?)
2. a string (which one?)
3. a list (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: it produces the string `'ABC'`: first it takes all characters from the list `['a','b','c']` and it joins them with empty space `''` separator to form `'abc'`, then this string is set all uppercase with `upper()`.

</div>

QUESTION: What does this code produce?

```python
'a'.join('KRT') + 'E'
```

1. a string (which one?)
2. an error (which one?)
3. a list (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1: produces the string `'KaRaTE'` - we said that `join` takes as input a sequence, so we are not bounded to pass lists but we can directly pass any string, which is a character sequence. `join` will then interval each character in the string with the separator we provide before the dot.

</div>

QUESTION: What does this code produce?

```
'\''.join('mmmm')
```

1. an error (which one?)
2. a string (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: `\'` is an escape sequence which represents the single character apex `'`, so we will obtain `m'm'm'm`

</div>

QUESTION: Given any string `s` and a list of strings `la` of at least two characters, the following code will always give us the same result - which one? (think about it, and if you don't know how to answer try putting random values for `s` and `la`)

```
len(s) <= len(s.join(la))
```

1. an errore (which one?)
2. a stringa (which one?)
3. something else (what?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3: the code will always produce the boolean `True` because `s.join(la)` produces a string containing all the strings in `la` alternated with the string `s`. So the length of this string will always be greater or equal to the length of `s`: by comparing the two lengths with <= operator, we will always obtain the boolean `True`.

Example:

```
s = "ab"
la = ['uief','cb','sd']
len(s) <= len(s.join(la))
```

</div>

Exercise - dub dab dib dob

Write some code which given a list of strings `la`, associates to variable `s` a string with the concatenated strings, separating them with a comma and a space.

Example - given:

```
la = ['dub', 'dab','dib', 'dob']
```

After your code, you should obtain this:

```
>>> print(s)
dub, dab, dib, dob
>>> len(s)
18
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[35]:
```python
la = ['dub', 'dab','dib', 'dob']

# write here

s = ', '.join(la)

#print(s)
#len(s)
```

</div>

[35]:
```python
la = ['dub', 'dab','dib', 'dob']

# write here
```

Exercise - ghirigori

Given a list of strings `la` and a list with three separators `seps`, write some code which prints the elements of `la` separated by first separator, followed by the second separator, followed by the elements of `la` separated by the third separator.

- your code must work with any list `la` and `seps`

Example - given:

```python
la = ['ghi','ri','go','ri']
seps = [',','_','+']
```

After your code, it must print:

```
ghi,ri,go,ri_ghi+ri+go+ri
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[36]:
```python
la = ['ghi','ri','go','ri']
seps = [',','_','+']

# write here

#print(seps[0].join(la) + seps[1] + seps[2].join(la))
```

</div>

[36]:
```python
la = ['ghi','ri','go','ri']
seps = [',','_','+']
```

(continues on next page)

(continued from previous page)

```
# write here
```

Exercise - welldone

Given the list

```
la = ["walnut", "eggplant", "lemon", "lime", "date", "onion", "nectarine", "endive" ]:
```

1. Create another list (call it `new`) containing the first character of every element of `la`
2. Add a space to `new` at position 4 and attach an exclamation mark (`'!'`) at the end
3. Print the list
4. Print the list content by joining all elements with an empty space

You should get:

```
['w', 'e', 'l', 'l', ' ', 'd', 'o', 'n', 'e', '!']

 well done!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[37]:
```
la = ["walnut", "eggplant", "lemon", "lime", "date", "onion", "nectarine", "endive" ]

# write here

new = []
new.append(la[0][0])
new.append(la[1][0])
new.append(la[2][0])
new.append(la[3][0])
new.append(la[4][0])
new.append(la[5][0])
new.append(la[6][0])
new.append(la[7][0])

new.insert(4," ")
new.append("!")

print(new)
print("\n", "".join(new))
```

```
['w', 'e', 'l', 'l', ' ', 'd', 'o', 'n', 'e', '!']

 well done!
```

</div>

[37]:
```
la = ["walnut", "eggplant", "lemon", "lime", "date", "onion", "nectarine", "endive" ]

# write here
```

(continues on next page)

(continued from previous page)

```
['w', 'e', 'l', 'l', ' ', 'd', 'o', 'n', 'e', '!']

 well done!
```

5.8.6 `insert` method

`insert` MODIFIES the list by inserting an element at a specific index - all elements starting from that index will be shifted of one position to the right.

```
[38]:     #0 1 2 3
      la = [6,7,8,9]
```

```
[39]: la.insert(2,55)  # insert the number 55 at index 2
```

```
[40]: la
```

```
[40]: [6, 7, 55, 8, 9]
```

```
[41]: la.insert(0,77)  # insert the number 77 at index 0
```

```
[42]: la
```

```
[42]: [77, 6, 7, 55, 8, 9]
```

We can insert after the end:

```
[43]: la.insert(6,88)  # insert the number 88 at index 6
```

```
[44]: la
```

```
[44]: [77, 6, 7, 55, 8, 9, 88]
```

Note that if we go beyond the end, the element is placed right after the end and no empty cells are created:

```
[45]: la.insert(1000,99)  # insert number 99 at index 7
```

```
[46]: la
```

```
[46]: [77, 6, 7, 55, 8, 9, 88, 99]
```

QUESTION: Given any list `x`, what does this code produce? Can we rewrite it in some other way?

```
x.insert(len(x),66)
```

1. produces a new list (which one?)
2. modifies `x` (how?)
3. an error

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2 - the code MODIFIES the list `x` by adding the element `66` at the end. The code is then equivalent to code

```
x.append(66)
```

</div>

QUESTION: What does the following code produce?

```
la = [3,4,5,6]
la.insert(0,[1,2])
print(la)
```

1. prints `[1,2,3,4,5,6]`
2. an error (which one?)
3. something else (what?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3 - the code inserts in `la` the list `[1,2]` as zero-th element. The print will then show `[[1, 2], 3, 4, 5, 6]`

</div>

QUESTION: What does the following code produce?

```
la = [4,5,6]
la.insert(0,1,2,3)
print(la)
```

1. prints `[1,2,3,4,5,6]`
2. an error (which one?)
3. something else (what?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2 - an error, because we can only pass 2 parameters to `insert`, the insertion index and the single object to insert.

</div>

QUESTION: What does the following code produce?

```
la = [4,5,6]
lb = la.insert(0,3)
lc = lb.insert(0,2)
ld = lc.insert(0,1)
print(ld)
```

1. prints `[1,2,3,4,5,6]`
2. an error (which one?)
3. something else (what?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2 - an error: like almost all list methods, `insert` returns `None`, so by writing `lb = la.insert(0,3)` we are associating `None` to `lb`, so when Python in the next line encounters `lc = lb.insert(0,2)` and tries to execute `None.insert(0,2)` it will complain because `None` not being a list doesn't have the `insert` method.

</div>

Exercise - insertando

Given the list

```
la = [7,6,8,5,6]
```

write some code which MODIFIES the list by using only calls to `insert`. After your code, `la` should appear like this:

```
>>> print(la)
[7, 77, 99, 6, 8, 88, 5, 6, 55]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[47]:

```
la = [7,6,8,5,6]

# write here

la.insert(3,88)
la.insert(1,99)
la.insert(1,77)
la.insert(len(la),55)

#print(la)
```

</div>

[47]:

```
la = [7,6,8,5,6]

# write here
```

> **WARNING**: calling `insert` is much slower than `append` !!
>
> A call to `insert` rewrites all the cells after the insertion point, while `append` instead adds only one cell. Given the computer is fast, very often we do not realize the difference, but whenever possible, and especially if you have to write programs which operate on big amounts of data, try writing code using `append` instead of `insert`.

Exercise - barzoletta

Given the string

```
sa = 'barzoletta'
```

write some code which creates a NEW string `sb` by changing the original string in such a way it results:

```
>>> print(sb)
'barzelletta'
```

- **USE** the method `insert` and cell reassignment
- **NOTE**: you cannot use them an a string, because it is IMMUTABLE - you will then first convert the string to a list

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[48]:
```
sa = 'barzoletta'

# write here

la = list(sa)
la[4] = 'e'
la.insert(5,'l')
sb = ''.join(la)
#print(sb)
```

</div>

[48]:
```
sa = 'barzoletta'

# write here
```

Exercise - insappend

This code takes as input an empty list `la` and a list of numbers `lb`. Try to understand what it does, and rewrite it using some `append`.

[49]:
```
la = []
lb = [7,6,9,8]
la.insert(0,lb[0]*2)
la.insert(0,lb[1]*2)
la.insert(0,lb[2]*2)
la.insert(0,lb[3]*2)
print(la)
```

```
[16, 18, 12, 14]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```
la = []
lb = [7,6,9,8]

# write here
la.append(lb[-1]*2)
la.append(lb[-2]*2)
la.append(lb[-3]*2)
la.append(lb[-4]*2)
#print(la)
```

</div>

[50]:
```
la = []
lb = [7,6,9,8]
```

(continues on next page)

(continued from previous page)

```
# write here
```

5.8.7 `remove` method

`remove` takes an object as parameter, searches for the FIRST cell containing that object and eliminates it:

```
[51]: #      0 1 2 3 4 5
      la = [6,7,9,5,9,8]    # the 9 is in the first cell with index 2 and 4
```

```
[52]: la.remove(9)    # searches first cell containing 9
```

```
[53]: la
```

```
[53]: [6, 7, 5, 9, 8]
```

As you can see, the cell which was at index 2 and that contained the FIRST occurrence of `9` has been eliminated. The cell containing the SECOND occurrence of `9` is still there.

If you try removing an object which is not present, you will receive an error:

```
la.remove(666)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-121-5d04a71f9d33> in <module>
----> 1 la.remove(666)

ValueError: list.remove(x): x not in list
```

Exercise - nob

Write some code which removes from list `la` all the numbers contained in the 3 elements list `lb`.

- your code must work with any list `la` and `lb` of three elements
- you can assume that list `la` contains exactly TWO occurrences of all the elements of `lb` (plus also other numbers)

Example - given:

```
lb = [8,7,4]
la = [7,8,11,8,7,4,5,4]
```

after your code it must result:

```
>>> print(la)
[11, 5]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[54]:
```
lb = [8,7,4]
la = [7,8,11,8,7,4,5,4]

# write here

la.remove(lb[0])
la.remove(lb[0])
la.remove(lb[1])
la.remove(lb[1])
la.remove(lb[2])
la.remove(lb[2])
#print(la)
```

</div>

[54]:
```
lb = [8,7,4]
la = [7,8,11,8,7,4,5,4]

# write here
```

5.8.8 `pop` method

`pop` method does two things: when called without arguments MODIFIES the list by removing the last element, and also RETURNS the removed element:

[55]:
```
basket = ['melon','strawberry', 'apple']
```

[56]:
```
basket.pop()
```

[56]:
```
'apple'
```

[57]:
```
basket
```

[57]:
```
['melon', 'strawberry']
```

[58]:
```
basket.pop()
```

[58]:
```
'strawberry'
```

[59]:
```
basket
```

[59]:
```
['melon']
```

Since the last element is *returned* by `pop`, we can also assign it to a variable:

[60]:
```
fruit = basket.pop()
```

Note we don't see no result printed because the returned element was assigned to the variable `fruit`:

[61]:
```
fruit
```

[61]:
```
'melon'
```

We also notice that `basket` was MODIFIED indeed:

[62]:
```
basket
```

[62]:
```
[]
```

If you further call `pop` on an empty list you will get an error:

```
basket.pop()
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-67-086f38c9fbc0> in <module>()
----> 1 basket.pop()

IndexError: pop from empty list
```

Optionally, to remove an element from a specific position we can pass `pop` an index from `0` INCLUDED to the length of the list EXCLUDED:

[63]:
```
#           0           1              2         3
tools = ['hammer', 'screwdriver', 'plier', 'hammer']
```

[64]:
```
tools.pop(2)
```

[64]:
```
'plier'
```

[65]:
```
tools
```

[65]:
```
['hammer', 'screwdriver', 'hammer']
```

QUESTION: Have a look at following code snippets, and for each of them try to guess the result it produces (or if it gives an error):

1. ```
   la = ['a']
   print(la.pop())
   print(la.pop())
   ```

2. ```
   la = [4,3,2,1]
   print(la.pop(4))
   print(la)
   ```

3. ```
   la = [1,2,3,4]
   print(la.pop(3))
   print(la)
   ```

4. ```
   la = [1,2,3,4]
   print(la.pop(-1))
   print(la)
   ```

5. ```
   s = 'grezzo'
   print(s.pop())
   print(s)
   ```

6. ```
   la = ['molto', 'grezzo']
   print(la.pop())
   print(la)
   ```

7.
```
la = ['a', ['a']]
print(la.pop())
print(la)
```

Exercise - popcorn

Given the list `corn` of exactly 4 characters, write some code which transfers in reverse order all the characters from `corn` to another list `box` which is initially empty.

- **DO NOT** use methods like `reverse` or functions like `reversed`
- Your code must work with *any* list `corn` of 4 elements

Example - given:

```
corn = ['G','u','r','u']
box = []
```

after your code, it must result:

```
>>> print(corn)
[]
>>> print(box)
['u', 'r', 'u', 'G']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[66]:
```
corn = ['G','u','r','u']
box = []

# write here

box.append(corn.pop())
box.append(corn.pop())
box.append(corn.pop())
box.append(corn.pop())
print(box)
```

```
['u', 'r', 'u', 'G']
```

</div>

[66]:
```
corn = ['G','u','r','u']
box = []

# write here
```

```
['u', 'r', 'u', 'G']
```

Exercise - zonzo

Given a list `la` containing some characters, and a list `lb` containing exactly two positions *in ascending order,* write some code which eliminates from `la` the characters at positions specified in `lb`.

- **WARNING**: by calling `pop` the first time you will MODIFY `la`, so the index from the second element to eliminate will need to be properly adjusted !
- **DO NOT** create new lists, so no rows beginning with `la =`
- Your code must work with *any* `la` and *any* `lb` of two elements

Example - given:

```
#       0   1   2   3   4
la = ['z','o','n','z','o']
lb = [2,4]
```

at position 2 in `la` we find the `n` and at 4th the `o`, so after your code it must result:

```
>>> print(la)
['z', 'o', 'z']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[67]: #       0   1   2   3   4
la = ['z','o','n','z','o']
lb = [2,4]

# write here
la.pop(lb[0])
la.pop(lb[1]-1)
#print(la)
```

```
[67]: 'o'
```

</div>

```
[67]: #       0   1   2   3   4
la = ['z','o','n','z','o']
lb = [2,4]

# write here
```

```
[67]: 'o'
```

5.8.9 `reverse` method

`reverse` method MODIFIES the list on which it is called by inverting the order of elements.

Let's see an example:

[68]:
```
la = [7,6,8,4]
```

[69]:
```
la.reverse()
```

[70]:
```
la
```

[70]:
```
[4, 8, 6, 7]
```

> **WARNING**: `reverse` **RETURNS NOTHING!**
>
> To be precise, it returns `None`

[71]:
```
lb = [7,6,8,4]
```

[72]:
```
x = lb.reverse()
```

[73]:
```
print(x)
```

```
None
```

[74]:
```
print(lb)
```

```
[4, 8, 6, 7]
```

QUESTION: Which effect does the following code produce?

```
s = "transatlantic"
s.reverse()
print(s)
```

1. an error (which one?)
2. prints the string in reverse

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `.reverse()` is a method ONLY present in LISTS, so by using it on strings we will get an error. And we have to expect it, as `reverse` MODIFIES the object on which it is called and since strings are *immutable* no string method can possibly modify the string on which it is called.

</div>

QUESTION: If `x` is some list, which effect does the following produce?

```
x.reverse().reverse()
```

1. changes the list (how?)
2. it doesn't change the list
3. generates an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3 - generates an error, because `reverse()` returns `None` which is not a list so it doesn't have `reverse()` method.

</div>

Exercise - good manners

Write some code which given two lists `la` and `lb` MODIFY `la` adding all the elements of `lb` and then reversing the whole list.

- you code must work with any `la` and `lb`
- **DO NOT** modify `lb`

Example - given:

```
la = ['g','o','o','d']
lb = ['m','a','n','n','e','r','s']
```

After your code, it must print:

```
>>> print('la=',la)
la= ['s', 'r', 'e', 'n', 'n', 'a', 'm', 'd', 'o', 'o', 'g']
>>> print('lb=',lb)
lb= ['m', 'a', 'n', 'n', 'e', 'r', 's']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[75]:
```
la = ['g','o','o','d']
lb = ['m','a','n','n','e','r','s']

# write here
la.extend(lb)
la.reverse()
print('la=',la)
print('lb=',lb)
```

```
la= ['s', 'r', 'e', 'n', 'n', 'a', 'm', 'd', 'o', 'o', 'g']
lb= ['m', 'a', 'n', 'n', 'e', 'r', 's']
```

</div>

[75]:
```
la = ['g','o','o','d']
lb = ['m','a','n','n','e','r','s']

# write here
```

```
la= ['s', 'r', 'e', 'n', 'n', 'a', 'm', 'd', 'o', 'o', 'g']
lb= ['m', 'a', 'n', 'n', 'e', 'r', 's']
```

5.8.10 Exercise - precious things

Given two lists `la` and `lb` write some code which PRINTS a list with the elements of `la` and `lb` in reverse order.

- **DO NOT** modify `la` and **DO NOT** modify `lb`
- your code must work with any list `la` and `lb`

Example - given:

```
la = ['p','r','e','c','i','o','u','s']
lb = ['t','h','i','n','g','s']
```

After your code, it must print:

```
['s', 'g', 'n', 'i', 'h', 't', 's', 'u', 'o', 'i', 'c', 'e', 'r', 'p']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[76]:
```
la = ['p','r','e','c','i','o','u','s']
lb = ['t','h','i','n','g','s']

# write here
lc = la + lb   # the + creates a NEW list
lc.reverse()
print(lc)
```

```
['s', 'g', 'n', 'i', 'h', 't', 's', 'u', 'o', 'i', 'c', 'e', 'r', 'p']
```

</div>

[76]:
```
la = ['p','r','e','c','i','o','u','s']
lb = ['t','h','i','n','g','s']

# write here
```

```
['s', 'g', 'n', 'i', 'h', 't', 's', 'u', 'o', 'i', 'c', 'e', 'r', 'p']
```

5.8.11 Exercise - powers

The following code uses some `insert` which as we already said it is not very efficient. Try to understand what it does, and rewrite it using only `append` and `reverse`

- your code must work for any value of `x`

[77]:
```
x = 2
la = [x]
la.insert(0,la[0]*2)
la.insert(0,la[0]*2)
la.insert(0,la[0]*2)
la.insert(0,la[0]*2)
print(la)
```

```
[32, 16, 8, 4, 2]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[78]:
```
x = 2
la = [x]

# write here
la.append(la[-1]*2)
la.append(la[-1]*2)
la.append(la[-1]*2)
la.append(la[-1]*2)
la.reverse()
#print(la)
```

</div>

[78]:
```
x = 2
la = [x]

# write here
```

5.8.12 `sort` method

If a list contains homogenous elements, it is possible to sort it rapidly with the `sort` method, which MODIFIES the list on which it is called (also called sorting *in-place*):

[79]:
```
la = [8,6,7,9]
```

[80]:
```
la.sort()  # NOTE: sort returns nothing !!!
```

[81]:
```
la
```

[81]:
```
[6, 7, 8, 9]
```

Strings are also sortable[215]

[82]:
```
lb = ['Boccaccio', 'Alighieri', 'Manzoni', 'Leopardi']
```

[83]:
```
lb.sort()
```

[84]:
```
lb
```

[84]:
```
['Alighieri', 'Boccaccio', 'Leopardi', 'Manzoni']
```

A list with non-comparable elements it's not sortable, and Python will complain:

[96]:
```
lc = [3,4,'cabbage',7,'potatoes']
```

```
>>> lc.sort()

---------------------------------------------------------------------------
```

(continues on next page)

[215] https://en.softpython.org/strings/strings2-sol.html#Comparing-characters

(continued from previous page)

```
TypeError                                 Traceback (most recent call last)
<ipython-input-288-0cabfae30939> in <module>
----> 1 lc.sort()

TypeError: '<' not supported between instances of 'str' and 'int'
```

Optionally, for reverse order you can pass the parameter `reverse=True` :

[98]:
```
la = [4,2,5,3]
la.sort(reverse=True)
```

[100]:
```
la
```

[100]:
```
[5, 4, 3, 2]
```

Custom sorting

If you have custom needs like for example a lists of strings in the format `'name surname'` that you want to sort according to the surname, you should use optional parameter `key` with `lambda` functions, see Python docs[216]

Exercise - manylines

Given the following string of text:

```
"""This is a string
of text on
several lines which tells nothing."""
```

1. print it
2. prints how many lines, words and characters it contains
3. sort the words in alphabetical order and print the first and last ones in lexicographical order

You should obtain:

```
This is a string
of text on
several lines which tells nothing.

Lines: 3   words: 12   chars: 62

['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\n',
↪ 'o', 'f', ' ', 't', 'e', 'x', 't', ' ', 'o', 'n', '\n', 's', 'e', 'v', 'e', 'r', 'a
↪', 'l', ' ', 'l', 'i', 'n', 'e', 's', ' ', 'w', 'h', 'i', 'c', 'h', ' ', 't', 'e',
↪'l', 'l', 's', ' ', 'n', 'o', 't', 'h', 'i', 'n', 'g', '.']
62

First word:  This
Last word: which
['This', 'a', 'is', 'lines', 'nothing.', 'of', 'on', 'several', 'string', 'tells',
↪'text', 'which']
```

[216] https://docs.python.org/3/howto/sorting.html#key-functions

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[101]:
```
s = """This is a string
of text on
several lines which tells nothing."""

# write here

# 1) print
print(s)
print("")

# 2) prints the lines, words and characters
lines = s.split('\n')

# NOTE: words are separated by a space or a newline

words = lines[0].split(' ') + lines[1].split(' ') + lines[2].split(' ')
num_chars = len(s)
print("Lines:", len(lines), "  words:", len(words), "  chars:", num_chars)

# alternative method for number of characters
print("")
characters = list(s)
num_chars2 = len(characters)
print(characters)
print(num_chars2)

# 3. alphabetically order the words and prints the first and last one in␣
↪lexicographical order

words.sort() # NOTE: it returns NOTHING !!!!
print("")
print("First word: ", words[0])
print("Last word:", words[-1])
print(words)
```

```
This is a string
of text on
several lines which tells nothing.

Lines: 3   words: 12   chars: 62

['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\n',
↪ 'o', 'f', ' ', 't', 'e', 'x', 't', ' ', 'o', 'n', '\n', 's', 'e', 'v', 'e', 'r', 'a
↪', 'l', ' ', 'l', 'i', 'n', 'e', 's', ' ', 'w', 'h', 'i', 'c', 'h', ' ', 't', 'e',
↪'l', 'l', 's', ' ', 'n', 'o', 't', 'h', 'i', 'n', 'g', '.']
62

First word:  This
Last word: which
['This', 'a', 'is', 'lines', 'nothing.', 'of', 'on', 'several', 'string', 'tells',
↪'text', 'which']
```

</div>

[101]:
```
s = """This is a string
```

(continues on next page)

(continued from previous page)

```
of text on
several lines which tells nothing."""

# write here

```

```
This is a string
of text on
several lines which tells nothing.

Lines: 3   words: 12   chars: 62

['T', 'h', 'i', 's', ' ', 'i', 's', ' ', 'a', ' ', 's', 't', 'r', 'i', 'n', 'g', '\n',
↪ 'o', 'f', ' ', 't', 'e', 'x', 't', ' ', 'o', 'n', '\n', 's', 'e', 'v', 'e', 'r', 'a
↪', 'l', ' ', 'l', 'i', 'n', 'e', 's', ' ', 'w', 'h', 'i', 'c', 'h', ' ', 't', 'e',
↪'l', 'l', 's', ' ', 'n', 'o', 't', 'h', 'i', 'n', 'g', '.']
62

First word:  This
Last word: which
['This', 'a', 'is', 'lines', 'nothing.', 'of', 'on', 'several', 'string', 'tells',
↪'text', 'which']
```

Exercise - numlist

Given the list `la = [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]`

1. finds the `min`, `max` and the `median value` (HINT: sort it and extract the right values)
2. create a list only with elements at even indexes (i.e. [10, 72, 11, ..], note that ".." means the list is still not complete!) and ricalculates the values of min, max and median
3. redo the same with the elements at odd indexes (i.e. [60, 118,..])

You should obtain:

```
original:     [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]
even:         [10, 72, 11, 56, 120]
odd:          [60, 118, 71, 89, 175]

sorted:       [10, 11, 56, 60, 71, 72, 89, 118, 120, 175]
sorted even: [10, 11, 56, 72, 120]
sorted odd:  [60, 71, 89, 118, 175]

original:     Min: 10  Max. 175  Median:  72
even:         Min: 10  Max. 120  Median:  56
odd:          Min: 60  Max. 175  Median:  89
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[87]:
```
la = [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]

# write here

even = la[0::2]   # we take only elements at even indeces
```

(continues on next page)

(continued from previous page)

```
odd = la[1::2]     # we take only elements at odd indeces

print("original:    " , la)
print("even:        ", even)
print("odd:         ", odd)

la.sort()
even.sort()
odd.sort()

print()
print("sorted:      " , la)
print("sorted even:" , even)
print("sorted odd:  " , odd)
print()
print("original:     Min:", la[0], " Max." , la[-1], " Median: ", la[len(la) // 2])
print("even:         Min:", even[0], " Max." , even[-1], " Median: ", even[len(even) //
↪ 2])
print("odd:          Min:", odd[0], " Max." , odd[-1], " Median: ", odd[len(odd) // 2])
```

```
original:    [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]
even:        [10, 72, 11, 56, 120]
odd:         [60, 118, 71, 89, 175]

sorted:      [10, 11, 56, 60, 71, 72, 89, 118, 120, 175]
sorted even: [10, 11, 56, 72, 120]
sorted odd:  [60, 71, 89, 118, 175]

original:    Min: 10  Max. 175  Median:  72
even:        Min: 10  Max. 120  Median:  56
odd:         Min: 60  Max. 175  Median:  89
```

</div>

[87]:
```
la = [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]

# write here
```

```
original:    [10, 60, 72, 118, 11, 71, 56, 89, 120, 175]
even:        [10, 72, 11, 56, 120]
odd:         [60, 118, 71, 89, 175]

sorted:      [10, 11, 56, 60, 71, 72, 89, 118, 120, 175]
sorted even: [10, 11, 56, 72, 120]
sorted odd:  [60, 71, 89, 118, 175]

original:    Min: 10  Max. 175  Median:  72
even:        Min: 10  Max. 120  Median:  56
odd:         Min: 60  Max. 175  Median:  89
```

5.8.13 `index` method

The `index` method allows us to find the index of the FIRST occurrence of an element.

```
[88]: #       0    1    2    3    4
      la = ['p','a','e','s','e']
```

```
[89]: la.index('p')
```

```
[89]: 0
```

```
[90]: la.index('a')
```

```
[90]: 1
```

```
[91]: la.index('e')   # we find the FIRST occurrence
```

```
[91]: 2
```

If the element we're looking for is not present, we will get an error:

```
>>> la.index('z')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-303-32d9c064ebe0> in <module>
----> 1 la.index('z')

ValueError: 'z' is not in list
```

5.8.14 `count` method

We can find the number of occurrences of a certain element in a list by using the method `count`

```
[92]: la = ['a', 'n', 'a', 'c', 'o', 'n', 'd', 'a']
```

```
[93]: la.count('n')
```

```
[93]: 2
```

```
[94]: la.count('a')
```

```
[94]: 3
```

```
[95]: la.count('d')
```

```
[95]: 1
```

5.8.15 Continue

You can find more exercises in the worksheet Lists 4[217]

```
[ ]:
```

5.9 Tuple

5.9.1 Download exercise zip

Browse files online[218]

A tuple in Python is an *immutable* sequence of heterogenous elements which allows duplicates, so we can put inside the objects we want, of different types, and with repetitions.

5.9.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
tuples
    tuples.ipynb
    tuples-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `tuples.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** which will ask to write Python commands in the following cells. Exercises are graded by difficulty, from one star ✪ to four ✪✪✪✪

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[217] https://en.softpython.org/lists/lists4-sol.html
[218] https://github.com/DavidLeoni/softpython-en/tree/master/tuples

5.9.3 Creating tuples

Tuples are created with round parenthesis `()` and by separating the elements with commas `,`

Some example:

[2]:
```
numbers = (6,7,5,7,7,9)
```

[3]:
```
print(numbers)
```

```
(6, 7, 5, 7, 7, 9)
```

Tuples of one element: You can create a tuple of a single element **by adding a comma after the element**:

[4]:
```
little_tup = (4,)  # notice the comma !!!
```

Let's verify the type is the expected one:

[5]:
```
type(little_tup)
```

[5]:
```
tuple
```

To see the difference, we write down here `(4)` without comma and we verify the type of the obtained object:

[6]:
```
fake = (4)
```

[7]:
```
type(fake)
```

[7]:
```
int
```

We see that `fake` is an `int`, because `4` has been evaluated as an expression inside round brackets so the result is the content inside the parenthesis.

Empty tuple

We can also create an empty tuple:

[8]:
```
empty = ()
```

[9]:
```
print(empty)
```

```
()
```

[10]:
```
type(empty)
```

[10]:
```
tuple
```

Tuples without brackets

When we assign values to some variable, (and *only* when we assign values to variables) it is possible to use a notation like the following, in which on the left of = we put names of variables and on the right we place a sequence of values:

[11]:
```python
a,b,c = 1, 2, 3
```

[12]:
```python
a
```

[12]:
```
1
```

[13]:
```python
b
```

[13]:
```
2
```

[14]:
```python
c
```

[14]:
```
3
```

If we ask ourselves what that `1,2,3` is, we can try putting on the left a single variable:

[15]:
```python
# WARNING: BETTER AVOID THIS!
x = 1,2,3
```

[16]:
```python
type(x)
```

[16]:
```
tuple
```

We see that Python considered that `1,2,3` as a tuple. Typically, you would never write assignments with less variables than values to put, but if it happens, probably you will find yourself with some undesired tuple !

QUESTION: Have a look at the following code snippets, and for each try guessing which result it produces (or if it gives an error)

1. ```python
   z,w = 5,6
   print(type(z))
   print(type(w))
   ```

2. ```python
   a,b = 5,6
   a,b = b,a
   print('a=',a)
   print('b=',b)
   ```

3. ```python
   z = 5,
   print(type(z))
   ```

4. ```python
   z = ,
   print(type(z))
   ```

Heterogenous elements

In a tuple we can put elements of different types, like numbers and strings:

[17]:
```
stuff = (4, "paper", 5, 2,"scissors", 7)
```

[18]:
```
stuff
```

[18]:
```
(4, 'paper', 5, 2, 'scissors', 7)
```

[19]:
```
type(stuff)
```

[19]:
```
tuple
```

We can also insert other tuples:

[20]:
```
salad = ( ("lettuce", 3), ("tomatoes",9), ("carrots",4) )
```

[21]:
```
salad
```

[21]:
```
(('lettuce', 3), ('tomatoes', 9), ('carrots', 4))
```

[22]:
```
type(salad)
```

[22]:
```
tuple
```

And also lists:

[23]:
```
mix = ( ["when", "it", "rains"], ["I", "program"], [7,3,9] )
```

> **WARNING: avoid mutable objects inside tuples!**
>
> Inserting *mutable* objects like lists inside tuples may cause problems in some situations like when you later want to use the tuple as element of a set or a key in a dictionary (we will see the details in the respective tutorials)

Let's see how the previous examples are represented in Python Tutor:

[24]:
```
# WARNING: before using the function jupman.pytut() which follows,
#          it is necessary to first execute this cell with Shift+Enter (once is␣
↪enough)

import jupman
```

[25]:
```
stuff = (4, "paper", 5, 2,"scissors", 7)
salad = ( ("lettuce", 3), ("tomatoes",9), ("carrots",4) )
mix = ( ["when", "it", "rains"], ["I", "program"], [7,3,9] )

jupman.pytut()
```

[25]:
```
<IPython.core.display.HTML object>
```

Creating tuples from sequences

You can create a tuple from any sequence, like for example a list:

```
[26]: tuple( [8,2,5] )
```

```
[26]: (8, 2, 5)
```

Or a string (which is a character sequence):

```
[27]: tuple("abc")
```

```
[27]: ('a', 'b', 'c')
```

Creating sequences from tuples

Since the tuple is a sequence, it is also possible to generate lists from tuples:

```
[28]: list( (3,4,2,3)  )
```

```
[28]: [3, 4, 2, 3]
```

QUESTION: Does is it make sense creating a tuple from another tuple like this? Can we rewrite the code in a more concise way?

```
[29]: x = (4,2,5)
      y = tuple(x)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: since a tuple is IMMUTABLE, once we create in memory the object `(4,2,5)` we are sure nobody will modify it, so it's not necessary to create a new tuple and we can directly write:

```
x = (4,2,5)
y = x
```

</div>

QUESTION: Have a look at the following expressions, and for each try to guess which result produces (or if it gives an error):

1. `(1.2,3.4)`
2. `(1;2;3;4)`
3. `(1,2;3,4)`
4. `(1,2,3,4)`
5. `(())`
6. `type(())`
7. `((),)`

8. `tuple([('a'),('b'),('c')])`

9. `tuple(tuple(('z','u','m')))`

10. `str(('a','b','c'))`

11. `"".join(('a','b','c'))`

5.9.4 Operators

The following operators work on tuples and behave exactly as in lists:

Operator	Result	Meaning
`len`(tuple)	`int`	Return the length of a tuple
tuple`[`int`]`	`object`	Reads an element at specified index
tuple`[`int`:`int`]`	`tuple`	Extracts a sub-tuple - return a NEW tuple
tuple + tuple	`tuple`	Concatenates two tuples - return a NEW tuple
obj `in` tuple	`bool`	Checks whether an element is present in a tuple
tuple `*` int	`tuple`	Replicates the tuple - return a NEW tuple
`==`,`!=`	`bool`	Checks if two tuples are equal or different

len

`len` function returns the tuple length:

```
[30]: len( (4,2,3) )
```

```
[30]: 3
```

```
[31]: len( (7,) )
```

```
[31]: 1
```

```
[32]: len( () )
```

```
[32]: 0
```

QUESTION: Have a look at following expressions, and for each try to guess the result (or if it gives an error)

1. `len(3,2,4)`

2. `len((3,2,4))`

3. `len(('a',))`

4. `len(('a,'))`

5. `len(((),(),()))`

6. `len(len((1,2,3,4)))`

7. `len([('d','a','c','d'),(('ab')),[('a','b','c')]])`

```
[ ]:
```

Reading an element

Like in strings and lists by using brackets we can read an element at a certain position:

```
[33]: #       0  1  2  3
      tup = (10,11,12,13)
```

```
[34]: tup[0]
```

```
[34]: 10
```

```
[35]: tup[1]
```

```
[35]: 11
```

```
[36]: tup[2]
```

```
[36]: 12
```

```
[37]: tup[3]
```

```
[37]: 13
```

We can also use negative indexes:

```
[38]: tup[-1]
```

```
[38]: 13
```

QUESTION: Have a look at the following expressions and for each of them try to guess the result or if it produces an error:

1. `(1,2,3)[0]`

2. `(1,2,3)[3]`

3. `(1,2,3)0`

4. `()[0]`

5. `(())[0]`

6. `type((())[0])`

7. `('a,')[0]`

8. `('a',)[0]`

9. `(1,2,3)[-0]`

10. `(1,2,3)[-1]`

11. `(1,2,3)[-3]`

```
[ ]:
```

Exercise - animals

Given the string `animals = "Siamese cat,dog,canary,piglet,rabbit,hamster"`

1. convert it to a list
2. create a tuple of tuples where each tuple has two elements: the animal name and the name length, i.e. ((“dog”,3), (“hamster”,7))
3. print the tuple

You should obtain:

```
Siamese cat,dog,canary,piglet,rabbit,hamster

(('Siamese cat', 11), ('dog', 3), ('canary', 6), ('piglet', 6), ('rabbit', 6), (
↪'hamster', 7))
```

- you can assume `animals` always contains exactly 6 animals

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[39]: animals = "Siamese cat,dog,canary,piglet,rabbit,hamster"

# write here
my_list = animals.split(',')

#print(animals)
print()

animals_tuple = ( (my_list[0], len(my_list[0])),
                  (my_list[1], len(my_list[1])),
                  (my_list[2], len(my_list[2])),
                  (my_list[3], len(my_list[3])),
                  (my_list[4], len(my_list[4])),
                  (my_list[5], len(my_list[5])))

#print(animals_tuple)
```

</div>

```
[39]: animals = "Siamese cat,dog,canary,piglet,rabbit,hamster"
```

(continues on next page)

(continued from previous page)

```
# write here
```

Slices

As with strings and lists, by using *slices* we can also extract subsequences from a tuple, that is, on the right of the tuple we can write square brackets with inside a start index INCLUDED, a colon `:` and an end index EXCLUDED:

```
[40]: tup = (10,11,12,13,14,15,16,17,18,19)
```

```
[41]: tup[2:6]   # from index 2 INCLUDED to 6 EXCLUDED
```

```
[41]: (12, 13, 14, 15)
```

It is possible to alternate the gathering of elements by adding the number of elements to skip as a third numerical parameter in the square brackets, for example:

```
[42]: tup = (10,11,12,13,14,15,16,17)
```

```
[43]: tup[0:8:5]
```

```
[43]: (10, 15)
```

```
[44]: tup[0:8:2]
```

```
[44]: (10, 12, 14, 16)
```

```
[45]: tup[1:8:1]
```

```
[45]: (11, 12, 13, 14, 15, 16, 17)
```

WARNING: remeber that slices produce a NEW tuple !

QUESTION: Have a look at the following code snippets, and for each try to guess which result it produces (or if it gives an error)

1. `(7,6,8,9,5)(1:3)`
2. `(7,6,8,9,5)[1:3]`
3. `(10,11,12,13,14,15,16)[3:100]`
4. `(10,11,12,13,14,15,16)[-3:5]`
5. `(1,0,1,0,1,0)[::2]`
6. `(1,2,3)[::1]`

7. `(1,0,1,0,1,0)[1::2]`

8. `tuple("postcards")[0::2]`

9. `(4,5,6,3,4,7)[0:::2]`

Concatenation

It is possible to concatenate two tuples by using the operator +, which creates a NEW tuple:

[46]:
```python
t = (1,2,3) + (4,5,6,7,8)
```

[47]:
```python
t
```

[47]:
```
(1, 2, 3, 4, 5, 6, 7, 8)
```

[48]:
```python
type(t)
```

[48]:
```
tuple
```

Let's verify that original tuples are not modified:

[49]:
```python
x = (1,2,3)
y = (4,5,6,7,8)
```

[50]:
```python
t = x + y
```

[51]:
```python
t
```

[51]:
```
(1, 2, 3, 4, 5, 6, 7, 8)
```

[52]:
```python
x
```

[52]:
```
(1, 2, 3)
```

[53]:
```python
y
```

[53]:
```
(4, 5, 6, 7, 8)
```

Let's see how they are represented in Python Tutor:

[54]:
```python
x = (1,2,3)
y = (4,5,6,7,8)
t = x + y
print(t)
print(x)
print(y)

jupman.pytut()
```

```
(1, 2, 3, 4, 5, 6, 7, 8)
(1, 2, 3)
(4, 5, 6, 7, 8)
```

[54]:
```
<IPython.core.display.HTML object>
```

QUESTION: Have a look at the following code snippets, and for each try to guess which result it produces (or if it gives an error)

1. ```
   ()+()
   ```
2. ```
   type(()+())
   ```
3. ```
   len(()+())
   ```
4. ```
   ()+[]
   ```
5. ```
   []+()
   ```
6. ```
   (2,3,4) + tuple([5,6,7])
   ```
7. ```
   "crazy"+('r','o','c','k','e','t')
   ```

Membership

As in all sequences, if we want to verify whether an element is contained in a tuple we can use the operator `in` which returns a boolean value:

[55]:
```
'e' in ('h','e','l','m','e','t')
```

[55]:
```
True
```

[56]:
```
'z' in ('h','e','l','m','e','t')
```

[56]:
```
False
```

not in

To check whether something is **not** belonging to a tuple, we can use two forms:

not in - form 1:

[57]:
```
"carrot" not in ("watermelon","banana","apple")
```

[57]:
```
True
```

[58]:
```
"watermelon" not in ("watermelon","banana","apple")
```

[58]:
```
False
```

not in - form 2

[59]:
```
not "carrot" in ("watermelon","banana","apple")
```

[59]:
```
True
```

[60]:
```
not "watermelon" in ("watermelon","banana","apple")
```

[60]:
```
False
```
```

QUESTION: Have a look at the following code snippets, and for each try to guess which result it produces (or if it gives an error)

1. ```
   3 in (1.0, 2.0,3.0)
   ```
2. ```
   3.0 in (1,2,3)
   ```
3. ```
   3 not in (3)
   ```
4. ```
   3 not in (3,)
   ```
5. ```
   6 not in ()
   ```
6. ```
   0 in (0)[0]
   ```
7. ```
   [] in ()
   ```
8. ```
   () in []
   ```
9. ```
   not [] in ()
   ```
10. ```
    () in ()
    ```
11. ```
    () in (())
    ```
12. ```
    () in ((),)
    ```
13. ```
    'ciao' in ('c','i','a','o')
    ```

[ ]:

Multiplication

To replicate the elements in a tuple, it is possible to use the operator `*` which produces a NEW tuple:

[61]:
```
(7,8,5) * 3
```

[61]:
```
(7, 8, 5, 7, 8, 5, 7, 8, 5)
```

[62]:
```
(7,8,5) * 1
```

[62]:
```
(7, 8, 5)
```

[63]:
```
(7,8,5) * 0
```

[63]:
```
()
```

QUESTION: What is the following code going to print?

```
x = (5,6,7)
y = x * 3
print('x=',x)
print('y=',y)
```
```

ANWSER: It will print:

```
x = (5, 6, 7)
y = (5, 6, 7, 5, 6, 7, 5, 6, 7)
```

because the multiplication generates a NEW tuple which is associated to `y`. The tuple associated to `x` remains unchanged.

QUESTION: Have a look at the following expressions, and for each try to guess which result it produces (or if it gives an error)

1. ```
   (5,6,7)*(3.0)
   ```
2. ```
   (5,6,7)*(3,0)
   ```
3. ```
   (5,6,7)*(3)
   ```
4. ```
   (5,6,7)*3
   ```
5. ```
   (4,2,3)*int(3.0)
   ```
6. ```
   (1,2)*[3][0]
   ```
7. ```
   (1,2)*(3,4)[-1]
   ```
8. ```
   [(9,8)]*4
   ```
9. ```
   (1+2,3+4)*5
   ```
10. ```
    (1+2,)*4
    ```
11. ```
    (1+2)*4
    ```
12. ```
    (1,2,3)*0
    ```
13. ```
    (7)*0
    ```
14. ```
    (7,)*0
    ```

[ ]:

```

```

Exercise - welcome

Given a tuple `x` containing exactly 3 integers, and a tuple `y` containing exactly 3 tuples of characters, write some code to create a tuple `z` containing each tuple of `y` replicated by the corresponding integer in `x`.

Example - given:

```
x = (2,4,3)
y = (('w','e','l','c'),('o',),('m','e'))
```

after your code it should print:

```
>>> print(z)
('w', 'e', 'l', 'c', 'w', 'e', 'l', 'c', 'o', 'o', 'o', 'o', 'm', 'e', 'm', 'e', 'm',
↪'e')
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[64]:
```python
x = (2,4,3)
y = (('w','e','l','c'),('o',),('m','e'))

# write here
z = y[0]*x[0] + y[1]*x[1] + y[2]*x[2]

print(z)
```

```
('w', 'e', 'l', 'c', 'w', 'e', 'l', 'c', 'o', 'o', 'o', 'o', 'm', 'e', 'm', 'e', 'm',
↪'e')
```

</div>

[64]:
```python
x = (2,4,3)
y = (('w','e','l','c'),('o',),('m','e'))

# write here

```

```
('w', 'e', 'l', 'c', 'w', 'e', 'l', 'c', 'o', 'o', 'o', 'o', 'm', 'e', 'm', 'e', 'm',
↪'e')
```

5.9.5 Write an element

Tuples are *immutable*, so trying to i.e. write an assignment for placing the number `12` into the cell at index `3` provokes an error:

```python
#       0 1 2 3 4
tup = (5,8,7,9,11)
tup[3] = 666

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-118-83949b0c81e2> in <module>
      1 tup = (5,8,7,9,11)
----> 2 tup[3] = 666

TypeError: 'tuple' object does not support item assignment
```

What we can do is to create a NEW tuple by composing it from sequences takes from the original one:

[65]:
```python
#       0   1   2   3   4   5   6
tup = (17,54,34,87,26,95,34)
```

[66]:
```python
tup =  tup[0:3] + (12,) + tup[4:]
```

[67]:
```python
tup
```

[67]:
```
(17, 54, 34, 12, 26, 95, 34)
```

WARNING: `append`, `extend`, `insert`, `sort` **DO NOT WORK WITH TUPLES !**

All the methods you used to modify lists will *not* work with tuples.

Exercise - badmod

Try writing down here `(1,2,3).append(4)` and see which error appears:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[68]:
```
# write here
#(1,2,3).append(4)
```

</div>

[68]:
```
# write here
```

Exercise - abde

Given a tuple `x`, save in a variable `y` another tuple containing:

- at the beginning, the same elements of `x` *except* the last one
- at the end, the elements `'d'` and `'e'` .
- Your code should work with *any* tuple `x`

Example - given:

```
x = ('a','b','c')
```

after your code, you should see printed:

```
x = ('a', 'b', 'c')
y = ('a', 'b', 'd', 'e')
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[69]:
```
x = ('a','b','c')

# write here
y = x[:-1] + ('d','e')

#print('x=',x)
#print('y=',y)
```

</div>

[69]:
```python
x = ('a','b','c')

# write here

```

Exercise - charismatic

Given a tuple `t` having alternating uppercase / lowercase characters, write some code which modifies the assignment of `t` so that `t` becomes equal to a tuple having all characters lowercase as first ones and all uppercase characters as last ones.

Example - given:

```python
t = ('C', 'h','A', 'r', 'I', 's', 'M', 'a', 'T', 'i', 'C')
```

after your code it must result:

```python
>>> print(t)
('C', 'A', 'I', 'M', 'T', 'C', 'h', 'r', 's', 'a', 'i')
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[70]:
```python
t = ('C', 'h','A', 'r', 'I', 's', 'M', 'a', 'T', 'i', 'C')

# write here
t = t[::2] + t[1::2]
#print(t)
```

</div>

[70]:
```python
t = ('C', 'h','A', 'r', 'I', 's', 'M', 'a', 'T', 'i', 'C')

# write here

```

Exercise - sorting

Given a tuple `x` of unordered numbers, write some code which changes the assignment of `x` so that `x` results assigned to a sorted tuple

- your code must work for *any* tuple `x`
- **HINT**: as we've already written, tuples DO NOT have `sort` method (because it would mutate them), but lists have it …

Example - given:

```python
x = (3,4,2,5,5,5,2,3)
```

after your code it must result:

```python
>>> print(x)
(2, 2, 3, 3, 4, 5, 5, 5)
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[71]:
```
x = (3,4,2,5,5,5,2,3)

# write here
y = list(x)
y.sort()
x = tuple(y)
#print(x)
```

</div>

[71]:
```
x = (3,4,2,5,5,5,2,3)

# write here
```

5.9.6 Methods

Tuples are objects of type `typle` and have methods which allows to operate on them:

Method	Return	Description
`tuple.index(obj)`	`int`	Searches for the first occurence of an element and returns its position
`tuple.count(obj)`	`int`	Count the occurrences of an element

`index` method

`index` method allows to find the index of the FIRST occurrence of an element.

[72]:
```
tup = ('b','a','r','a','t','t','o')
```

[73]:
```
tup.index('b')
```

[73]:
```
0
```

[74]:
```
tup.index('a')
```

[74]:
```
1
```

[75]:
```
tup.index('t')
```

[75]:
```
4
```

If the element we’re looking for is not present, we will get an error:

```
>>> tup.index('z')

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-318-96cf33478b69> in <module>
----> 1 tup.index('z')

ValueError: tuple.index(x): x not in tuple
```

QUESTION: Have a look at the following expressions, and for each try to guess which result (or if it gives an error)

1. ```
   (3,4,2).index(4)
   ```
2. ```
   (3,4,---1).index(-1)
   ```
3. ```
   (2.2,.2,2,).index(2)
   ```
4. ```
   (3,4,2).index(len([3,8,2,9]))
   ```
5. ```
   (6,6,6).index(666)
   ```
6. ```
   (4,2,3).index(3).index(3)
   ```
7. ```
   tuple("GUG").index("g")
   ```
8. ```
   (tuple("ci") + ("a","o")).index('a')
   ```
9. ```
   (()).index(())
   ```
10. ```
    ((),).index(())
    ```

`count` method

We can obtain the number of occurrences of a certain element in a list by using the method `count`:

[76]:
```
t = ('a', 'c', 'a', 'd', 'e', 'm', 'i', 'a')
```

[77]:
```
t.count('a')
```

[77]:
```
3
```

[78]:
```
t.count('d')
```

[78]:
```
1
```

If an element is not present `0` is returned:

[79]:
```
t.count('z')
```

[79]:
```
0
```

Exercise - fruits

Given the string `s = "apple|pear|apple|cherry|pear|apple|pear|pear|cherry|pear|strawberry"`

Insert the elements separated by `"|"` (pipe character) in a list.

1. How many elements must the list have?
2. Knowing the list created at previous point has only four distinct elements (es `"apple"`, `"pear"`, `"cherry"`, and `"strawberry"`), create another list where each element is a tuple containing the name of the fruit and its multiplicity (that is, the number of times it appears in the original list).

Example - given:

```
counts = [("apple", 3), ("pear",5), ...]
```

Here you can write code which works given a specific constant, so you don't need cycles.

3. Print the content of each tuple in a separate line (i.e.: first libe; "apple" is present 3 times)

You should obtain:

```
[('apple', 3), ('pear', 5), ('cherry', 2), ('strawberry', 1)]

apple is present 3 times
pear is present 5 times
cherry is present 2 times
strawberry is present 1 times
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[80]:
```python
s = "apple|pear|apple|cherry|pear|apple|pear|pear|cherry|pear|strawberry"

# write here

words = s.split("|")
#print(words)

tapples = ("apple", words.count("apple"))
tpears = ("pear", words.count("pear"))
tcherries = ("cherry", words.count("cherry"))
tstrawberries = ("strawberry", words.count("strawberry"))
counts =[tapples, tpears, tcherries, tstrawberries]

print(counts)
print()
print(tapples[0], "is present", tapples[1], "times")
print(tpears[0], "is present", tpears[1], "times")
print(tcherries[0], "is present", tcherries[1], "times")
print(tstrawberries[0], "is present", tstrawberries[1], "times")
```

```
[('apple', 3), ('pear', 5), ('cherry', 2), ('strawberry', 1)]

apple is present 3 times
pear is present 5 times
cherry is present 2 times
strawberry is present 1 times
```

</div>

[80]:
```python
s = "apple|pear|apple|cherry|pear|apple|pear|pear|cherry|pear|strawberry"

# write here
```

```
[('apple', 3), ('pear', 5), ('cherry', 2), ('strawberry', 1)]

apple is present 3 times
pear is present 5 times
```

(continues on next page)

(continued from previous page)

```
cherry is present 2 times
strawberry is present 1 times
```

5.9.7 Exercises with functions

WARNING: following exercises require to know:

Control flow[219]

Functions[220]

Exercise - touples

⊛⊛ Let's call a *touple* a tuple with a couple of elements. Write a function `touples` which given a tuple, RETURNs a list having as elements *touples* each taken in alternation from `t`.

- if the input tuple `t` has an odd number of elements, the last tuple in the list to return will be made of only one element

Example:

```
>>> touples( ('c', 'a', 'r', 'p', 'e', 't') )    # even length
[('c', 'a'), ('r', 'p'), ('e', 't')]
>>> touples( ('s','p','i','d','e','r','s') )    # odd length
[('s', 'p'), ('i', 'd'), ('e', 'r'), ('s',)]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[81]:
```python
# write here
def touples(t):
    ret = []
    i = 0
    while i < len(t)-1:
        ret.append((t[i],t[i+1]))
        i += 2
    if i == len(t)-1:
        ret.append((t[-1],))
    return ret

#touples( ('c', 'a', 'r', 'p', 'e', 't') )
#touples( ('s','p','i','d','e','r','s') )
```

</div>

[81]:
```python
# write here
```

219 https://en.softpython.org/#control-flow
220 https://en.softpython.org/functions/functions-sol.html

Exercise - joined

⊛⊛ Write a function which given two tuples of characters `ta` and `tb` having each different characters (may also be empty), return a tuple made like this:

- if the tuple `ta` terminates with the same character `tb` begins with, RETURN the concatenation of `ta` and `yb` WITHOUT duplicated characters
- otherwise RETURN an empty tuple

Example:

```
>>> joined(('a','b','c'), ('c','d','e'))
('a', 'b', 'c', 'd', 'e')
>>> joined(('a','b'), ('b','c','d'))
('a', 'b', 'c', 'd')
>>> joined((),('e','f','g'))
()
>>> joined(('a',),('e','f','g'))
()
>>> f(('a','b','c'),())
()
>>> f(('a','b','c'),('d','e'))
()
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[82]:
```python
# write here

def joined(ta,tb):
    if len(ta) > 0 and len(tb) > 0:
        if ta[-1] == tb[0]:
            return ta[:-1] + tb

    return ()

#joined(('a','b','c'), ('c','d','e'))
#joined(('a','b'), ('b','c','d'))
#joined((),('e','f','g'))
#joined(('a',),('e','f','g'))
#f(('a','b','c'),())
#f(('a','b','c'),('d','e'))
```

</div>

[82]:
```python
# write here
```

5.9.8 Verify comprehension

> **WARNING**
>
> The following exercises contain tests with *assert*. To understand how to do them, read first Error handling and testing[221]

doubles

⊕⊕ Take as input a list of n integer numbers, and RETURN a NEW list which contains n tuples each having two elements. Each tuple contains a number taken from the corresponding position in the original list, and its double.

Example - given:

```
doubles([ 5, 3, 8])
```

it must produce the list:

```
[(5,10), (3,6), (8,16)]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[83]:
```python
def doubles(lst):

    ret = []
    for element in lst:
        ret.append((element, element * 2))
    return ret


# TEST START - DO NOT TOUCH !
assert doubles([]) == []
assert doubles([3]) == [(3,6)]
assert doubles([2,7]) == [(2,4),(7,14)]
assert doubles([5,3,8]) == [(5,10), (3,6), (8,16)]

# verify the original list was not changed
la = [6]
lb = doubles(la)
assert la == [6]
assert lb == [(6,12)]
# TEST END
```

</div>

[83]:
```python
def doubles(lst):
    raise Exception('TODO IMPLEMENT ME !')

# TEST START - DO NOT TOUCH !
assert doubles([]) == []
assert doubles([3]) == [(3,6)]
assert doubles([2,7]) == [(2,4),(7,14)]
assert doubles([5,3,8]) == [(5,10), (3,6), (8,16)]
```

(continues on next page)

221 https://en.softpython.org/errors-and-testing/errors-and-testing-sol.html

(continued from previous page)

```
# verify the original list was not changed
la = [6]
lb = doubles(la)
assert la == [6]
assert lb == [(6,12)]
# TEST END
```

nasty

⊛⊛⊛ Given two tuples `ta` and `b`, `ta` made of characters and `tb` of positive integer numbers , write a function `nasty` which RETURNS a tuple having two character strings: the first character is taken from `ta`, the second is a number taken from the corresponding position in `tb`. The strings are repeated for a number of times equal to that number.

```
>>> nasty(('u','r','g'), (4,2,3))
('u4', 'u4', 'u4', 'u4', 'r2', 'r2', 'g3', 'g3', 'g3')

>>> nasty(('g','a','s','p'), (2,4,1,3))
('g2', 'g2', 'a4', 'a4', 'a4', 'a4', 's1', 'p3', 'p3', 'p3')
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[84]:
```
# write here

def nasty(ta, tb):

    i = 0
    ret = []
    while i < len(tb):
        s = ta[i]+str(tb[i])
        ret.extend( (s,) * tb[i] )
        i += 1
    return tuple(ret)


# TEST START - DO NOT TOUCH !
assert nasty(('a',), (3,)) == ('a3','a3','a3')
assert nasty(('a','b'), (3,1)) == ('a3','a3','a3','b1')
assert nasty(('u','r','g'), (4,2,3)) == ('u4', 'u4', 'u4', 'u4', 'r2', 'r2', 'g3', 'g3
↪', 'g3')
assert nasty(('g','a','s','p'), (2,4,1,3)) == ('g2', 'g2', 'a4', 'a4', 'a4', 'a4', 's1
↪', 'p3', 'p3', 'p3')
# TEST END
```

</div>

[84]:
```
# write here
```

5.9.9 References

- Think Python, Chapter 12, Tuples[222]
- W3Resources - tuples[223] : contains many simple exercises

```
[ ]:
```

5.10 Sets

5.10.1 Download exercises zip

Browse online files[224]

A set is a *mutable unordered* collection of *immutable distinct* elements (that is, without duplicates). The Python datatype to represent sets is called `set`.

5.10.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
sets
    sets.ipynb
    sets-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `sets.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[222] http://greenteapress.com/thinkpython2/html/thinkpython2013.html
[223] https://www.w3resource.com/python-exercises/tuple/
[224] https://github.com/DavidLeoni/softpython-en/tree/master/sets

5.10.3 Creating a set

We can create a set using curly brackets, and separating the elements with commas `,`

Let's try a set of characters:

```
[2]: s = {'b','a','d','c'}
```

```
[3]: type(s)
```

```
[3]: set
```

WARNING: SETS ARE *NOT* ORDERED !!!

DO NOT BELIEVE IN WHAT YOU SEE !!

Let's try printing the set:

```
[4]: print(s)
```

```
{'a', 'b', 'c', 'd'}
```

The output shows the order in which the print was made is different from the order in which we built the set. Also, according to the Python version you're using, on your computer it might be even different!

This is because the order in sets is NOT guaranteed: the only thing that matters is whether or not an element belongs to a set.

As a further demonstration, we may ask Jupyter to show the content of the set, by writing only the variable `s` WITHOUT `print`:

```
[5]: s
```

```
[5]: {'a', 'b', 'c', 'd'}
```

Now it appears in alphabetical order! It happens like so because Jupyter show variables by implicitly using the pprint[225] (*pretty* print), which ONLY for sets gives us the courtesy to order the result before printing it. We can thank Jupyter, but let's not allow it to confuse us!

Elements index: since sets have no order, asking Python to extract an element at a given position would make no sense. Thus, differently from strings, lists and tuples, with sets it's NOT possible to extract an element from an index:

```
s[0]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-352-c9c96910e542> in <module>
----> 1 s[0]

TypeError: 'set' object is not subscriptable
```

We said that a set has only *distinct* elements, that is without duplicates - what happens if we try to place some duplicate anyway?

```
[6]: s = {6,7,5,9,5,5,7}
```

[225] https://docs.python.org/3/library/pprint.html

[7]:
```
s
```

[7]:
```
{5, 6, 7, 9}
```

We note that Python silently removed the duplicates.

Converting sequences to sets

As for lists and strings, we can create a `set` from another sequence:

[8]:
```
set('acacia') # from string
```

[8]:
```
{'a', 'c', 'i'}
```

[9]:
```
set( [1,2,3,1,2,1,2,1,3,1] ) # from list
```

[9]:
```
{1, 2, 3}
```

[10]:
```
set( (4,6,1,5,1,4,1,5,4,5) ) # from tuple
```

[10]:
```
{1, 4, 5, 6}
```

Again, we notice in the generated set there are no duplicates

REMEMBER: Sets are useful to remove duplicates from a sequence

Mutable elements and hashes

Let's see again the definition from the beginning:

> A set is a *mutable unordered* collection of *immutable distinct* elements

So far we only created the set using *immutable* elements like numbers and strings.

What happens if we place some mutable elements, like lists?

```
>>> s = { [1,2,3], [4,5] }

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-a6c538692ccb> in <module>
----> 1 s = { [1,2,3], [4,5]  }

TypeError: unhashable type: 'list'
```

We obtain `TypeError: unhashable type: 'list'`, which literally means Python didn't manage to calculate the *hash* of the list. What could this particular dish ever be?

What is the hash? The *hash* of an object is a number that Python can associate to it, for example you can see the `hash` of an object by using the function with the same name:

[11]:
```
hash( "This is a nice day" )  # string
```

[11]:
```
-1276273449930679549
```

```
[12]: hash( 111112222223333333344444445555555555 )    # number
```

```
[12]: 651300278308214397
```

Imagine the *hash* is some kind of label with these properties:

- it is too short to completely describe the object to which it is associated (that is: given a hash label, you *cannot* reconstruct the object it represents)
- it is enough long to identify *almost uniquely* the object…
- … even if in the world there *might* be different objects which have associated exactly the same label

What's the relation with our sets? The *hash* has various applications, but typically Python uses it to quickly find an object in collections which are based on hashes, like sets and dictionaries. How much fast? Very: even with homongous sets, we always obtain an answer in a constant very short time! In other words, the answer speed *does not* depend on the set dimension (except for pathological cases we don't review here).

This velocity is permitted by the fact that given some object to search, Python is able to rapidly calculate its *hash* label: then, with the label in the hand, so to speak, it can manage to quickly find in the memory store whether there are objects which have the same label. If they are found, they will almost surely be very few, so Python will only need to compare them with the searched one.

***Immutable* objects always have the same hash label** from when they are created until the end of the program. Instead, the *mutable* ones behave differently: each time we change an object, the *hash* also changes. Imagine a market where employees place food by looking at labels and separating accordingly for example the coffee in the shelves for the breakfast and bleach in the shelves for detergents. If you are a customer and you want some coffee, you look at signs and directly go toward the shelves for breakfast stuff. Image what could happen if an evil sorcerer could transform the objects already placed into other objects, like for example the coffee into bleach (let's assume that at the moment of the transmutation the *hash* label also changes). Much confusion would certainly follow, and, if we aren't cautious, also a great stomachache or worse.

So to offer you the advantage of a fast search while avoiding disastrous situations, Python imposes to place inside sets only objects with a stable *hash,* that is *immutable* objects.

QUESTION: Can we insert a tuple inside a set? Try to verify your intuition with a code example.

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Yes, tuples are *immutable,* so they have a corrispending *hash* which remains stable for all the program duration, for example this is a tuple set:`{(1,2), (3,4,5)}`

Note we can consider a tuple as really immutable only if it contains elements which are also immutable.

</div>

Empty set

WARNING: If you write `{}` you will obtain a dictionary, NOT a set !!!

To create an empty set we must call the function `set()`:

```
[13]: s = set()
```

```
[14]: s
```

[14]:
```
set()
```

EXERCISE: try writing `{}` in the cell below and look at the object type obtained with `type`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[15]:
```
# write here
```

</div>

[15]:
```
# write here
```

QUESTION: Can we try inserting a set inside another set? Have a careful look at the set definition, then verify your suppositions by writing some code to create a set which has another set inside.

WARNING: To perform the check, DO NOT use the `set` function, only use creation with curly brackets

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: A set is *mutable*, so we *cannot* insert it as an element of another set (its *hash* label could vary over time). By writing `{{1,2,3}}` you will get an error.

</div>

QUESTION: If we write something like this, what do we get? (careful!)

```
set(set(['a','b']))
```

1. a set with `a` and `b` inside
2. a set containing another set which contains `a` and `b` as elements
3. an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 1:

- inside we have the expression `set(['a','b'])` which generates the set `{'a','b'}`
- outside we have the expression `set( set(['a','b']) )` which is given the set just created, so we can rewrite it as `set({'a','b'})`
- Since `set` when used as a function expects a sequence, and a set *is* a sequence, the external `set` takes all the elements it finds inside the sequence `{'a','b'}` we passed, and generates a new set with `'a'` and `'b'` inside.

</div>

QUESTION: Have a look at following expressions, and for each of them try to guess which result it produces (or if it gives an error):

1.
```
{'oh','la','la'}
```

2. `set([3,4,2,3,2,2,2,-1])`

3. `{(1,2),(2,3)}`

4. `set('aba')`

5. `str({'a'})`

6. `{1;2;3}`

7. `set(  1,2,3  )`

8. `set( {1,2,3} )`

9. `set( [1,2,3] )`

10. `set( (1,2,3) )`

11. `set(  "abc"  )`

12. `set(  "1232"  )`

13. `set( [ {1,2,3,2} ] )`

14. `set( [ [1,2,3,2] ] )`

15. `set( [ (1,2,3,2) ] )`

16. `set( [ "abcb"   ] )`

17. `set( [ "1232"   ] )`

18. `set((1,2,3,2))`

19. `set([(),()])`

20. `set([])`

21. `set(list(set()))`

Exercise - dedup

Write some brief code to create a list `lb` which contains all the elements of the list `la` without duplicates and alphabetically sorted.

- DO NOT change original list `la`
- DO NOT use cycles
- your code should work for any `la`

```
la = ['c','a','b','c','d','b','e']
```

After your code, you should obtain:

```
>>> print(la)
['c', 'a', 'b', 'c', 'd', 'b', 'e']
>>> print(lb)
['a', 'b', 'c', 'd', 'e']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[16]:
```
la = ['c','a','b','c','d','b','e']

# write here

lb = list(set(la))
lb.sort()
#lb = list(sorted(set(la)))  # alternative, NOTE sorted generates a NEW sequence

print("la =",la)
print("lb =",lb)
```

```
la = ['c', 'a', 'b', 'c', 'd', 'b', 'e']
lb = ['a', 'b', 'c', 'd', 'e']
```

</div>

[16]:
```
la = ['c','a','b','c','d','b','e']

# write here

```

```
la = ['c', 'a', 'b', 'c', 'd', 'b', 'e']
lb = ['a', 'b', 'c', 'd', 'e']
```

Frozenset

INFO: this topic is optional for the purposes of the book

In Python also exists *immutable* sets which are called `frozenset`. Here we just remind that since frozensets are *immutable* they do have associated a *hash* label and thus they can be inserted as elements of other sets. For other info we refer to the official documentation[226].

[226] https://docs.python.org/3/library/stdtypes.html#frozenset

5.10.4 Operators

Operator	Result	Description
`len`(set)	`int`	the number of elements in the set
el `in` set	`bool`	verifies whether an element is contained in the set
set | set	`set`	union, creates a NEW set
set `&` set	`set`	intersetion, creates a NEW set
set − set	`set`	difference, creates a NEW set
set ^ set	`set`	symmetric difference, creates a NEW set
==, !=	`bool`	checks whether two sets are equal or different

len

```
[17]: len( {'a','b','c'}  )
```

```
[17]: 3
```

```
[18]: len( set() )
```

```
[18]: 0
```

Exercise - distincts

Given a string `word`, write some code that:

- prints the distinct characters present in `word` as alphabetically ordered (without the square brackets!), together with their number
- prints the number of duplicate characters found in total

Example 1 - given:

```
word = "ababbbbcdd"
```

after your code it must print:

```
word      : ababbbbcdd
4 distincts : a,b,c,d
6 duplicates
```

Example 2 - given:

```
word = "cccccaaabbbb"
```

after your code it must print:

```
word      : cccccaaabbbb
3 distinct : a,b,c
9 duplicates
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[19]:
```python
# write here
word = "ababbbbcdd"
#word = "cccccaaabbbb"
s = set(word)
print("word       :", word)
la = list(s)
la.sort()
print(len(s), 'distincts :', ",".join(la))
#print(len(s), 'distincts :', list(sorted(s)))  # ALTERNATIVE WITH SORTED
print(len(word) - len(s), 'duplicates')
```

```
word       : ababbbbcdd
4 distincts : a,b,c,d
6 duplicates
```

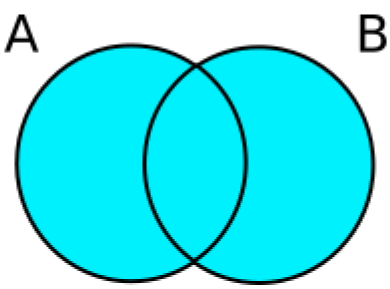
</div>

[19]:
```python
# write here
```

```
word       : ababbbbcdd
4 distincts : a,b,c,d
6 duplicates
```

Union

The union operator `|` (called *pipe*) produces a NEW set containing all the elements from both the first and second set.



[20]:
```python
{'a','b','c'} | {'b','c','d','e'}
```

[20]:
```
{'a', 'b', 'c', 'd', 'e'}
```

Note there aren't duplicated elements

EXERCISE: What if we use the +? Try writing in a cell `{'a','b'} + {'c','d','e'}`. What happens?

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[21]:
```python
# write here
```

</div>

[21]:
```python
# write here
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if they give an error):

1. `{'a','d','b'}|{'a','b','c'}`
2. `{'a'}|{'a'}`
3. `{'a'|'b'}`
4. `{1|2|3}`
5. `{'a'|'b'|'a'}`
6. `{{'a'}|{'b'}|{'a'}}`
7. `[1,2,3] | [3,4]`
8. `(1,2,3) | (3,4)`
9. `"abc" | "cd"`
10. `{'a'} | set(['a','b'])`
11. `set(".".join('pacca'))`
12. `'{a}'|'{b}'|'{a}'`
13. `set((1,2,3))|set([len([4,5])])`
14. `{()}|{()}`
15. `{'|'}|{'|'}`

QUESTION: Given two sets `x` and `y`, the expression

```
len(x | y) <= len(x) + len(y)
```

produces:

1. an error (which one?)
2. always `True`
3. always `False`
4. sometimes `True` sometimes `False` according to values of `x` and `y`

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 2: the number of elements from the union will always be lesser or equal to the sum of the number of elements of each single set we are going to merge, so from the <= comparison we will always get `True`.

</div>

Exercise - everythingbut 1

Write some code which creates a set `s4` which contains all the elements of `s1` and `s2` but does not contain the elements of `s3`.

- Your code should work with *any* set `s1`, `s2`, `s3`

Example - given:

```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])
```

After your code you should obtain:

```
>>> print(s4)
{'d', 'a', 'c', 'g', 'e'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])

# write here
s4 = (s1 | s2) - s3
#print(s4)
```

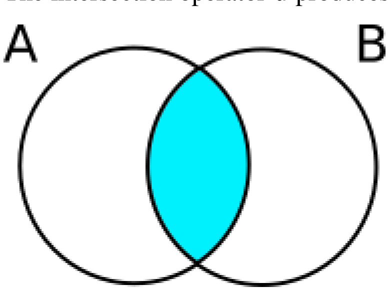
</div>

[22]:
```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])

# write here
```

Intersection

The intersection operator `&` produces a NEW set which contains all the common elements of the first and second set.



[23]:
```
{'a','b','c'} & {'b','c','d','e'}
```

[23]:
```
{'b', 'c'}
```

QUESTION: Look at the following expressions, and for each try guessing wthe result (or if it gives an error):

1. `{0}&{0,1}`

2. `{0,1}&{0}`

3. `set("capra") & set("campa")`

4. `set("cba") & set("dcb")`

5. `{len([1,2,3]),4} & {len([5,6,7])}`

6. `{1,2}&{1,2}`

7. `{0,1}&{}`

8. `{0,1}&set()`

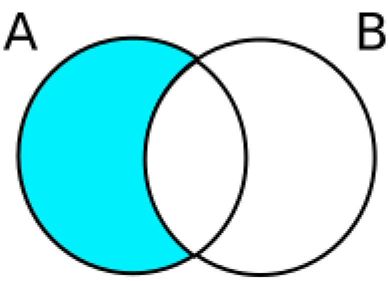
9. `set([1,2,3,4,5][::2]) & set([1,2,3,4,5][2::2])`

10. `{((),)}&{()}`

11. `{(())}&{()}`

Difference

The difference operator `-` produces a NEW set containing all the elements of the first set except the ones from the second:




```
[24]: {'a','b','c','d'} - {'b','c','e','f','g'}
```

```
[24]: {'a', 'd'}
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `{3,4,2}-2`

2. `{1,2,3}-{3,4}`

3. `'{"a"}-{"a"}'`

4. `{1,2,3}--{3,4}`

5. `{1,2,3}-(-{3,4})`
6. `set("chiodo") - set("chiave")`
7. `set("prova") - set("prova".capitalize())`
8. `set("BarbA") - set("BARBA".lower())`
9. `set([(1,2),(3,4),(5,6)]) - set([(2,3),(4,5)])`
10. `set([(1,2),(3,4),(5,6)]) - set([(3,4),(5,6)])`
11. `{1,2,3} - set()`
12. `set() - {1,2,3}`

QUESTION: Given two sets `x` and `y`, what does the following code produce? An error? Is it simplifiable?

```
(x & y) | (x-y)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

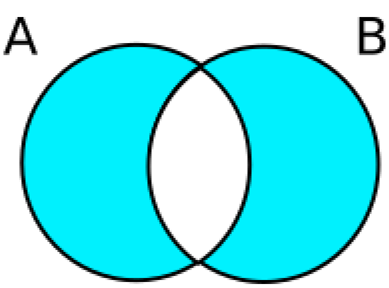
ANSWER: We are merging the common elements between `x` and `y`, with the elements present in `x` but not in `y`. Thus, we are taking all the elements of `x`, so the expression can be greatly simplified by just writing:

```
x
```

</div>

Symmetric difference

The symmetric difference of two sets is their union except their intersection, that is all elements except the common ones:



In Python you can directly express it with the `^` operator:

```
[25]: {'a','b','c'} ^ {'b','c','d','e'}
```

```
[25]: {'a', 'd', 'e'}
```

Let's check the result corresponds to the definition:

```
[26]: s1 = {'a','b','c'}
      s2 = {'b','c','d','e'}

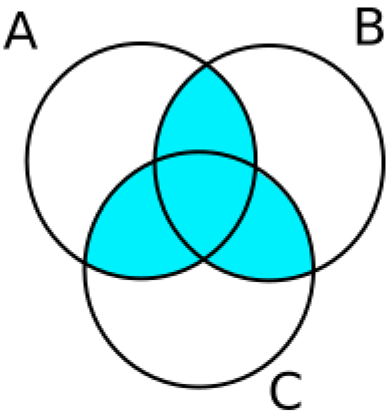
      (s1 | s2) - (s1 & s2)
```

```
[26]: {'a', 'd', 'e'}
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `{'p','e','p','p','o'} ^ {'p','a','p','p','e'}`
2. `{'ab','cd'} ^ {'ba','dc'}`
3. `set('brodino') ^ set('bordo')`
4. `set((1,2,5,3,2,3,1)) ^ set((1,4,3,2))`

QUESTION: given 3 sets A, B, C, what's the expression to obtain the azure part?



Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER:

```
(A & B) | (A & C) | (B & C)
```

</div>

QUESTION: If we use the following values in the previous exercise, what would the set which denotes the azure part contain?

```
A = {'a','ab','ac','abc'}
B = {'b','ab','bc','abc'}
C = {'c','ac','bc','abc'}
```

Once you guessed the result, try executing the formula you obtained in the previous exercise with the provided values and compare the results with the solution.

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: If the formula is correct you should obtain:

```
{'abc', 'ac', 'bc', 'ab'}
```

</div>

Membership

As for any sequence, when we want to check whether an element is contained in a set we can use the `in` operator which returns a boolean value:

```
[27]: 'a' in {'m','e','n','t','a'}
```

```
[27]: True
```

```
[28]: 'z' in {'m','e','n','t','a'}
```

```
[28]: False
```

> `in` **WHEN USED IN SETS IS VERY FAST**
>
> The speed of `in` operator DOES NOT depend on the set dimension

This is a substantial difference with respect to other sequences we've already seen: if you try searching for an element with `in` in strings, lists or tuples, and the element to find is toward the end (or there isn't at all), Python will have to look through the whole sequence.

not in

To check whether something is **not** belonging to a sequence, we can use two forms:

not in - form 1:

```
[29]: "carrot" not in {"watermelon","banana","apple"}
```

```
[29]: True
```

```
[30]: "watermelon" not in {"watermelon","banana","apple"}
```

```
[30]: False
```

not in - forma 2

```
[31]: not "carrot" in {"watermelon","banana","apple"}
```

```
[31]: True
```

```
[32]: not "watermelon" in {"watermelon","banana","apple"}
```

```
[32]: False
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `2*10 in {10,20,30,40}`

2. `'four' in {'f','o','u','r'}`

3. `'aa' in set('aa')`

4. `'a' in set(['a','a'])`

5. `'c' in (set('parco') - set('cassa'))`

6. `'cc' in (set('pacca') & set('zucca'))`

7. `[3 in {3,4}, 6 in {3,4} ]`

8. `4 in set([1,2,3]*4)`

9. `2 in {len('3.4'.split('.'))}`

10. `4 not in {1,2,3}`

11. `'3' not in {1,2,3}`

12. `not 'a' in {'b','c'}`

13. `not {} in set([])`

14. `{not 'a' in {'a'}}`

15. `4 not in set((4,))`

16. `() not in set([()])`

QUESTION: the following expressions are similar. What do they have in common? What is the difference with the last one (beyond the fact it is a set)?

1. `'e' in 'abcde'`

2. `'abcde'.find('e') >= 0`

3. `'abcde'.count('e') > 0`

4. `'e' in ['a','b','c','d','e']`

5. `['a','b','c','d','e'].count('e') > 0`

6. `'e' in ('a','b','c','d','e')`

7. `('a','b','c','d','e').count('e') > 0`

8. `'e' in {'a','b','c','d','e'}`

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: All the expressions reported above return a boolean which is `True` if the element `'e'` is present in the sequence.

All the operations of search and/counting (`in`, `find`, `index`, `count`) on strings, lists and tuples take a search time which in the worst case like here can be equal to the sequence dimension (`'e'` is at the end).

On the other hand, since sets (expression 8.) are based on *hashes,* they allow an immediate search, independently from the set dimension or the elements position (so creating the set with `e` at the end makes no difference).

To make performant searches it's preferable to use hash based collections, like sets or dictionaries !

</div>

5.10.5 Equality

We can check whether two sets are equal by using the equality operator ==, which given two sets return `True` if they contain the same elements or `False` otherwise:

```
[33]: {4,3,6} == {4,3,6}
```

```
[33]: True
```

```
[34]: {4,3,6} == {4,3}
```

```
[34]: False
```

```
[35]: {4,3,6} == {4,3,6, 'hello'}
```

```
[35]: False
```

Careful about removal of duplicates !

```
[36]: {2,8} == {2,2,8}
```

```
[36]: True
```

To verify the inequality, we can use the != operator:

```
[37]: {2,5} != {2,5}
```

```
[37]: False
```

```
[38]: {4,6,0} != {2,8}
```

```
[38]: True
```

```
[39]: {4,6,0} != {4,6,0,2}
```

```
[39]: True
```

Beware of duplicates and order!

```
[40]: {0,1} != {1,0,0,0,0,0,0,0}
```

[40]:
```
False
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. ```
   {2 == 2, 3 == 3}
   ```
2. ```
   {1,2,3,2,1} == {1,1,2,2,3,3}
   ```
3. ```
   {'aa'} == {'a'}
   ```
4. ```
   set('aa') == {'a'}
   ```
5. ```
   [{1,2,3}] == {[1,2,3]}
   ```
6. ```
   set({1,2,3}) == {1,2,3}
   ```
7. ```
   set((1,2,3)) == {(1,2,3)}
   ```
8. ```
   {'aa'} != {'a', 'aa'}
   ```
9. ```
   {set() != set()}
   ```
10. ```
    set('scarpa') == set('capras')
    ```
11. ```
    set('papa') != set('pappa')
    ```
12. ```
    set('pappa') != set('reale')
    ```
13. ```
    {(),()} == {(())}
    ```
14. ```
    {(),()} != {(()), (())}
    ```
15. ```
    [set()] == [set(),set()]
    ```
16. ```
    (set('gosh') | set('posh')) == (set('shopping') - set('in'))
    ```

5.10.6 Methods like operators

There are methods which behave like the operators `|`, `&`, `-`, `^` by creating a **NEW** set.

NOTE: differently from operators, these methods accept as parameter *any* sequence, not just sets:

Method	Result	Description	Related operator
`set.union(seq)`	`set`	union, creas a NEW set	|
`set.intersection(seq)`	`set`	intersection, creates a NEW set	`&`
`set.difference(seq)`	`set`	difference, creates a NEW set	`-`
`set.symmetric_difference(seq)`	`set`	symmetric difference, creates a NEW set	`^`

Methods which **MODIFY** the first set on which they are called (and return `None`!):

Method	Result	Description
`setA.update(setB)`	`None`	union, MODIFIES `setA`
`setA.intersection_update(setB)`	`None`	intersection, MODIFIES `setA`
`setA.difference_update(setB)`	`None`	difference, MODIFIES `setA`
`setA.symmetric_difference_update(setB)`	`None`	symmetric difference, MODIFIES `setA`

union

We'll only have a look at `union`/`update`, all other methods behave similarly

With `union`, given a set and a generic sequence (so not necessarily a set) we can create a NEW set:

[41]:
```
sa = {'g','a','r','a'}
```

[42]:
```
la = ['a','g','r','a','r','i','o']
```

[43]:
```
sb = sa.union(la)
```

[44]:
```
sb
```

[44]:
```
{'a', 'g', 'i', 'o', 'r'}
```

EXERCISE: with `union` we can use any sequence, but that's not the case with operators. Try writing `{1,2,3} | [2,3,4]` and see what happens.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[45]:
```
# write here
```

</div>

[45]:
```
# write here
```

We can verify `union` creates a new set with Python Tutor:

[46]:
```
sa = {'g','a','r','a'}
la = ['a','g','r','a','r','i','o']
sb = sa.union(la)

jupman.pytut()
```

[46]:
```
<IPython.core.display.HTML object>
```

update

If we want to MODIFY the first set instead, we can use the methods ending with `update`:

```
[47]: sa = {'g','a','r','a'}
```

```
[48]: la = ['a','g','r','a','r','i','o']
```

```
[49]: sa.update(la)
```

```
[50]: print(sa)
```

```
{'r', 'i', 'g', 'a', 'o'}
```

QUESTION: what did the call to `update` return?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: since Jupyter didn't show anything, it means the call to `update` method implicitly returned the `None` object.

</div>

Let's look what at happened with Python Tutor - we also added a `x =` to put in evidence what was returned by calling `.update`:

```
[51]: sa = {'g','a','r','a'}
la = ['a','g','r','a','r','i','o']
x = sa.update(la)
print(sa)
print(x)

jupman.pytut()
```

```
{'r', 'i', 'g', 'a', 'o'}
None
```

```
[51]: <IPython.core.display.HTML object>
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. ```
   set('case').intersection('sebo') == 'se'
   ```

2. ```
   set('naso').difference('caso')
   ```

3. ```
   s = {1,2,3}
   s.intersection_update([2,3,4])
   print(s)
   ```

4. ```
   s = {1,2,3}
   s = s & [2,3,4]
   ```

5. ```
   s = set('cartone')
   s = s.intersection('parto')
   print(s)
   ```
   ```

6.
```
sa = set("mastice")
sb = sa.difference("mastro").difference("collo")
print(sa)
print(sb)
```

7.
```
sa = set("mastice")
sb = sa.difference_update("mastro").difference_update("collo")
print(sa)
print(sb)
```

[ ]:

Exercise - everythingbut 2

Given sets `s1`, `s2` e `s3`, write some code which MODIFIES `s1` so that it also contains the elements of `s2` but not the elements of `s3`:

- Your code should work with *any* set `s1`, `s2`, `s3`
- **DO NOT** create new sets

Example - given:

```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])
```

After your code you should obtain:

```
>>> print(s1)
{'a', 'g', 'e', 'd', 'c'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[52]:
```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])

# write here
s1.update(s2)
s1.difference_update(s3)
print(s1)
```

```
{'e', 'c', 'a', 'd', 'g'}
```

</div>

[52]:
```
s1 = set(['a','b','c','d','e'])
s2 = set(['b','c','f','g'])
s3 = set(['b','f'])

# write here
```

```
{'e', 'c', 'a', 'd', 'g'}
```

5.10.7 Other methods

Method	Result	Description
set.add(el)	None	adds the specified element - if already present does nothing
set.remove(el)	None	removes the specified element - if not present raises an error
set.discard(el)	None	removes the specified element - if not present does nothing
`set.pop()`	obj	removes an arbitrary element from the set and returns it
`set.clear()`	None	removes all the elements
setA.issubset(setB)	bool	checks whether `setA` is a subset of `setB`
setA.issuperset(setB)	bool	checks whether `setA` contains all the elements of `setB`
setA.isdisjoint(setB)	bool	checks whether `setA` has no element in common with `setB`

`add` method

Given a set, we can add an element with the method `.add`:

[53]:
```
s = {3,7,4}
```

[54]:
```
s.add(5)
```

[55]:
```
s
```

[55]:
```
{3, 4, 5, 7}
```

If we add the same element twice, nothing happens:

[56]:
```
s.add(5)
```

[57]:
```
s
```

[57]:
```
{3, 4, 5, 7}
```

QUESTION: If we write this code, which result do we get?

```
s = {'a','b'}
s.add({'c','d','e'})
print(s)
```

1. prints `{'a','b','c','d','e'}`
2. prints `{{'a','b','c','d','e'}}`
3. prints `{'a','b',{'c','d','e'}}`
4. an error (which one?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 4 - produces `TypeError: unhashable type: 'set'` : we are trying to insert a set as element of another set, but sets are *mutable* so their *hash* label (which allows Python to find them quickly) might vary over time.

</div>

QUESTION: Look at the following code, which result does it produce?

```
x = {'a','b'}
y = set(x)
x.add('c')
print('x=',x)
print('y=',y)
```

1. an error (which one?)
2. `x` and `y` will be the same (how?)
3. `x` and `y` will be different (how?)

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 3. It will print:

```
x= {'c', 'a', 'b'}
y= {'a', 'b'}
```

because `y=set(x)` creates a NEW set by copying all the elements in the input sequence `x`.

Let's verify with Python Tutor:

</div>

[58]:
```
x = {'a','b'}
y = set(x)
x.add('c')

jupman.pytut()
```

[58]:
```
<IPython.core.display.HTML object>
```

`remove` method

The `remove` method takes the specified element out of the set. If it doesn't exist, it produces an error:

[59]:
```
s = {'a','b','c'}
```

[60]:
```
s.remove('b')
```

[61]:
```
s
```

[61]:
```
{'a', 'c'}
```

[62]:
```
s.remove('c')
```

[63]:
```
s
```

[63]:
```
{'a'}
```

```
s.remove('z')

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-266-a9e7a977e50c> in <module>
----> 1 s.remove('z')

KeyError: 'z'
```

Exercise - babbabiba

Given a string `word` of exactly 4 syllabs of two characters each, create a set `s` which contains tuples with 2 characters each. Each tuple must represent a syllab taken from `word`.

- to add elements to the set, only use `add`
- your code must work for any `word` of 4 bisyllabs

Example 1 - given:

```
word = "babbabiba"
```

after your code, it must result:

```
>>> print(s)
{('b', 'a'), ('b', 'i')}
```

Example 2 - given

```
word = "rubareru"
```

after your code, it must result:

```
>>> print(s)
{('r', 'u'), ('b', 'a'), ('r', 'e')}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[64]:
```
word = "babbabiba"
#word = "rubareru"

# write here

s = set()
s.add(tuple(word[:2]))
s.add(tuple(word[2:4]))
s.add(tuple(word[4:6]))
s.add(tuple(word[6:8]))
print(s)
```

```
{('b', 'i'), ('b', 'a')}
```

</div>

[64]:
```
word = "babbabiba"
#word = "rubareru"

# write here
```

```
{('b', 'i'), ('b', 'a')}
```

`discard` method

The `discard` method removes the specifed element from the set. If it doesn't exists, it does nothing (we may also say it *silently* discards the element):

[65]:
```
s = {'a','b','c'}
```

[66]:
```
s.discard('a')
```

[67]:
```
s
```

[67]:
```
{'b', 'c'}
```

[68]:
```
s.discard('c')
```

[69]:
```
s
```

[69]:
```
{'b'}
```

[70]:
```
s.discard('z')
```

[71]:
```
s
```

[71]:
```
{'b'}
```

Exercise - trash

⊛⊛ A waste processing plant receives a load of `trash`, which we represent as a set of strings:

```
trash = {'alkenes','vegetables','mercury','paper'}
```

To remove the contaminant elements which *might* be present (NOTE: they're not always present), the plant has exactly 3 `filters` (as list of strings) which will apply in series to the trash:

```
filters = ['cadmium','mercury','alkenes']
```

In order to check whether filters have effectively removed the contaminant(s), for each applied filter we want to see the state of the processed `trash`.

At the end, we also want to print all and *only* the contaminants which were actually removed (put them together in the variable `separated`)

- **DO NOT** use `if` commands
- **DO NOT** use cycles (the number of filters is fixed to 3, so you can jsut copy and paste code)
- Your code must work for *any* list `filters` of 3 elements and *any* set `trash`

Example - given:

```
filters = ['cadmium','mercury','alkenes']
trash = {'alkenes','vegetables','mercury','paper'}
```

After your code, it must show:

```
Initial trash: {'mercury', 'alkenes', 'vegetables', 'paper'}
Applying filter for cadmium : {'mercury', 'alkenes', 'vegetables', 'paper'}
Applying filter for mercury : {'alkenes', 'vegetables', 'paper'}
Applying filter for alkenes : {'vegetables', 'paper'}

Separated contaminants: {'mercury', 'alkenes'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[72]:
```python
filters = ['cadmium','mercury','alkenes']
trash = {'alkenes','vegetables','mercury','paper'}

separated = trash.intersection(filters) # creates a NEW set

# write here
s = "Applying filter for"
print("Initial trash:", trash)
trash.discard(filters[0])
print(s,filters[0],":", trash)
trash.discard(filters[1])
print(s,filters[1],":", trash)
trash.discard(filters[2])
print(s,filters[2],":", trash)
print("")


print("Separated contaminants:", separated)
```

```
Initial trash: {'vegetables', 'mercury', 'alkenes', 'paper'}
Applying filter for cadmium : {'vegetables', 'mercury', 'alkenes', 'paper'}
Applying filter for mercury : {'vegetables', 'alkenes', 'paper'}
Applying filter for alkenes : {'vegetables', 'paper'}

Separated contaminants: {'alkenes', 'mercury'}
```

</div>

[72]:
```python
filters = ['cadmium','mercury','alkenes']
trash = {'alkenes','vegetables','mercury','paper'}

separated = trash.intersection(filters) # creates a NEW set

# write here
```

```
Initial trash: {'vegetables', 'mercury', 'alkenes', 'paper'}
Applying filter for cadmium : {'vegetables', 'mercury', 'alkenes', 'paper'}
Applying filter for mercury : {'vegetables', 'alkenes', 'paper'}
Applying filter for alkenes : {'vegetables', 'paper'}

Separated contaminants: {'alkenes', 'mercury'}
```

`issubset` method

To check whether all elements in a set `sa` are contained in another set `sb` we can write `sa.issubset(sb)`. Examples:

```
[73]: {2,4}.issubset({1,2,3,4})
```

```
[73]: True
```

```
[74]: {3,5}.issubset({1,2,3,4})
```

```
[74]: False
```

WARNING: the empty set is always considered a subset of any other set

```
[75]: set().issubset({3,4,2,5})
```

```
[75]: True
```

`issuperset` method

To verify whether a set `sa` contains all the elements of another set `sb` we can write `sa.issuperset(sb)`. Examples:

```
[76]: {1,2,3,4,5}.issuperset({1,3,5})
```

```
[76]: True
```

```
[77]: {1,2,3,4,5}.issuperset({2,4})
```

```
[77]: True
```

```
[78]: {1,2,3,4,5}.issuperset({1,3,5,7,9})
```

```
[78]: False
```

WARNING: the empty set is always considered a subset of any other set

```
[79]: {1,2,3,4,5}.issuperset({})
```

```
[79]: True
```

`isdisjoint` method

A set is disjoint from another one if it doesn't have any element in common, we can check for disjointness by using the method `isdisjoint`:

```
[80]: {1,3,5}.isdisjoint({2,4})
```

```
[80]: True
```

```
[81]: {1,3,5}.isdisjoint({2,3,4})
```

[81]:
```
False
```

QUESTION: Given a set x, what does the following expression produce?

```
x.isdisjoint(x)
```

1. an error (which one?)
2. always True
3. always False
4. True or False according to the value of x

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: 4, True or False according to the value otx.

Probably you thought the expression always returns False: after all, how could a set ever be disjoint from itself? In fact the expression almost always returns False *except* for the particular case of the empty set:

```
x = set()
x.isdisjoint(x)
```

in which it returns True.

MORAL OF THE STORY: ALWAYS CHECK FOR THE EMPTY SET !

For this and many other methods the empty set often causes behaviours which aren't always intuitive, so we invite you to always check case by case.

</div>

5.10.8 Exercise - matrioska

⊛⊛ Given a list sets of exactly 4 sets, we define it a *matrioska* if each set contains all the elements of the previous set (plus eventually others). Write some code which PRINTS True if the sequence is a matrioska, otherwise PRINTS False.

- **DO NOT** use if
- your code must work for *any* sequence of exactly 4 sets
- **HINT**: you can create a list of 3 booleans which verify whether a set is contained in the next one …

Example 1 - given:

```
sets = [{'a','b'},
        {'a','b','c'},
        {'a','b','c','d','e'},
        {'a','b','c','d','e','f','g','h','i'}]
```

after your code, it must print:

```
Is the sequence a matrioska? True
```

Example 2 - given:

```
sets = [{'a','b'},
        {'a','b','c'},
        {'a','e','d'},
        {'a','b','d','e'}]
```

after your code, it must print:

```
Is the sequence a matrioska? False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[82]:
```python
sets = [{'a','b'},
        {'a','b','c'},
        {'a','b','c','d','e'},
        {'a','b','c','d','e','f','g','h','i'}]


#sets = [{'a','b'},
#         {'a','b','c'},
#         {'a','e','d'},
#         {'a','b','d','e'}]


# write here

checks = [ sets[0].issubset(sets[1]),
           sets[1].issubset(sets[2]),
           sets[2].issubset(sets[3]) ]

print("Is the sequence a matrioska?", checks.count(True) == 3)
```

```
Is the sequence a matrioska? True
```

</div>

[82]:
```python
sets = [{'a','b'},
        {'a','b','c'},
        {'a','b','c','d','e'},
        {'a','b','c','d','e','f','g','h','i'}]


#sets = [{'a','b'},
#         {'a','b','c'},
#         {'a','e','d'},
#         {'a','b','d','e'}]


# write here
```

```
Is the sequence a matrioska? True
```

5.10.9 Exercise with functions

WARNING: The following exercises require to know:

Control flow[227]

Functions[228]

Exercise - syllabs

Write a function `syllabs` which given a string `word` made by only bisyllabs and a set `found`, finds all the distinct bisyllabs and puts them into the set `found`.

- **NOTE**: the function `syllabs` return NOTHING !

Example 1:

```
>>> found = set()
>>> syllabs("banana", found)
>>> print(found)
{'an', 'ba'}
```

Example 2:

```
>>> found = set()
>>> syllabs("bonobo", found)
>>> print(found)
{'bo', 'on'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[83]:
```
# write here
def syllabs(word, t):
    for i in range(len(word)//2):
        t.add(word[i:i+2])

found = set()
syllabs("banana", found)
#print(found)

found = set()
syllabs("bonobo", found)
#print(found)
```

</div>

[83]:
```
# write here
```

[227] https://en.softpython.org/#control-flow
[228] https://en.softpython.org/functions/functions-sol.html

Exercise - distinguish

⊛⊛ Write a function `distinguish` which given a list `big_list` containing sublists of *two* characters each, RETURN a NEW LIST containing all the *distinct* sublists (ignoring the duplicated sublists)

- the returned list must have the elements *in the same order* in which they were found in `big_list`
- to know fast whether a sublist was already found, **use a set**
- **DO NOT** search in lists (so no `count`, `index`, `in` in lists - they're slow!)
- **DO NOT** remove from lists (so no `remove` from lists - it's slow!)
- **HINT**: lists are *mutable*, can we place them in a set? If it's not possible, what can we do?

Example:

```
>>> big_list= [ ['d','d'],['a','b'],['d','d'],['c','a'],['c','a'],['d','d'],['a','b']␣
↪]
>>> print(distinguish( big_list))
[['d', 'd'], ['a', 'b'], ['c', 'a']]
#NOTE: variable big_list MUST NOT be modified:
>>> print(big_list)
[ ['d','d'],['a','b'],['d','d'],['c','a'],['c','a'],['d','d'],['a','b'] ]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[84]:
```python
# write here

def distinguish(blist):
    s = set()
    ret = []

    for sublist in blist:
        #  In sets we can't place lists because they are mutable,
        #  but we can insert tuples
        tup = tuple(sublist)

        #  Checking whether an element belongs to a set it's very fast:
        #  it is independent from the set dimension!

        if tup not in s:
            ret.append(sublist)
            # Adding an element to a set is very fast:
            # it is independent from the set dimension!
            s.add(tup)

    return ret

big_list = [ ['d','d'],['a','b'],['d','d'],['c','a'],['c','a'],['d','d'],['a','b'] ]
#print('distincts:', distinguish(big_list))
#print('big_list:', big_list)
```

</div>

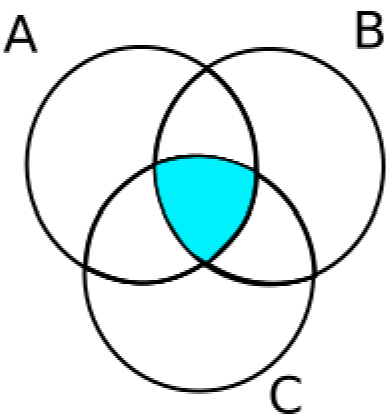
[84]:
```python
# write here
```

5.10.10 Verifify comprehension

WARNING

The following exercises contain tests with *asserts*. To understand how to execute them, read first Error handling and testing[229]

Exercise - intersectron



Given a list `sets` containing an arbitrary number of sets, RETURN a NEW set which contains the elements common to all sets.

To solve the exercise, you can intersecate a set at a time with a `for` cycle (slow) or with the technique described here[230] (short and fast).

- try to solve it in **both** ways
- **BEWARE** of the empty list!
- your code must work with **any** number of sets (the image is just an example)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[85]:
```python
def inter_for(sets):


    if len(sets) == 0:
        return set()

    first = True

    for el in sets:
        if first:
            ret = set(el)
            first = False
        else:
            ret.intersection_update(el)
    return ret
```

(continues on next page)

[229] https://en.softpython.org/errors-and-testing/errors-and-testing-sol.html
[230] https://stackoverflow.com/a/2541814

(continued from previous page)

```
# TEST START - DO NOT TOUCH !
assert inter_for([]) == set()
assert inter_for([set(),set()]) == set()
assert inter_for([set(),set(),set()]) == set()
assert inter_for([{'a'},{'a'},{'a'}]) == {'a'}
assert inter_for([{'a','b'},{'b'},{'b'}]) == {'b'}
assert inter_for([{'a'},{'a','b'},{'a'}]) == {'a'}
assert inter_for([{'c'},{'c'},{'c','b'}]) == {'c'}
assert inter_for([{'a','b'},{'a','b'},{'a','b'}]) == {'a','b'}
assert inter_for([{'a','b','c'},{'a','b','c','d'},{'b','c','d'}, {'b','c'}]) == {'b',
↪'c'}
# check we didn't modify the input sets
s = {'a','b'}
assert inter_for([s,{'b','c'}]) == {'b'}
assert s == {'a','b'}
# TEST END
```

</div>

[85]:
```
def inter_for(sets):

    raise Exception('TODO IMPLEMENT ME !')

# TEST START - DO NOT TOUCH !
assert inter_for([]) == set()
assert inter_for([set(),set()]) == set()
assert inter_for([set(),set(),set()]) == set()
assert inter_for([{'a'},{'a'},{'a'}]) == {'a'}
assert inter_for([{'a','b'},{'b'},{'b'}]) == {'b'}
assert inter_for([{'a'},{'a','b'},{'a'}]) == {'a'}
assert inter_for([{'c'},{'c'},{'c','b'}]) == {'c'}
assert inter_for([{'a','b'},{'a','b'},{'a','b'}]) == {'a','b'}
assert inter_for([{'a','b','c'},{'a','b','c','d'},{'b','c','d'}, {'b','c'}]) == {'b',
↪'c'}
# check we didn't modify the input sets
s = {'a','b'}
assert inter_for([s,{'b','c'}]) == {'b'}
assert s == {'a','b'}
# TEST END
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[86]:
```
def inter_fast(sets):


    if len(sets) == 0:
        return set()

    return set.intersection(*sets)


# TEST START - DO NOT TOUCH !
assert inter_fast([]) == set()
assert inter_fast([set(),set()]) == set()
```

(continues on next page)

(continued from previous page)

```
assert inter_fast([set(),set(),set()]) == set()
assert inter_fast([{'a'},{'a'},{'a'}]) == {'a'}
assert inter_fast([{'a','b'},{'b'},{'b'}]) == {'b'}
assert inter_fast([{'a'},{'a','b'},{'a'}]) == {'a'}
assert inter_fast([{'c'},{'c'},{'c','b'}]) == {'c'}
assert inter_fast([{'a','b'},{'a','b'},{'a','b'}]) == {'a','b'}
assert inter_fast([{'a','b','c'},{'a','b','c','d'},{'b','c','d'}, {'b','c'}]) == {'b',
↪'c'}
# check we didn't modify the input sets
s = {'a','b'}
assert inter_fast([s,{'b','c'}]) == {'b'}
assert s == {'a','b'}
# TEST END
```

</div>

[86]:
```
def inter_fast(sets):

    raise Exception('TODO IMPLEMENT ME !')

# TEST START - DO NOT TOUCH !
assert inter_fast([]) == set()
assert inter_fast([set(),set()]) == set()
assert inter_fast([set(),set(),set()]) == set()
assert inter_fast([{'a'},{'a'},{'a'}]) == {'a'}
assert inter_fast([{'a','b'},{'b'},{'b'}]) == {'b'}
assert inter_fast([{'a'},{'a','b'},{'a'}]) == {'a'}
assert inter_fast([{'c'},{'c'},{'c','b'}]) == {'c'}
assert inter_fast([{'a','b'},{'a','b'},{'a','b'}]) == {'a','b'}
assert inter_fast([{'a','b','c'},{'a','b','c','d'},{'b','c','d'}, {'b','c'}]) == {'b',
↪'c'}
# check we didn't modify the input sets
s = {'a','b'}
assert inter_fast([s,{'b','c'}]) == {'b'}
assert s == {'a','b'}
# TEST END
```

5.10.11 References

- Think Python, Chapter 19.5, The Goodies - sets[231]
- W3 Resources - sets[232]

[231] http://greenteapress.com/thinkpython2/html/thinkpython2020.html#sec227
[232] https://www.w3resource.com/python-exercises/sets/

5.11 Dictionaries 1 - Introduction

5.11.1 Download exercises zip

Browse files online[233]

Dictionaries are mutable containers which allow us to rapidly associate elements called *keys* to some *values*

- *Keys* are immutable, don't have order and there cannot be duplicates
- *Values* can be duplicated

Given a key, we can find the corresponding value very fast.

5.11.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
sets
    dictionaries1.ipynb
    dictionaries1-sol.ipynb
    dictionaries2.ipynb
    dictionaries2-sol.ipynb
    dictionaries3.ipynb
    dictionaries3-sol.ipynb
    dictionaries4.ipynb
    dictionaries4-sol.ipynb
    dictionaries5.ipynb
    dictionaries5-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `dictionaries1.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ⊛ to ⊛⊛⊛⊛ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[233] https://github.com/DavidLeoni/softpython-it/tree/master/dictionaries

5.11.3 Creating a dictionary

In everyday life, when thinking about a dictionary we typically refer to a book which given an item (for example `'chair'`), allows us to **rapidly** find the related description (i.e. `a piece of furniture to sit on`).

In Python we have a data structure called `dict` which provides an easy way to represent dictionaries.

Following the previous example, we might create a `dict` with different items like this:

```
[2]: {'chair':'a piece of furniture to sit on',
      'cupboard':'a cabinet for storage',
      'lamp': 'a device to provide illumination'
     }
```

```
[2]: {'chair': 'a piece of furniture to sit on',
      'cupboard': 'a cabinet for storage',
      'lamp': 'a device to provide illumination'}
```

Let's be clear about the naming:

> Dictionaries are mutable containers which allow us to rapidly associate elements called **keys** to some **values**.

The definition says we have *keys* (in the example `'chair'`, `'cupboard'`, etc), while the descriptions from the example (`'a piece of furniture to sit on'`) in Python are going to be called *values*.

When we create a dictionary, we first write a curly bracket `{`, then we follow it with a series of key `:` value couples, each followed by a comma `,` (except the last one, in which the comma is optional). At the end we close with a a curly bracket `}`

Placing spaces or newlines inside **is optional**. So we can also write like this:

```
[3]: {'chair'    : 'a piece of furniture to sit on',

      'cupboard' :     'a cabinet for storage',
      'lamp'     : 'a device to provide illumination'
     }
```

```
[3]: {'chair': 'a piece of furniture to sit on',
      'cupboard': 'a cabinet for storage',
      'lamp': 'a device to provide illumination'}
```

Or also everything on a row:

```
[4]: {'chair':'a piece of furniture to sit on','cupboard':'a cabinet for storage','lamp':
     ↪'a device to provide illumination'}
```

```
[4]: {'chair': 'a piece of furniture to sit on',
      'cupboard': 'a cabinet for storage',
      'lamp': 'a device to provide illumination'}
```

Note if we use short words Python will probably print the dictionary in single a row anyway:

```
[5]: {'barca': 'remo',
      'auto': 'ruota',
      'aereo': 'ala'}
```

```
[5]: {'aereo': 'ala', 'auto': 'ruota', 'barca': 'remo'}
```

Putting a comma after the last couple does not give errors:

```
[6]: {
 'ship': 'paddle',
 'car': 'wheel',
 'airplane': 'wing',  # note 'extra' comma
}
```

```
[6]: {'airplane': 'wing', 'car': 'wheel', 'ship': 'paddle'}
```

Let's see how a dictionary is represented in Python Tutor - to ease the job, we will assign the variable `furniture` to it

```
[7]: # WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
     #          (it's sufficient to execute it only once)

     import jupman
```

```
[8]:
furniture = {
    'chair'    : 'a piece of furniture to sit on',
    'cupboard' : 'a cabinet for storage',
    'lamp'     : 'a device to provide illumination'
}
print(furniture)

jupman.pytut()
```

```
{'cupboard': 'a cabinet for storage', 'chair': 'a piece of furniture to sit on', 'lamp
↪': 'a device to provide illumination'}
```

```
[8]: <IPython.core.display.HTML object>
```

We note that once executed, an arrow appears pointing from `furniture` to an orange/yellow memory region. The keys have orange background, while the corresponding values have yellow background. Looking at arrows and colors, we can guess that whenever we're assigning variables, dictionaries behave like other data structures, like lists and sets.

QUESTION: Look at the following code, and try guessing what happens during execution - at the end, how will memory be organized? What will be printed? Where will arrows go?

```
[9]:
da = {
    'chair'    : 'a piece of furniture to sit on',
    'cupboard' : 'a cabinet for storage',
    'lamp'    : 'a device to provide illumination'
}

db = {
 'ship': 'paddle',
 'car': 'wheel',
 'airplane': 'wing'
}
dc = db
db = da
da = dc
dc = db
#print(da)
#print(db)
#print(dc)

jupman.pytut()
```

```
[9]: <IPython.core.display.HTML object>
```

The keys

Let's try to better understand which keys we can use by looking again at the definition:

> Dictionaries are mutable containers which allow us to rapidly associate elements called *keys* to some *values*
>
> - **Keys are immutable, don't have order and there cannot be duplicates**
> - Values can be duplicated

QUESTION: have a careful look at the words in bold - can you tell a data structure we've already seen which has these features?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The keys of dictionaries for many aspects behave like elements of a set.

Have you read the tutorial on sets?[234]**?**

Before going on, make sure to understand well the section on mutable elements and hashes[235]

</div>

Keys are immutable

QUESTION: The definition does not force us to use strings as keys, other types are also allowed. But can we use all the types we want?

For each of the following examples, try to tell whether the dictionary can be created or we will get an error (which one?). Also check how they are represented in Python Tutor.

1. integers

```
{
  4 : 'cats',
  3 : 'dogs'
}
```

2. float

```
{
  4.0 : 'cats',
  3.0 : 'dogs'
}
```

3. strings

[234] https://en.softpython.org/sets/sets-sol.html
[235] https://eb.softpython.org/sets/sets-sol.html#Mutable-elements-and-hashes

```
{
  'a' : 'cats',
  'b' : 'dogs'
}
```

4. lists

```
{
    [1,2] : 'zam',
    [3,4] : 'zum'
}
```

5. tuples

```
{
    (1,2) : 'zam',
    (4,3) : 'zum'
}
```

6. sets

```
{
    {1,2} : 'zam',
    {3,4} : 'zum'
}
```

7. other dictionaries (check the first part of the definition !)

```
{
    {'a':'x','b':'y'} : 'zam',
    {'c':'w','d':'z'} : 'zum'
}
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: integers, float, strings and tuples are IMMUTABLE and so we can use them as keys (see definition). Instead, lists, sets (and other dictionaries) are MUTABLE, so we cannot use them as keys. If we try using a MUTABLE element such as a list like if it were the key of a dictionary, Python will complain, telling us the object is not *hashable* (exactly as it would complain if we tried to insert it in a set)

```
>>> {   [1,2]:'zam',
        [3,4]:'zum'}

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-12-c3c2d6cc97b8> in <module>
      1 { [1,2]:'zam',
----> 2   [3,4]:'zum'}

TypeError: unhashable type: 'list'
```

</div>

Keys don't have order

In a real-life dictionary, items are always ordered according to some criteria, typically in alphabetical order.

With Python we need to consider this important difference:

- The keys are immutable, **don't have order** and there cannot be duplicates

When we say that a collection 'does not have order', it means that the order of elements we see when we insert or print them does not matter to determine whether a collection is equal to another one. In dictionaries, it means that if we specify couples in a different order, we obtain dictionaries that Python considers as equal.

For example, the following dictionaries can all be considered as equal:

[10]:
```
{
 'ships' :'port',
 'airplanes': 'airport',
 'trains': 'station'
}
```

[10]:
```
{'airplanes': 'airport', 'ships': 'port', 'trains': 'station'}
```

[11]:
```
{
 'airplanes': 'airport',
 'ships' :'port',
 'trains': 'station'
}
```

[11]:
```
{'airplanes': 'airport', 'ships': 'port', 'trains': 'station'}
```

[12]:
```
{
 'trains': 'station',
 'ships' :'port',
 'airplanes': 'airport'
}
```

[12]:
```
{'airplanes': 'airport', 'ships': 'port', 'trains': 'station'}
```

Printing a dictionary: you may have noticed that Jupyter always prints the keys in alphabetical order. This is just a courtesy for us, but do not be fooled by it! If we try a native `print` we will obtain a different result!

[13]:
```
print({
 'ships' :'port',
 'airplanes': 'airport',
 'trains': 'station'
})
```

```
{'trains': 'station', 'airplanes': 'airport', 'ships': 'port'}
```

Key duplicates

- Keys are immutable, don't have order and **there cannot be duplicates**

We might ask ourselves how Python manages duplicates in keys. Let's try to create a duplicated couple on purpose:

[14]:
```
{
    'chair'   : 'a piece of furniture to sit on',
    'chair'   : 'a piece of furniture to sit on',
    'lamp'    : 'a device to provide illumination'
}
```

[14]:
```
{'chair': 'a piece of furniture to sit on',
 'lamp': 'a device to provide illumination'}
```

We notice Python didn't complain and silently discarded the duplicate.

What if we try inserting a couple with the same key but different value?

[15]:
```
{
    'chair'   : 'a piece of furniture to sit on',
    'chair'   : 'a type of seat',
    'lamp'    : 'a device to provide illumination'
}
```

[15]:
```
{'chair': 'a type of seat', 'lamp': 'a device to provide illumination'}
```

Notice Python kept only the last couple.

The values

Let's see once again the definition:

> Dictionaries are mutable containers which allow us to rapidly associate elements called keys to some values
>
> - Keys are immutable, don't have order and there cannot be duplicates
> - **Values can be duplicated**

Seems like values have less constraints than keys.

QUESTION: For each of the following examples, try to tell whether we can create the dictionary or we will get an error (which one?). Check how they are represented in Python Tutor.

1. integers

```
{
    'a':3,
    'b':4
}
```

2. duplicated integers

```
{
    'a':3,
    'b':3
}
```

3. float

```
{
    'a':3.0,
    'b':4.0
}
```

4. strings

```
{
    'a' : 'ice',
    'b' : 'fire'
}
```

5. lists

```
{
    'a' : ['t','w'],
    'b' : ['x'],
    'c' : ['y','z','k']
}
```

6. duplicated lists

```
{
    'a' : ['x','y','z'],
    'b' : ['x','y','z']
}
```

7. lists containing duplicates

```
{
    'a' : ['x','y','y'],
    'b' : ['z','y','z']
}
```

8. tuples

```
{
    'a': (6,9,7),
    'b': (8,1,7,4)
}
```

9. sets

```
{
    'a' : {6,5,6},
    'b' : {2,4,1,5}
}
```

10. dictionaries

```
{
    'a': {
            'x':3,
            'y':9
         },
    'b': {
            'x':3,
```

(continues on next page)

(continued from previous page)

```
               'y':9,
               'z':10
           },
}
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: We can freely put whatever we please as values, Python will not complain. In particular, notice how different keys can have the same value.

</div>

Empty dictionary

We can create an empty dictionary by writing `{}`:

WARNING: THIS IS NOT THE EMPTY SET[236] !!

[16]:
```
{}
```

[16]:
```
{}
```

[17]:
```
type({})
```

[17]:
```
dict
```

A dictionary is a collection, and as we've already seen (with lists, tuples and sets), we can create an empty collection by typing its type, in this case `dict`, followed by round brackets:

[18]:
```
dict()
```

[18]:
```
{}
```

Let's see how it's represented in Python Tutor:

[19]:
```
diz = dict()

jupman.pytut()
```

[19]:
```
<IPython.core.display.HTML object>
```

Keys and heterogenous values

So far we've always used keys all of the same type and values all of the same type, but this is not mandatory. (the only required thing is for key types to be immutable):

[20]:
```
{
    "a": 3,
    "b": ["a", "list"],
     7 : ("this","is","a","tuple")
}
```

236 https://en.softpython.org/sets/sets-sol.html#Empty-set

```
[20]: {7: ('this', 'is', 'a', 'tuple'), 'a': 3, 'b': ['a', 'list']}
```

NOTE: Although mixing types is possible, it's not advisable!

Throwing different types inside a dictionary often brings misfortune, as it increases probability of incurring into bugs.

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. ```
   {'a':'b'        ,
    'c':'d'

    }
   ```

2. ```
   {'a b':'c',
    'c d':'e f'}
   ```

3. ```
   {'a' = 'c',
    'b' = 'd'}
   ```

4. ```
   {'a':'b':
    'c':'d'}
   ```

5. ```
   {
       "1":[2,3],
       "2,3":1,
   }
   ```

6. ```
   type({'a:b,c:d'})
   ```

7. ```
   {'a':'b';
    'c':'d'}
   ```

8. ```
   {'a:b',
    'c:d'}
   ```

9. ```
   {5,2:
    4,5}
   ```

10. ```
    {1:2,
     1:3}
    ```

11. ```
    {2:1,
     3:1}
    ```

12. ```
    {'a':'b',
     'c':'d',}
    ```

13. ```
    type({'a','b',
          'c','d'})
    ```

14. ```
    {'a':'b',
     'c':'d',
     'e','f'}
    ```

15. `{{}: 2}`

16. `{(1,2):[3,4]}`

17. `{[1,2]:(3,4)}`

18. `{'[1,2]':(3,4)}`

19. `{{1,2}:(3,4)}`

20. `{len({1,2}):(3,4)}`

21. `{5:{'a':'b'}}`

22. `{"a":{1:2}}`

23. `{"a":{[1]:2}}`

24. `{"a":{1:[2]}}`

25. `{["a":{1:[2]}]}`

26. `set([{2:4}])`

5.11.4 Dictionary from a sequence of couples

We can obtain a dictionary by specifying a sequence of key/value couples as parameter of the function `dict`. For example we could pass a list of tuples:

[21]:
```
dict(   [
          ('flour',500),
          ('eggs',2),
          ('sugar',200),
        ])
```

[21]:
```
{'eggs': 2, 'flour': 500, 'sugar': 200}
```

We can also use other sequences, the important bit is that subsequences must all have two elements. For example, here is a tuple of lists:

[22]:
```
dict(   (
          ['flour',500],
          ['eggs',2],
          ['sugar',200],
        ))
```

[22]:
```
{'eggs': 2, 'flour': 500, 'sugar': 200}
```

If a subsequence has a number of elements different from two, we obtain this error:

```
>>> dict(  (
         ['flour',500],
         ['rotten','eggs', 3],
         ['sugar',200],
       ))

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-88-563d301b4aef> in <module>
      2           ['flour',500],
      3           ['rotten','eggs', 3],
      4           ['sugar',200],
      5         ))

ValueError: dictionary update sequence element #1 has length 3; 2 is required
```

QUESTION: Compare the following expressions. Do they do the same thing? If so, which one would you prefer?

```
dict( {
        ('a',5),
        ('b',8),
        ('c',3)
      } )
```

```
dict( (
        {'a',5},
        {'b',8},
        {'c',3}
      )
    )
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The expressions do NOT produce the same result, and we must definitely prefer the first one.

On our pc, we obtained this:

WARNING: on your computer you may get different results!

```
# first
>>> dict( {
            ('a',5),
            ('b',8),
            ('c',3)} )

{'b': 8, 'a': 5, 'c': 3}
```

```
# second
>>> dict( (
            {'a',5},
            {'b',8},
            {'c',3}) )

{'a': 5, 8: 'b', 3: 'c'}
```

In the first case we started with a set of tuples: since it is a set, the elements inside it are memorized in an order we *cannot* predict. When Python checks the tuples inside, for each of them obtains a key/value couple. Now, from the dictionary definition we know dictionary keys are also memorized without a precise order. Thus, inserting keys in an order or another doesn't matter, the only important thing is keeping the key/value distinction. In the dictionary print we see the same couples we specified, only in different order: the proper couples have been created because tuples *are* ordered indeed.

In the second case we started instead from a tuple of sets, so Python visited the elements of the tuple in the same order as the one we see: alas, by specifying the couples like sets the order in which Python read the elements becomes unpredictable. On our computer, with the first set we've been lucky and Python first read `'a'` and then `5`, with the following sets it read instead first the number and then the character! On your computer you might see a completely different result!

</div>

QUESTION: Look at the following expressions, and for each try guessing which result it produces (or if it gives an error):

1.
```
dict('abcd')
```
2.
```
dict(['ab','cd'])
```
3.
```
dict(['a1','c2'])
```
4.
```
dict([])
```
5.
```
dict(())
```
6.
```
dict(('  ',))    # nasty
```

5.11.5 Dictionary from keyword arguments

As further creation method, we can specify keys as they were parameters with a name:

[23]:
```
dict(a=5,b=6)
```

[23]:
```
{'a': 5, 'b': 6}
```

WARNING: keys will be subject to the same restrictive rules of function parameter names!

For example, by using curly brackets this dictionary is perfectly lecit:

[24]:
```
{'a b' : 2,
 'c d' : 6}
```

[24]:
```
{'a b': 2, 'c d': 6}
```

But if we try creating it using `a b` as argument of `dict`, we will incur into problems:

```
>>> dict(a b=2, c d=6)

  File "<ipython-input-97-444f8661585a>", line 1
    dict(a b=2, c d=6)
           ^
SyntaxError: invalid syntax
```

Strings will also give trouble:

```
>>> dict('a b'=2,'c d'=6)

  File "<ipython-input-98-45aafbb56e81>", line 1
    dict('a b'=2,'c d'=6)
        ^
SyntaxError: keyword can't be an expression
```

And be careful about tricks like using variables, we won't obtain the desired result:

[25]:
```
ka = 'a b'
kc = 'c d'

dict(ka=2,kc=6)
```

[25]:
```
{'ka': 2, 'kc': 6}
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. ```
   dict(3=5,2=8)
   ```
2. ```
   dict('costs'=9,'benefits'=15)
   ```
3. ```
   dict(_costs=9,_benefits=15)
   ```
4. ```
   dict(33trentini=5)
   ```
5. ```
   dict(trentini33=5)
   ```
6. ```
   dict(trentini_33=5)
   ```
7. ```
   dict(trentini-33=5)
   ```
8. ```
   dict(costs=1=2,benefits=3=3)
   ```
9. ```
   dict(costs=1==2,benefits=3==3)
   ```
10. ```
    v1 = 6
    v2 = 8
    dict(k1=v1,k2=v2)
    ```

5.11.6 Copying a dictionary

There are two ways to copy a dictionary, you can either do a *shallow* copy or a *deep* copy.

Shallow copy

It is possible to create a shallow copy by passing another dictionary to function `dict`:

[26]:
```
da = {'x':3,
      'y':5,
      'z':1}
```

[27]:
```
db = dict(da)
```

[28]:
```
print(da)
```

```
{'y': 5, 'z': 1, 'x': 3}
```

[29]:
```
print(db)
```

```
{'y': 5, 'z': 1, 'x': 3}
```

In Python Tutor we will see two different memory regions:

[30]:
```
da = {'x':3,
      'y':5,
      'z':1}
db = dict(da)

jupman.pytut()
```

[30]:
```
<IPython.core.display.HTML object>
```

QUESTION: can we also write like this? With respect to the previous example, will we obtain different results?

[31]:
```
da = {'x':3,
      'y':5,
      'z':1}
db = dict(dict(da))

jupman.pytut()
```

[31]:
```
<IPython.core.display.HTML object>
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The code produces the same results of previous example, although it is not efficient (a temporary dictionary will be created by the internal `dict` and then it will be immediately discarded)

</div>

Mutable values: In the example we used integer values, which are *immutable*. If we tried *mutable* values like lists, what would happen?

[32]:
```
da = {'x':['a','b','c'],
      'y':['d'],
      'z':['e','f']}
db = dict(da)

jupman.pytut()
```

[32]:
```
<IPython.core.display.HTML object>
```

If you try executing Python Tutor, you will see an explosion of arrows which go from the new dictionary `db` to the values of `da` (which are lists). No panic! We are going to give a better explanation in the next notebook, for now just note that **with the shallow copy of mutable values the new dictionary will have memory regions in common with the original dictionary.**

Deep copy

When there are mutable shared memory regions like in the case above, it's easy to do mistakes and introduce subtle bugs you might notice much later in the development cycle.

In order to have completely separated memory regions, we can use *deep copy*.

First we must tell Python we intend to use functions from the module `copy`, and then we will be allowed to call its `deepcopy` function:

[33]:
```
from copy import deepcopy

da = {'x':['a','b','c'],
      'y':['d'],
      'z':['e','f']}
db = deepcopy(da)

jupman.pytut()
```

[33]:
```
<IPython.core.display.HTML object>
```

If you execute the code in Python Tutor, you will notice that by following the arrow from `db` we will end up in an totally new orange/yellow memory region, which shares nothing with the memory region pointed by `da`.

QUESTION: Have a look at the following code - after its execution, will you see arrows going from `db` to elements of `da`?

[34]:
```
da = {'x': {1,2,3},
      'y': {4,5}}
db = dict(da)
jupman.pytut()
```

[34]:
```
<IPython.core.display.HTML object>
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Yes, because the values from `da` are sets which are mutable.

</div>

5.11.7 Continue

Go on reading Dictionaries 2[237]

[ ]:
```
```

[237] https://en.softpython.org/dictionaries/dictionaries2-sol.html

5.12 Dictionaries 2 - operators

5.12.1 Download exercise zip

Browse online files[238]

There are several operators to manipulate dictionaries:

Operator	Return	Description
`len`(dict)	`int`	Retorn the number of keys
dict`[`chiave`]`	obj	Return the value associated to the key
dict`[`chiave`]` = valore		Adds or modify the value associated to the key
`del` dict`[`chiave`]`		Removes the key/value couple
obj `in` dict	`bool`	Return `True` if the key obj is present in dict
`==`,`!=`	`bool`	Checks whether two dictionaries are equal or different

5.12.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
sets
    dictionaries1.ipynb
    dictionaries1-sol.ipynb
    dictionaries2.ipynb
    dictionaries2-sol.ipynb
    dictionaries3.ipynb
    dictionaries3-sol.ipynb
    dictionaries4.ipynb
    dictionaries4-sol.ipynb
    dictionaries5.ipynb
    dictionaries5-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `dictionaries2.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[238] https://github.com/DavidLeoni/softpython-en/tree/master/dictionaries

5.12.3 len

We can obtain the number of key/value associations in a dictionary by using the function `len`:

[2]:
```python
len({'a':5,
     'b':9,
     'c':7
})
```

[2]:
```
3
```

[3]:
```python
len({3:8,
     1:3
})
```

[3]:
```
2
```

[4]:
```python
len({})
```

[4]:
```
0
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `len(dict())`
2. `len({'a':{}})`
3. `len({(1,2):{3},(4,5):{6},(7,8):{9}})`
4. `len({1:2,1:2,2:4,2:4,3:6,3:6})`
5. `len({1:2,',':3,',':4,})`
6. `len(len({3:4,5:6}))`

5.12.4 Reading a value

At the end of dictionaries definition, it is reported:

> **Given a key, we can find the corresponding value very fast**

How can we specify the key to search? It's sufficient to use square brackets `[ ]`, a bit like we already did for lists:

[5]:
```python

furniture = {
      'chair'    : 'a piece of furniture to sit on',
      'cupboard' : 'a cabinet for storage',
      'lamp'     : 'a device to provide illumination'
    }

```

[6]:
```python
furniture['chair']
```

[6]:
```
'a piece of furniture to sit on'
```

```
[7]: furniture['lamp']
```

```
[7]: 'a device to provide illumination'
```

WARNING: What we put in square parenthesis **must** be a key present in the dictionary

If we put keys which are not present, we will get an error:

```
>>> furniture['armchair']

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-19-ee891f51417b> in <module>
----> 1 furniture['armchair']

KeyError: 'armchair'
```

Fast disorder

Whenever we give a key to Python, how fast is it in getting the corresponding value? Very fast, so much so the speed *does not depend on the dictionary dimension.* Whether it is small or huge, given a key it will always find the associated value in about the same time.

When we hold a dictionary in real life, we typically have an item to search for and we turn pages until we get what we're looking for: the fact items are sorted allows us to rapidly find the item.

We might expect the same also in Python, but if we look at the definition we find a notable difference:

> Dictionaries are mutable containers which allow us to rapidly associate elements called keys to some values
>
> Keys are immutable, **don't have order** and there cannot be duplicates Values can be duplicated

If keys are *not* ordered, how can Python get the values so fast? The speed stems from the way Python memorizes keys, which is based on *hashes,* similarly for what happens with sets[239]. The downside is we can only *immutable* objects as keys.

QUESTION: If we wanted to print the value `'a device to provide illumination'` we see at the bottom of the dictionary, without knowing it corresponds to `lamp`, would it make sense to write something like this?

```
furniture = {'chair':'a piece of furniture to sit on',
             'cupboard':'a cabinet for storage',
             'lamp': 'a device to provide illumination'
}

print( furniture[2] )
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Absolutely NOT. The couples key/value in the dictionary *are not* ordered, so it makes no sense to get a value at a given position.

</div>

QUESTION: Look at the following expressions, and for each try guessing which result it produces (or if it gives an error):

[239] https://en.softpython.org/sets/sets-sol.html#Mutable-elements-and-hashes

```
kabbalah = {
    1 : 'Progress',
    3 : 'Love',
    5 : 'Creation'
}
```

- ```
  kabbalah[0]
  ```
- ```
  kabbalah[1]
  ```
- ```
  kabbalah[2]
  ```
- ```
  kabbalah[3]
  ```
- ```
  kabbalah[4]
  ```
- ```
  kabbalah[5]
  ```
- ```
  kabbalah[-1]
  ```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: In the dictionary we have keys which are integer numbers: so we can use numbers among square brackets, which we will call *keys,* but not *positions.*

The unique expressions which will produce results are those for which the number specified among the square brackets is effectively present among the keys:

```
>>> kabbalah[1]
'Progress'
>>> kabbalah[3]
'Love'
>>> kabbalah[5]
'Creation'
```

All others will give `KeyError`, like:

```
>>> kabbalah[2]

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-29-de66b9721e9b> in <module>
      5 }
      6
----> 7 kabbalah[2]

KeyError: 2
```

</div>

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1. ```
   {'a':4,'b':5}('a')
   ```

2.
```
{1:2,2:3,3:4}[2]
```

3.
```
{'a':1,'b':2}['c']
```

4.
```
{'a':1,'b':2}[a]
```

5.
```
{'a':1,'b':2}[1]
```

6.
```
{'a':1,'b':2,'c':3}['c']
```

7.
```
{'a':1,'b':2,'c':3}[len(['a','b','c'])]
```

8.
```
{(3,4):(1,2)}[(1,2)]
```

9.
```
{(1,2):(3,4)}[(1,2)]
```

10.
```
{[1,2]:[3,4]}[[1,2]]
```

11.
```
{'a','b','c'}['a']
```

12.
```
{'a:b','c:d'}['c']
```

13.
```
{'a':4,'b':5}{'a'}
```

14.
```
d1 = {'a':'b'}
d2 = {'b':'c'}
print(d1[d2['c']])
```

15.
```
d1 = {'a':'b'}
d2 = {'b':'c'}
print(d2[d1['a']])
```

16.
```
{}[]
```

17.
```
{[]:3}[[]]
```

18.
```
{1:7}['1']
```

19.
```
{'':7}"[]"
```

20.
```
{'':7}[""]
```

21.
```
{"":7}['']
```

22.
```
{'"':()}['']
```

23.
```
{():7}[()]
```

24. `{(()):7}[()]`

25. `{(()):7}[((),)]`

Exercise - z7

⊛ Given a dictionary `d1` with keys `'b'` and `'c'` and integer values, create a dictionary `d2` containing the key `'z'` and associate to it the sum of values of keys from `d1`

- your code must work for *any* `d1` with keys `'b'` and `'c'`

Example - given:

```
d1 = {'a':6, 'b':2,'c':5}
```

After your code, it must result:

```
>>> print(d2)
{'z': 7}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[8]:
```
d1 = {'a':6, 'b':2,'c':5}

# write here

d2 = {'z' : d1['b'] + d1['c']}

print(d2)
```

```
{'z': 7}
```

</div>

[8]:
```
d1 = {'a':6, 'b':2,'c':5}

# write here
```

```
{'z': 7}
```

5.12.5 Writing in the dictionary

Can we write in a dictionary?

> **Dictionaries are mutable containers** which allow us to rapidly associate elements called keys to some values

The definition talks about mutability, so we are allowed to modify dictionaries after creation.

Dictionaries are collections of key/value couples, and among the possible modifications we find:

1. adding a key/value couple
2. associate an existing key to a different value

3. remove a key/value couple

Writing - adding key/value

Suppose we created our dictionary `furniture`:

[9]:
```
furniture = {
    'chair'    : 'a piece of furniture to sit on',
    'cupboard' : 'a cabinet for storage',
    'lamp'     : 'a device to provide illumination'
}
```

and afterwards we want to add a definition for `'armchair'`. We can reuse the variable `furniture` followed by square brackets with inside the key we want to add `['armchair']` and after the brackets we will put an equality sign `=`

[10]:
```
furniture['armchair'] = 'a chair with armrests'
```

Note Jupyter didn't show results, because the previous operation is an assignment *command* (only *expressions* generate results).

But something did actually happen in memory, we can check it by `furniture`:

[11]:
```
furniture
```

[11]:
```
{'armchair': 'a chair with armrests',
 'chair': 'a piece of furniture to sit on',
 'cupboard': 'a cabinet for storage',
 'lamp': 'a device to provide illumination'}
```

Note the dictionary associated to the variable `furniture` was MODIFIED with the addition of `'armchair'`.

When we add a key/value couple, we can use heterogenous types:

[12]:
```
trashcan = {
    'bla' : 3,
     4    : 'boh',
    (7,9) : ['gar','bage']
}
```

[13]:
```
trashcan[5.0] = 'a float'
```

[14]:
```
trashcan
```

[14]:
```
{(7, 9): ['gar', 'bage'], 4: 'boh', 5.0: 'a float', 'bla': 3}
```

We are subject to the same constraints on keys we have during the creation, so we can only use *immutable* keys. If we try inserting a *mutable* type, for example a list, we will get an error:

```
>>> trashcan[ ['some', 'list']  ] = 8

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-195ac9c21bcd> in <module>
----> 1 trashcan[ ['some', 'list']  ] = 8

TypeError: unhashable type: 'list'
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if gives an error):

1. ```
   d = {1:'a'}
   d[2] = 'a'
   print(d)
   ```

2. ```
   d = {}
   print(len(d))
   d['a'] = 'b'
   print(len(d))
   ```

3. ```
   d1 = {'a':3, 'b':4}
   diz2 = diz1
   diz1['a'] = 5
   print(diz1)
   print(diz2)
   ```

4. ```
   diz1 = {'a':3, 'b':4}
   diz2 = dict(diz1)
   diz1['a'] = 5
   print(diz1)
   print(diz2)
   ```

5. ```
   la = ['a','c']
   diz = {'a':3,
          'b':4,
          'c':5}
   diz['d'] = diz[la[0]] + diz[la[1]]
   print(diz)
   ```

6. ```
   diz = {}
   diz[()]: ''
   diz[('a',)]: 'A'
   diz[('a','b')]: 'AB'
   print(diz)
   ```

7. ```
   la = [5,8,6,9]
   diz = {}
   diz[la[0]]=la[2]
   diz[la[2]]=la[0]
   print(diz)
   ```

8. ```
   diz = {}
   diz[(4,5,6)[2]] = 'c'
   diz[(4,5,6)[1]] = 'b'
   diz[(4,5,6)[0]] = 'a'
   print(diz)
   ```

9. ```
   diz1 = {
       'a' : 'x',
       'b' : 'x',
       'c' : 'y',
       'd' : 'y',
   }
   ```

(continues on next page)
   ```

(continued from previous page)

```
diz2 = {}
diz2[diz1['a']] = 'a'
diz2[diz1['b']] = 'b'
diz2[diz1['c']] = 'c'
diz2[diz1['d']] = 'd'
print(diz2)
```

Writing - reassociate a key

Let's suppose to change the definition of a `lamp`:

[15]:
```
furniture = {'chair':'a piece of furniture to sit on',
             'cupboard':'a cabinet for storage',
             'lamp': 'a device to provide illumination'
}
```

[16]:
```
furniture['lamp'] = 'a device to provide visible light from electric current'
```

[17]:
```
furniture
```

[17]:
```
{'chair': 'a piece of furniture to sit on',
 'cupboard': 'a cabinet for storage',
 'lamp': 'a device to provide visible light from electric current'}
```

Exercise - workshop

⊛ MODIFY the dictionary `workshop`:

1. set the `'bolts'` key value equal to the value of the `'pincers'` key
2. increment the value of `wheels` key of `1`

- your code must work with any number associated to the keys
- **DO NOT** create new dictionaries, so no lines beginning with `workshop = {`

Example - given:

```
workshop = {'wheels':3,
            'bolts':2,
            'pincers':5}
```

after your code, you should obtain:

```
>>> print(workshop)
{'bolts': 5, 'wheels': 4, 'pincers': 5}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```
workshop = {'wheels' : 3,
            'bolts'  : 2,
            'pincers': 5}

# write here
```

(continues on next page)

(continued from previous page)

```
workshop['wheels'] = workshop['wheels'] + 1
workshop['bolts'] = workshop['pincers']
#print(workshop)
```

</div>

[18]:
```
workshop = {'wheels' : 3,
            'bolts'  : 2,
            'pincers': 5}

# write here
```

QUESTION: Look at the following code fragments expressions, and for each try guessing the result it produces (or if it gives an error):

1. ```
   diz = {'a':'b'}
   diz['a'] = 'a'
   print(diz)
   ```

2. ```
   diz = {'1':'2'}
   diz[1] = diz[1] + 5    # nasty
   print(diz)
   ```

3. ```
   diz = {1:2}
   diz[1] = diz[1] + 5
   print(diz)
   ```

4. ```
   d1 = {1:2}
   d2 = {2:3}
   d1[1] = d2[d1[1]]
   print(d1)
   ```

Writing - deleting

To remove a key/value couple the special command `del` is provided. Let's take a dictionary:

[19]:
```
kitchen = {
    'pots'  : 3,
    'pans'  : 7,
    'forks' : 20
}
```

If we want to eliminate the couple `pans : 7`, we will write `del` followed by the name of the dictionary and the key to eliminate among square brackets:

[20]:
```
del kitchen['pans']
```

[21]:
```
kitchen
```

[21]:
```
{'forks': 20, 'pots': 3}
```

Trying to delete a non-existemt key will produce an error:

```
>>> del cucina['crankshaft']

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-34-c0d541348698> in <module>
----> 1 del cucina['crankshaft']

KeyError: 'crankshaft'
```

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1.
```
diz = {'a':'b'}
del diz['b']
print(diz)
```

2.
```
diz = {'a':'b', 'c':'d'}
del diz['a']
print(diz)
```

3.
```
diz = {'a':'b', 'c':'d'}
del diz['a']
del diz['a']
print(diz)
```

4.
```
diz = {'a':'b'}
new_diz = del diz['a']
print(diz)
print(new_diz)
```

5.
```
diz1 = {'a':'b', 'c':'d'}
diz2 = diz1
del diz1['a']
print(diz1)
print(diz2)
```

6.
```
diz1 = {'a':'b', 'c':'d'}
diz2 = dict(diz1)
del diz1['a']
print(diz1)
print(diz2)
```

7.
```
diz = {'a':'b'}
del diz['c']
print(diz)
```

8.
```
diz = {'a':'b'}
diz.del('a')
print(diz)
```

9.
```
diz = {'a':'b'}
diz['a'] = None
print(diz)
```

Exercise - desktop

Given a dictionary `desktop`:

```
desktop = {
    'paper'   :5,
    'pencils':2,
    'pens'    :3
}
```

write some code which MODIFIES it so that after executing your code, the dictionary appears like this:

```
>>> print(desktop)
{'pencil sharpeners': 1, 'paper': 5, 'pencils': 2, 'papers': 4}
```

- **DO NOT** write lines which begin with `desktop =`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```
desktop = {
    'paper'   :5,
    'pencils':2,
    'pens'    :3
}

# write here
desktop['papers'] = 4
del desktop['pens']
desktop['pencil sharpeners'] = 1
#print(desktop)
```

</div>

[22]:
```
desktop = {
    'paper'   :5,
    'pencils':2,
    'pens'    :3
}

# write here
```

Exercise - garden

You have a dictionary `garden` which associates the names of present objects and their quantity. You are given:

- a list `to_remove` containing the names of exactly two objects to eliminate
- a dictionary `to_add` containing exactly two names of flowers associated to their quantity to add

MODIFY the dictionary `garden` according to the quantities given in `to_remove` (**deleting the keys**) and `to_add` (**increasing** the corresponding values)

- assume that `garden` always contains the objects given in `to_remove` and `to_add`
- assume that `to_add` always and only contains `tulips` and `roses`

Example - given:

```
to_remove = ['weeds', 'litter']
to_add = { 'tulips':4,
           'roses' : 2
}

garden = { 'sunflowers':3,
           'tulips':7,
           'weeds' : 10,
           'roses' : 5,
           'litter' : 6,
}
```

after your code, it must result:

```
>>> print(garden)
{'roses': 7, 'tulips': 11, 'sunflowers': 3}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```
to_remove = ['weeds', 'litter']
to_add = { 'tulips':4,
           'roses' : 2
}

garden = { 'sunflowers':3,
           'tulips':7,
           'weeds' : 10,
           'roses' : 5,
           'litter' : 6,
}

# write here

del garden[to_remove[0]]
del garden[to_remove[1]]
garden['roses'] = garden['roses'] + to_add['roses']
garden['tulips'] = garden['tulips'] + to_add['tulips']
#print(garden)
```

</div>

[23]:
```
to_remove = ['weeds', 'litter']
to_add = { 'tulips':4,
           'roses' : 2
}

garden = { 'sunflowers':3,
           'tulips':7,
           'weeds' : 10,
           'roses' : 5,
           'litter' : 6,
}

# write here
```

(continues on next page)

(continued from previous page)

```
```

Exercise - translations

Given two dictionaries `en_it` and `it_es` of English-Italian and Italian-Spanish translations, write some code which MODIFIES a third dictionary `en_es` by placing translations from English to Spanish

- assume that `en_it` always and only contains translations of `hello` and `road`
- assume that `it_es` always and only contains translations of `ciao` and `strada`
- in the solution, **ONLY** use the constants `'hello'` and `'road'`, you will take the others you need from the dictionaries
- **DO NOT** create a new dictionary - so no lines beginning with `en_es = {`

Example - given:

```
en_it = {
    'hello' : 'ciao',
    'road' : 'strada'
}

it_es = {
    'ciao' : 'hola',
    'strada' : 'carretera'
}
en_es = {}
```

after your code, it must print:

```
>>> print(en_es)
{'hello': 'hola', 'road': 'carretera'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[24]:
```
en_it = {
    'hello' : 'ciao',
    'road' : 'strada'
}

it_es = {
    'ciao' : 'hola',
    'strada' : 'carretera'
}

en_es = {}

# write here
en_es['hello'] = it_es[en_it['hello']]
en_es['road'] = it_es[en_it['road']]
#print(en_es)
```

</div>

[24]:
```python
en_it = {
    'hello' : 'ciao',
    'road' : 'strada'
}

it_es = {
    'ciao' : 'hola',
    'strada' : 'carretera'
}

en_es = {}

# write here
```

5.12.6 Membership with `in`

We can check whether a *key* is present in a dictionary by using the operator `in`:

[25]:
```python
'a' in {'a':5,'b':7}
```

[25]:
```
True
```

[26]:
```python
'b' in {'a':5,'b':7}
```

[26]:
```
True
```

[27]:
```python
'z' in {'a':5,'b':7}
```

[27]:
```
False
```

WARNING: `in` searches among the *keys*, not in values!

[28]:
```python
5 in {'a':5,'b':7}
```

[28]:
```
False
```

As always when dealing with keys, we *cannot* search for a mutable object, like for example lists:

```
>>> [3,5] in {'a':'c','b':'d'}

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-41-3e3e336117aa> in <module>
----> 1 [3,5] in {'a':'c','b':'d'}

TypeError: unhashable type: 'list'
```

not in

It is possible to check for *non* belonging with the `not in` operator:

```
[29]: 'z' not in {'a':5,'b':7}
```

```
[29]: True
```

```
[30]: 'a' not in {'a':5,'b':7}
```

```
[30]: False
```

Equivalently, we can use this other form:

```
[31]: not 'z' in {'a':5,'b':7}
```

```
[31]: True
```

```
[32]: not 'a' in {'a':5,'b':7}
```

```
[32]: False
```

QUESTION: Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `('a') in {'a':5}`
2. `('a','b') in {('a','b'):5}`
3. `('a','b',) in {('a','b'):5}`
4. `['a','b'] in {('a','b'):5}`
5. `{3: 'q' in {'q':5}}`
6. `{'q' not in {'q':0} : 'q' in {'q':0}}`
7. `{'a' in 'b'}`
8. `{'a' not in {'b':'a'}}`
9. `len({'a':6,'b':4}) in {1:2}`
10. `'ab' in {('a','b'): 'ab'}`
11. `None in {}`
12. `None in {'None':3}`
13. `None in {None:3}`
14. `not None in {0:None}`

5.12.7 Dictionaries of sequences

So far we almost always associated a single value to keys. What if wanted to associate more? For example, suppose we are in a library and we want to associate users with the books they borrowed. We could represent everything as a dictionary where a list of borrowed books is associated to each customer:

[33]:
```
loans = {'Marco':  ['Les Misérables', 'Ulysses'],
         'Gloria': ['War and Peace'],
         'Rita':   ['The Shining','Dracula','1984']}
```

Let's see how it gets represented in Python Tutor:

[34]:
```
# WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#          (it's sufficient to execute it only once)

import jupman
```

[35]:
```
loans = {'Marco':  ['Les Misérables', 'Ulysses'],
         'Gloria': ['War and Peace'],
         'Rita':   ['The Shining','Dracula','1984']}
jupman.pytut()
```

[35]:
```
<IPython.core.display.HTML object>
```

If we try writing the expression:

[36]:
```
loans['Rita']
```

[36]:
```
['The Shining', 'Dracula', '1984']
```

Python shows the corresponding list: for all intents and purposes Python considers `loans['Rita']` as if it were a list, and we can use it as such. For example, if we wanted to access the 1-indexed book of the list, we would write `[1]` after the expression:

[37]:
```
loans['Rita'][1]
```

[37]:
```
'Dracula'
```

Equivalently, we might also save a pointer to the list by assigning the expression to a variable:

[38]:
```
ritas_list = loans['Rita']
```

[39]:
```
ritas_list
```

[39]:
```
['The Shining', 'Dracula', '1984']
```

[40]:
```
ritas_list[1]
```

[40]:
```
'Dracula'
```

Let's see everything in Python Tutor:

[41]:
```
loans = {'Marco':  ['Les Misérables', 'Ulysses'],
         'Gloria': ['War and Peace'],
         'Rita':   ['The Shining','Dracula','1984']}
ritas_list = loans['Rita']
print(rita_list[1])

jupman.pytut()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-41-678825b512c3> in <module>()
      3           'Rita':   ['The Shining','Dracula','1984']}
      4 ritas_list = loans['Rita']
----> 5 print(rita_list[1])
      6 
      7 jupman.pytut()

NameError: name 'rita_list' is not defined
```

If you execute the code in Python Tutor, you will notice that as soon as we assign `ritas_list`, the corresponding list appears to 'detach' from the dictionary. This is only a graphical effect caused by Python Tutor, but from the point of view of the dictionary nothing changed. The intention is to show the list now is *reachable* both from the dictionary and from the new variable `ritas_list`.

5.12.8 Exercise - loans

Write some code to extract and print:

1. The first book borrowed by Gloria (`'War and Peace'`) and the last one borrowed by Rita (`'1984'`)
2. The number of books borrowed by Rita
3. `True` if everybody among Marco, Gloria and Rita borrowed at least a book, `False` otherwise

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[42]:
```python
loans = {'Marco':  ['Les Misérables', 'Ulysses'],
         'Gloria': ['War and Peace'],
         'Rita':   ['The Shining','Dracula','1984']}

# write here
print("1. The first book borrowed by Gloria is", loans['Gloria'][0])
print("   The last book borrowed by Rita is", loans['Rita'][-1])
print("2. Rita borrowed", len(loans['Rita']), "book(s)")
res = len(loans['Marco']) > 0 and len(loans['Gloria']) > 0 and len(loans['Rita']) > 0
print("3. Have everybody borrowed at least a book?", res)
```

```
1. The first book borrowed by Gloria is War and Peace
   The last book borrowed by Rita is 1984
2. Rita borrowed 3 book(s)
3. Have everybody borrowed at least a book? True
```

</div>

[42]:
```python
loans = {'Marco':  ['Les Misérables', 'Ulysses'],
         'Gloria': ['War and Peace'],
         'Rita':   ['The Shining','Dracula','1984']}

# write here
```

```
1. The first book borrowed by Gloria is War and Peace
   The last book borrowed by Rita is 1984
```

(continues on next page)

(continued from previous page)

```
2. Rita borrowed 3 book(s)
3. Have everybody borrowed at least a book? True
```

5.12.9 Equality

We can verify whether two dictionaries are equal with == operator, which given two dictionaries return `True` if they contain kequal ey/value couples or `False` otherwise:

```
[43]: {'a':3, 'b':4} == {'a':3, 'b':4}
```

```
[43]: True
```

```
[44]: {'a':3, 'b':4} == {'c':3, 'b':4}
```

```
[44]: False
```

```
[45]: {'a':3, 'b':4} == {'a':3, 'b':999}
```

```
[45]: False
```

We can verify equality of dictionaries with a different number of elements:

```
[46]: {'a':3, 'b':4} == {'a':3}
```

```
[46]: False
```

```
[47]: {'a':3, 'b':4} == {'a':3,'b':3,'c':5}
```

```
[47]: False
```

… and with heterogenous elements:

```
[48]: {'a':3, 'b':4} == {2:('q','p'), 'b':[99,77]}
```

```
[48]: False
```

Equality and order

From the definition:

- Keys are immutable, **don't have order** and there cannot be duplicates

Since order has no importance, dictionaries created by inserting the same key/value couples in a differenct order will be considered equal.

For example, let's try direct creation:

```
[49]: {'a':5, 'b':7} == {'b':7, 'a':5}
```

```
[49]: True
```

What about incremental update?

[50]:
```python
diz1 = {}
diz1['a'] = 5
diz1['b'] = 7

diz2 = {}
diz2['b'] = 7
diz2['a'] = 5

print(diz1 == diz2)
```

```
True
```

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1. ```python
   {1:2} == {2:1}
   ```
2. ```python
   {1:2,3:4} == {3:4,1:2}
   ```
3. ```python
   {'a'.upper():3} == {'a':3}
   ```
4. ```python
   {'A'.lower():3} == {'a':3}
   ```
5. ```python
   {'a': {1:2} == {3:4}}
   ```
6. ```python
   diz1 = {}
   diz1[2] = 5
   diz1[3] = 7

   diz2 = {}
   diz2[3] = 7
   diz2[2] = 5
   print(diz1 == diz2)
   ```
7. ```python
   diz1 = {'a':3,'b':8}
   diz2 = diz1
   diz1['a'] = 7
   print(diz1 == diz2)
   ```
8. ```python
   diz1 = {}
   diz1['a']=3
   diz2 = diz1
   diz2['a']=4
   print(diz1 == diz2)
   ```
9. ```python
   diz1 = {}
   diz1['a']=3
   diz2 = diz1
   diz2['a']=4
   print(diz1 == diz2)
   ```
10. ```python
    diz1 = {'a':3, 'b':4, 'c':5}
    diz2 = {'a':3,'c':5}
    del diz1['a']
    print(diz1 == diz2)
    ```

11.
```
diz1 = {}
diz2 = {'a':3}
diz1['a'] = 3
diz1['b'] = 5
diz2['b'] = 5
print(diz1 == diz2)
```

Equality and copies

When duplicating containers which hold mutable objects, if we do not pay attention we might get surprises. Let's go back on the topic of shallow and deep copies of dictionaries, this time trying to verify the effective equality in Python.

> **WARNING: Have you read Dictionaries 1 - Copying a dictionary[240] ?**
>
> If not, do it now!

QUESTION: Let's see a simple example, with a 'manual' copy. If you execute the following code in Python Tutor, what will it print? How many memory regions will you see?

```
d1 = {'a':3,
      'b':8}
d2 = {'a':d1['a'],
      'b':d1['b'] }
d1['a'] = 6

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)
```

NOTE: all values (3 and 8) are **immutable**.

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: In this case we manually created a dictionary `d2` using *immutable* values taken from `d1`. So in Python Tutor we will see two distinct memory regions and a successive modification to `d1` will not alter `d2`:

</div>

[51]:
```
d1 = {'a':3,
      'b':8}
d2 = {'a':d1['a'],
      'b':d1['b'] }
d1['a'] = 6

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)

jupman.pytut()
```

```
equal? False
d1= {'a': 6, 'b': 8}
d2= {'a': 3, 'b': 8}
```

240 https://en.softpython.org/dictionaries/dictionaries1-sol.html#Copying-a-dictionary

[51]:
```
<IPython.core.display.HTML object>
```

QUESTION: If you execute the following code in Python Tutor, what will it print?

1. Which type of copy did we do? Shallow? Deep? (or both ...?)
2. How many memory regions will you see?

```python
d1 = {'a':3,
      'b':8}
d2 = dict(d1)
d1['a'] = 7

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: when used as a function, `dict` executes a *shallow* copy, that is, copies the structure of the dictionary without duplicating the mutable values. In this specific case, all values we have are immutable integers, so the copy can also be considered a complete duplication. When we assign the value `7` to the key `'a'` in `d1` we are modifying the original data structure , leaving the copy we just made `d2` unaltered, so `d1 == d2` will be `False`.

Let's verify it in Python Tutor:

</div>

[52]:
```python
d1 = {'a':3,
      'b':8}
d2 = dict(d1)
d1['a'] = 7

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)

jupman.pytut()
```

```
equal? False
d1= {'a': 7, 'b': 8}
d2= {'a': 3, 'b': 8}
```

[52]:
```
<IPython.core.display.HTML object>
```

QUESTION: If you execute the following code in Python Tutor, what will it print?

1. Which type of copy did we do? Shallow? Deep? (or both ...?)
2. How many memory regions will you see?

NOTE: the values are lists, thus they are **mutable**

```python
d1 = {'a':[1,2],
      'b':[4,5,6]}
d2 = dict(d1)
d1['a'].append(3)

print('equal?', d1 == d2)
```

(continues on next page)

(continued from previous page)

```
print('d1=', d1)
print('d2=', d2)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: We used `dict` like a function, so we did a *shallow copy*. In this case we have lists as values, which are *mutable* objects. This means the shallow copy only copied references to the lists, but *not* the lists themselves. For this reason you will see arrows going from the copy of the dictionary `d2` to memory regions of the original lists. This means that if you try to modify a list after the copy occurred (for example with the method `.append(3)`), as a matter of fact you will also modify the list reachable from the copied dictionary `d2`. Let's check this out in Python Tutor:

</div>

[53]:
```
d1 = {'a':[1,2],
      'b':[4,5,6]}
d2 = dict(d1)
d1['a'].append(3)

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)

jupman.pytut()
```

```
equal? True
d1= {'a': [1, 2, 3], 'b': [4, 5, 6]}
d2= {'a': [1, 2, 3], 'b': [4, 5, 6]}
```

[53]:
```
<IPython.core.display.HTML object>
```

QUESTION: If you execute the following code in Python Tutor, what will it print?

1. Which type of copy did we do? Shallow? Deep? (or both …?)
2. How many memory regions will you see?

NOTE: the values are lists, so they are **mutable**

```
import copy
d1 = {'a':[1,2],
      'b':[4,5,6]}
d2 = copy.deepcopy(d1)
d1['a'].append(3)

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: We used `copy.deepcopy`, making an in-depth copy. In this case we have mutable lists as values. The deep copy duplicated all the objects it was able to reach, lists included. So in this case we will obtain two completely distinct memory regions. After the copy, if we modify a list reachable from the original `d1`, we will be sure that we cannot tarnish objects reachable from `d2`. Let's check it in Python Tutor:

</div>

[54]:
```python
import copy
d1 = {'a':[1,2],
      'b':[4,5,6]}
d2 = copy.deepcopy(d1)
d1['a'].append(3)

print('equal?', d1 == d2)
print('d1=', d1)
print('d2=', d2)

jupman.pytut()
```

```
equal? False
d1= {'a': [1, 2, 3], 'b': [4, 5, 6]}
d2= {'a': [1, 2], 'b': [4, 5, 6]}
```

[54]:
```
<IPython.core.display.HTML object>
```

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1. ```python
   diz1 = {'a':[4,5],
           'b':[6,7]}
   diz2 = dict(diz1)
   diz2['a'] = diz1['b']
   diz2['b'][0] = 9
   print(diz1 == diz2)
   print(diz1)
   print(diz2)
   ```

2. ```python
   da = {'a':['x','y','z']}
   db = dict(da)
   db['a'] = ['w','t']
   dc = dict(db)
   print(da)
   print(db)
   print(dc)
   ```

3. ```python
   import copy

   la = ['x','y','z']
   diz1 = {'a':la,
           'b':la }
   diz2 = copy.deepcopy(diz1)
   diz2['a'][0] = 'w'
   print('uguali?', diz1 == diz2)
   print('diz1=', diz1)
   print('diz2=', diz2)
   ```
   ```

5.12.10 Exercise - ZOOM DOOM

Write some code which given a string `s` (i.e. `'ZOOM'`), creates a dictionary `zd` and assigns to keys `'a'`, `'b'` and `'c'` the *same identical list* containing the string characters as elements (i.e. `['Z','O','O','M']`).

- in Python Tutor you should see 3 arrows which go from keys to *the same identical memory region*
- by modifying the list associated to each key, you should see the modification also in the lists associated to other keys
- your code must work for *any* string `s`

Example - given:

```
s = 'ZOOM'
```

After your code, it should result:

```
>>> print(zd)
{'a': ['Z', 'O', 'O', 'M']
 'b': ['Z', 'O', 'O', 'M'],
 'c': ['Z', 'O', 'O', 'M'],
}
>>> zd['a'][0] = 'D'
>>> print(zd)
{'a': ['D', 'O', 'O', 'M']
 'b': ['D', 'O', 'O', 'M'],
 'c': ['D', 'O', 'O', 'M'],
}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[55]:
```
s = 'ZOOM'

# write here

zoom = list(s)

zd = {'a':zoom,
      'b':zoom,
      'c':zoom}
print(zd)
zd['a'][0] = 'D'
print(zd)

#jupman.pytut()
```

```
{'c': ['Z', 'O', 'O', 'M'], 'a': ['Z', 'O', 'O', 'M'], 'b': ['Z', 'O', 'O', 'M']}
{'c': ['D', 'O', 'O', 'M'], 'a': ['D', 'O', 'O', 'M'], 'b': ['D', 'O', 'O', 'M']}
```

</div>

[55]:
```
s = 'ZOOM'

# write here
```

```
{'c': ['Z', 'O', 'O', 'M'], 'a': ['Z', 'O', 'O', 'M'], 'b': ['Z', 'O', 'O', 'M']}
{'c': ['D', 'O', 'O', 'M'], 'a': ['D', 'O', 'O', 'M'], 'b': ['D', 'O', 'O', 'M']}
```

5.12.11 Continue

Go on reading Dictionaries 3[241]

```
[ ]:
```

5.13 Dictionaries 3 - Methods

5.13.1 Download exercise zip

Browse online files[242]

In this tutorial we will see the main methods to retrieve stuff from dictionaries and to manipulate them, along with some special classes.

Methods:

Method	Return	Description
`dict.keys()`	`dict_keys`	Return a *view* of keys which are present in the dictionary
dict.values()	`dict_values`	Return a *view* of values which are present in the dictionary
dict.items()	`dict_items`	Return a *view* of (key/value) couples present in the dictionary
d1.update(d2)	`None`	MODIFY the dictionary `d1` with the key / value couples found in `d2`

Classes:

Class	Description
OrderedDict[243]	Dictionary which allows to maintain the order of insertion of keys
Counter[244]	Dictionary which allows to rapidly calculate histograms

5.13.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
sets
    dictionaries1.ipynb
    dictionaries1-sol.ipynb
    dictionaries2.ipynb
    dictionaries2-sol.ipynb
    dictionaries3.ipynb
    dictionaries3-sol.ipynb
    dictionaries4.ipynb
```

(continues on next page)

241 https://en.softpython.org/dictionaries/dictionaries3-sol.html
242 https://github.com/DavidLeoni/softpython-en/tree/master/dictionaries
243 https://docs.python.org/3/library/collections.html#collections.OrderedDict
244 https://docs.python.org/3/library/collections.html#collections.Counter

(continued from previous page)

```
    dictionaries4-sol.ipynb
    dictionaries5.ipynb
    dictionaries5-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `dictionaries3.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.13.3 `keys` method

By calling the method `.keys()` we can obtain the dictionary keys:

```
[2]: vegetables = {'carrots'  : 5,
                   'tomatoes' : 8,
                   'cabbage'  : 3}
```

```
[3]: vegetables.keys()
```

```
[3]: dict_keys(['carrots', 'cabbage', 'tomatoes'])
```

WARNING: THE RETURNED SEQUENCE IS OF TYPE `dict_keys`

`dict_keys` might look like a list but it is well different!

In particular, the returned sequence `dict_keys` is **a view** on the original dictionary. In computer science, when we talk about *views* we typically intend collections which contain a part of the objects contained in another collection, *and if the original collection gets modified, so is the view at the same time.*

Let's see what this means. First let's assign the sequence of keys to a variable:

```
[4]: ks = vegetables.keys()
```

Then we modify the original dictionary, adding an association:

```
[5]: vegetables['potatoes'] = 8
```

If we now print `ks`, we should see the change:

```
[6]: ks
```

```
[6]: dict_keys(['carrots', 'potatoes', 'cabbage', 'tomatoes'])
```

Sequence returned by `.keys()` can change over time!

When reusing the sequence from `.keys()`, ask yourself if the dictionary could have changed in the meanwhile

If we want a stable version as a sort of static 'picture' of dictionary keys at a given moment in time, we must explicitly convert them to another sequence, like for example a `list`:

```
[7]: as_list = list(vegetables.keys())
```

```
[8]: as_list
```

```
[8]: ['carrots', 'potatoes', 'cabbage', 'tomatoes']
```

```
[9]: vegetables['cocumber'] = 9
```

```
[10]: as_list      # no cocumbers
```

```
[10]: ['carrots', 'potatoes', 'cabbage', 'tomatoes']
```

Let's see again the example in Python Tutor:

```
[11]: # WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#           (it's sufficient to execute it only once)

import jupman
```

```
[12]: vegetables = {'carrots'  : 5,
              'tomatoes' : 8,
              'cabbage'  : 3}
keys = vegetables.keys()
vegetables['potatoes'] = 8
as_list = list(vegetables.keys())
vegetables['cocumbers'] = 9
#print(as_list)

jupman.pytut()
```

```
[12]: <IPython.core.display.HTML object>
```

WARNING: WE CAN'T USE INDEXES WITH `dict_keys`

If we try, we will obtain an error:

```
>>> vegetables = {'carrots'  : 5,
                  'tomatoes' : 8,
                  'cabbage'  : 3}
>>> ks = vegetables.keys()
>>> ks[0]

---------------------------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
TypeError                                 Traceback (most recent call last)
<ipython-input-90-c888bf602918> in <module>()
----> 1 keys[0]

TypeError: 'dict_keys' object does not support indexing
```

WARNING: WE CANNOT DIRECTLY MODIFY `dict_keys`

There aren't operations nor methods which allow us to change the elements of `dict_keys`, you can only act on the original dictionary.

QUESTION: Look at the following code fragments, and for each try guessing if it can work (or if it gives an error):

1. ```
   diz = {'a':4,
          'b':5}

   ks = diz.keys()
   ks.append('c')
   ```

2. ```
   diz = {'a':4,
          'b':5}

   ks = diz.keys()
   ks.add('c')
   ```

3. ```
   diz = {'a':4,
          'b':5}

   ks = diz.keys()
   ks['c'] = 3
   ```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: None of the examples above can work, because we can't directly modify objects of type `dict_keys`. Operators like square brackets or methods like `.append`, `.add`, etc are not supported.

</div>

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1. ```
   diz = {'a':1,'b':2}
   s = set(diz.keys())
   s.add(('c',3))
   print(diz)
   print(s)
   ```

2. ```
   diz = {'a':3,'b':4}
   k = diz.keys()
   diz['c'] = 5
   print(len(k))
   ```
   ```

3.
```python
diz = {'a':'x',
       'b':'y'}
print('a' in diz.keys())
```

4.
```python
diz1 = {'a':1,'b':2}
chiavi = diz1.keys()
diz2 = dict(diz1)
diz2['c'] = 3
print('diz1=',diz1)
print('diz2=',diz2)
print('chiavi=',chiavi)
```

5.
```python
diz1 = {'a':'b','c':'d'}
diz2 = {'a':'b','b':'c'}
print( set(diz1.keys()) - set(diz2.keys())   )
```

6.
```python
diz1 = {'a':'b','c':'d'}
diz2 = {'e':'a','f':'c'}
ks = diz1.keys()
del diz1[diz2['e']]
del diz1[diz2['f']]
print(len(ks))
```

Exercise - messy keys

⊛ PRINT a LIST with all the keys in the dictionary

- **NOTE 1**: it is NOT necessary for the list to be sorted
- **NOTE 2**: to convert any sequence to a list, use the predefined function `list`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[13]:
```python
d = {'c':6, 'b':2,'a':5}

# write here

list(d.keys())
```

[13]:
```
['b', 'c', 'a']
```

</div>

[13]:
```python
d = {'c':6, 'b':2,'a':5}

# write here

```

[13]:
```
['b', 'c', 'a']
```

Exercise - sorted keys

⊕ PRINT a LIST with all the dictionary keys

- **NOTE 1**: Now it IS necessary for the list to be sorted
- **NOTE 2**: to convert any sequence to a list, use the predefined function `list`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[14]:
```python
d = {'c':6, 'b':2,'a':5}

# write here

my_list = list(d.keys())
my_list.sort()
print(my_list)
```

```
['a', 'b', 'c']
```

</div>

[14]:
```python
d = {'c':6, 'b':2,'a':5}

# write here

```

```
['a', 'b', 'c']
```

Exercise - keyring

Given the dictionaries `d1` and `d2`, write some code which puts into a **list** `ks` all the keys in the two dictionaries, **without duplicates** and **alphabetically sorted**, and finally prints the list.

- your code must work with any `d1` and `d2`

Example - given:

```python
d1 = {
    'a':5,
    'b':9,
    'e':2,
}
d2 = {'a':9,
      'c':2,
      'e':2,
      'f':6}
```

after your code, it must result:

```python
>>> print(keys)
['a', 'b', 'c', 'e', 'f']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[15]:
```
d1 = {
    'a':5,
    'b':9,
    'e':2,
}
d2 = {'a':9,
      'c':2,
      'e':2,
      'f':6}

# write here
ks = list(set(d1.keys()) | set(d2.keys()))
ks.sort()
#print(ks)
```

</div>

[15]:
```
d1 = {
    'a':5,
    'b':9,
    'e':2,
}
d2 = {'a':9,
      'c':2,
      'e':2,
      'f':6}

# write here
```

5.13.4 `values` method

Given a dictionary, we can obtain all the values by calling the method `.values()`

Imagine we have a dictionary `vehicles` which assigns an owner to each car plate:

[16]:
```
vehicles = {
    'AA111AA' : 'Mario',
    'BB222BB' : 'Lidia',
    'CC333CC' : 'Mario',
    'DD444DD' : 'Gino',
    'EE555EE' : 'Gino'
}

owners = vehicles.values()
```

> **WARNING: THE RETURNED SEQUENCE IS OF TYPE** `dict_values`
>
> `dict_values` may seem a list but it's not!

We've seen `dict_keys` is a view on the original dictionary, and so is `dict_values`, thus by adding an association to `vehicles` …

```
[17]: vehicles['FF666FF'] = 'Paola'
```

… the view `owners` will automatically result changed:

```
[18]: owners
```

```
[18]: dict_values(['Lidia', 'Mario', 'Gino', 'Paola', 'Mario', 'Gino'])
```

We also note that being *values* of a dictionary, duplicates are allowed.

WARNING: WE CANNOT USE INDEXES WITH `dict_values`

If we try, we will get an error:

```
>>> owners[0]

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-90-c888bf602918> in <module>()
----> 1 owners[0]

TypeError: 'dict_values' object does not support indexing
```

WARNING: WE CANNOT DIRECTLY MODIFY `dict_values`

There aren't operations nor methods that allow us to change the elements of `dict_values`, we can only act on the original dictionary.

QUESTION: Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1.
```
diz = {'a':4,
       'b':5}

vals = diz.values()
vals.append(4)
```

2.
```
d = {0:'a',
     1:'b',
     2:'b'}
vals = d.values()
d[2]='c'
print(vals)
```

3.
```
diz = {'a':4,
       'b':5}

vals = diz.values()
vals.add(5)
```

4.
```
diz = {0:1,
       1:2,
       2:3}
```

(continues on next page)

(continued from previous page)

```
diz[list(diz.values())[0]-1]
```

5. ```
   diz = {'a':4,
          'b':5}

   vals = diz.values()
   vals['c'] = 6
   ```

6. ```
   diz = {'a':4,
          'b':5}

   vals = diz.values()
   vals[6] = 'c'
   ```

5.13.5 Exercise - one by one

Given a dictionary `my_dict`, write some code which prints `True` if each key is associated to a value *different* from the values of all other keys. Otherwise prints `False`.

Example 1 - given:

```
my_dict = {'a' : 3,
           'c' : 6,
           'g' : 8}
```

After your code, it must print `True` (because `3`,`6` and `8` are all different)

```
True
```

Example 2 - given:

```
my_dict = {'x' : 5,
           'y' : 7,
           'z' : 5}
```

it must print:

```
False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[19]: my_dict = {'a' : 3,
                 'c' : 6,
                 'g' : 8}

      """
      my_dict= {'x' : 5,
                'y' : 7,
                'z' : 5}
      """

      # write here
```

(continues on next page)

(continued from previous page)

```
print(len(my_dict.keys()) == len(set(my_dict.values())))
```

```
True
```

</div>

[19]:
```
my_dict = {'a' : 3,
           'c' : 6,
           'g' : 8}

"""
my_dict= {'x' : 5,
          'y' : 7,
          'z' : 5}
"""

# write here
```

```
True
```

5.13.6 Exercise - bag

Given a dictionary `my_dict` of character associations, write some code which puts into the variable `bag` the sorted list of all the keys and values.

Example - given:

```
my_dict = {
    'a':'b',
    'b':'f',
    'c':'b',
    'd':'e'
}
```

After your code, it must print:

```
>>> print(bag)
['a', 'b', 'c', 'd', 'e', 'f']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[20]:
```
my_dict = {
    'a':'b',
    'b':'f',
    'c':'b',
    'd':'e'
}

# write here

bag = list(set(my_dict.keys()) | set(my_dict.values()))
bag.sort()
```

(continues on next page)

(continued from previous page)

```
print(bag)
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

</div>

[20]:
```
my_dict = {
    'a':'b',
    'b':'f',
    'c':'b',
    'd':'e'
}

# write here
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

Exercise - common values

Given two dictionaries `d1` and `d2`, write some code which PRINTS `True` if they have *at least* a value in common (without considering the keys)

Example 1 - given:

```
d1 = {
    'a':4,
    'k':2,
    'm':5
}

d2 = {
    'b':2,
    'e':4,
    'g':9,
    'h':1
}
```

after your code, it must print `True` (because they have the values `2` and `4` in common):

```
Common values? True
```

Example 2 - given:

```
d1 = {
    'd':1,
    'e':2,
    'f':6
}

d2 = {
    'a':3,
    'b':5,
    'c':9,
```

(continues on next page)

(continued from previous page)

```
    'd':7
}
```

after your code, it must print:

```
Common values? False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[21]:
```python
d1 = {
    'a':4,
    'k':2,
    'm':5
}

d2 = {
    'b':2,
    'e':4,
    'g':9,
    'h':1
}

"""
d1 = {
    'd':1,
    'e':2,
    'f':6
}

d2 = {
    'a':3,
    'b':5,
    'c':9,
    'd':7
}
"""

# write here

print('Common values?',len(set(d1.values()) & set(d2.values())) > 0)
```

```
Common values? True
```

</div>

[21]:
```python
d1 = {
    'a':4,
    'k':2,
    'm':5
}

d2 = {
    'b':2,
    'e':4,
    'g':9,
    'h':1
```

(continues on next page)

(continued from previous page)

```
}

"""
d1 = {
     'd':1,
     'e':2,
     'f':6
}

d2 = {
     'a':3,
     'b':5,
     'c':9,
     'd':7
}
"""

# write here
```

```
Common values? True
```

Exercise - small big

Given a dictionary `d` which has integers as keys and values, print `True` if the smaller key is equal to the greatest value.

Example 1 - given:

```
d = {
    14:1,
    11:7,
     7:3,
    70:5
}
```

after your code, it must print `True` (because the smallest key is `7` which is equal to the greates value `7`):

```
True
```

Example 2 - given:

```
d = {
    12:1,
    11:9,
     7:3,
     2:5,
     9:1
}
```

after your code, it must print `False` (because the smallest key `2` is different from the greatest value`9`):

```
False
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```python
d = {
    14:1,
    11:7,
     7:3,
    70:5
}

"""
d = {
    12:1,
    11:9,
     7:3,
     2:5,
     9:1
}
"""

# write here

min(d.keys()) == max(d.values())
```

[22]:
```
True
```

</div>

[22]:
```python
d = {
    14:1,
    11:7,
     7:3,
    70:5
}

"""
d = {
    12:1,
    11:9,
     7:3,
     2:5,
     9:1
}
"""

# write here
```

[22]:
```
True
```

5.13.7 `items` method

We can extract all the key/value associations as a list of couples of type tuple with the method `.items()`. Let's see an example which associates attractions to the city they are in:

```
[23]: holiday = {'Piazza S.Marco'  :'Venezia',
                 'Fontana di Trevi':'Roma',
                 'Uffizi'          :'Firenze',
                 'Colosseo'        :'Roma',
      }
```

```
[24]: holiday.items()
```

```
[24]: dict_items([('Colosseo', 'Roma'), ('Fontana di Trevi', 'Roma'), ('Uffizi', 'Firenze'),
      ↪ ('Piazza S.Marco', 'Venezia')])
```

In this case we see that an object of type `dict_items` is returned. As in previous cases, it is **a view** which we **can't** directly modify. If the original dictionary gets changed, the mutation will be reflected in the view:

```
[25]: attractions = holiday.items()
```

```
[26]: holiday['Palazzo Ducale'] = 'Venezia'
```

```
[27]: attractions
```

```
[27]: dict_items([('Colosseo', 'Roma'), ('Palazzo Ducale', 'Venezia'), ('Fontana di Trevi',
      ↪'Roma'), ('Uffizi', 'Firenze'), ('Piazza S.Marco', 'Venezia')])
```

QUESTION: Look at the following code fragments, and for each try guessing which result it produces (or if it gives an error):

1. ```
   {'a':7, 'b':9}.items()[0] = ('c',8)
   ```

2. ```
   dict({'a':7,'b':5}.items())['a']
   ```

3. ```
   len(set({'a':'b', 'a':'B'}.items()))
   ```

4. ```
   {'a':2}.items().find(('a',2))
   ```

5. ```
   {'a':2}.items().index(('a',2))
   ```

6. ```
   list({'a':2}.items()).index(('a',2))
   ```

7. ```
   diz1 = {'a':7,
           'b':5}
   diz2 = dict(diz1.items())
   diz1['a'] = 6
   print(diz1 == diz2)
   ```

8. ```
   ('a','b') in {'a':('a','b'), 'b':('a','b')}.items()
   ```

9. ```
   ('a','b') in list({'a':('a','b'), 'b':('a','b')}.items())[0]
   ```
   ```

Exercise - union without update

Given the dictionaries `d1` and `d2`, write some code which creates a NEW dictionary `d3` containing all the key/value couples from `d1` and `d2`.

- we suppose all the key/value couples are distinct
- **DO NOT** use cycles
- **DO NOT** use `.update()`
- your code must work for *any* `d1` and `d2`

Example - given:

```
d1 = {'a':4,
      'b':7}
d2 = {'c':5,
      'd':8,
      'e':2}
```

after your code, it must result (order is not important):

```
>>> print(d3)
{'a': 4, 'e': 2, 'd': 8, 'c': 5, 'b': 7}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[28]:
```
d1 = {'a':4,
      'b':7}
d2 = {'c':5,
      'd':8,
      'e':2}

# write here
diz3 = dict(list(d1.items()) + list(d2.items()))
#print(d3)
```

</div>

[28]:
```
d1 = {'a':4,
      'b':7}
d2 = {'c':5,
      'd':8,
      'e':2}

# write here
```

5.13.8 `update` method

Having a dictionary to start with, it is possibly to MODIFY it by joining another with the method `.update()`:

[29]:
```
d1 = {'goats'     :6,
      'cabbage'   :9,
      'shepherds':1}


d2 = {'goats'   :12,
      'cabbage':15,
      'benches':3,
      'hay'     :7}
```

[30]:
```
d1.update(d2)
```

[31]:
```
d1
```

[31]:
```
{'benches': 3, 'cabbage': 15, 'goats': 12, 'hay': 7, 'shepherds': 1}
```

Note how the common keys among the two dictionaries like `'goats'` and `'cabbage'` have values from the second.

If we will, it's also possible to pass a sequence of couples like this:

[32]:
```
d1.update([('hay',3),('benches',18), ('barns',4)])
```

[33]:
```
d1
```

[33]:
```
{'barns': 4,
 'benches': 18,
 'cabbage': 15,
 'goats': 12,
 'hay': 3,
 'shepherds': 1}
```

5.13.9 Exercise - axby

Given a dictionary `dcc` which associates characters to characters and a string `s` formatted with couples of characters like `ax` separated by a semi-colon `;`, substitute all the values in `dcc` with the corresponding values denoted in the string.

- your code must work for *any* dictionary `my_dict` and lists

Example - given:

```
dcc = {
    'a':'x',
    'b':'y',
    'c':'z',
    'd':'w'
}
s = 'bx;cw;ex'
```

after your code, it must result:

```
>>> dcc
{'a': 'x', 'b': 'x', 'c': 'w', 'd': 'w', 'e': 'x'}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[34]:
```python
dcc = {
    'a':'x',
    'b':'y',
    'c':'z',
    'd':'w'
}
s = 'bx;cw;ex'

# write here

la = s.split(';')
dcc.update(la)
dcc
```

[34]:
```
{'a': 'x', 'b': 'x', 'c': 'w', 'd': 'w', 'e': 'x'}
```

</div>

[34]:
```python
dcc = {
    'a':'x',
    'b':'y',
    'c':'z',
    'd':'w'
}
s = 'bx;cw;ex'

# write here
```

[34]:
```
{'a': 'x', 'b': 'x', 'c': 'w', 'd': 'w', 'e': 'x'}
```

5.13.10 Classes - OrderedDict

As we said before, when we print a dictionary with `print` or we leave the visualization to Jupyter, most of the times couples are not in insertion order. For the order to be predictable, you must use an `OrderedDict`

First you need to import it from the `collections` module:

[35]:
```python
from collections import OrderedDict
```

[36]:
```python
od = OrderedDict()
```

An `OrderedDict` appears and behaves like regular dictionaries:

[37]:
```python
od['some key'] = 5
od['some other key'] = 7
od[('an', 'immutable','tuple', 'as key')] = 3
od['Another key'] = 'now a string!'
od[123] = 'hello'
```

When visualizing with Jupyter, we see the insertion order:

[38]:
```
od
```

[38]:
```
OrderedDict([('some key', 5),
             ('some other key', 7),
             (('an', 'immutable', 'tuple', 'as key'), 3),
             ('Another key', 'now a string!'),
             (123, 'hello')])
```

As we see it with a regular `print`:

[39]:
```
print(od)
```

```
OrderedDict([('some key', 5), ('some other key', 7), (('an', 'immutable', 'tuple',
↪'as key'), 3), ('Another key', 'now a string!'), (123, 'hello')])
```

Let's see how it appears in Python Tutor:

[40]:
```
from collections import OrderedDict
od = OrderedDict()
od['some key'] = 5
od['some other key'] = 7
od[('an', 'immutable','tuple', 'as key')] = 3
od['Another key'] = 'now a string!'
od[123] = 'hello'

jupman.pytut()
```

[40]:
```
<IPython.core.display.HTML object>
```

Exercise - phonebook

Write some code which given three tuples with names and phone numbers, PRINTS an `OrderedDict` which associates names to phone numbers, in the order in which are proposed

- Your code must work with *any* tuple
- Do not forget to import `OrderedDict` from `collections`

Example:

```
t1 = ('Alice', '143242903')
t2 = ('Bob', '417483437')
t3 = ('Carlo', '423413213')
```

after your code, it should result:

```
OrderedDict([('Alice', '143242903'), ('Bob', '417483437'), ('Charles', '423413213')])
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[41]:
```
t1 = ('Alice', '143242903')
t2 = ('Bob', '417483437')
t3 = ('Charles', '423413213')

# write here

# first we need to import some collection
```

(continues on next page)

(continued from previous page)

```
from collections import OrderedDict

od = OrderedDict([t1, t2, t3])
#print(od)
```

</div>

[41]:
```
t1 = ('Alice', '143242903')
t2 = ('Bob', '417483437')
t3 = ('Charles', '423413213')

# write here
```

5.13.11 Exercise - OrderedDict copy

Given an `OrderedDict od1` containing English to Italian translations, create a NEW `OrderedDict` called `od2` which contains the same translations as input PLUS the translation `'water' : 'acqua'`

- NOTE 1: your code should work with any ordered dict as input
- NOTE 2: `od2` MUST be associated to a NEW `OrderedDict` !!

Example - given:

```
od1 = OrderedDict()
od1['dog'] = 'cane'
od1['home'] = 'casa'
od1['table'] = 'tavolo'
```

after your code, you should obtain:

```
>>> print(od1)
OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo')])
>>> print(od2)
OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo'), ('water', 'acqua
↪')])
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[42]:
```
from collections import OrderedDict

od1 = OrderedDict()
od1['dog'] = 'cane'
od1['home'] = 'casa'
od1['table'] = 'tavolo'

# write here
od2 = OrderedDict(od1)
od2['water'] = 'acqua'

print("od1=", od1)
print("od2=", od2)
```

```
od1= OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo')])
od2= OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo'), ('water',
↪'acqua')])
```

</div>

[42]:
```
from collections import OrderedDict

od1 = OrderedDict()
od1['dog'] = 'cane'
od1['home'] = 'casa'
od1['table'] = 'tavolo'

# write here
```

```
od1= OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo')])
od2= OrderedDict([('dog', 'cane'), ('home', 'casa'), ('table', 'tavolo'), ('water',
↪'acqua')])
```

5.13.12 Classes - Counter

If we need to know how many different elements there are in a sequence (in other words, if we need to calculate a frequence histogram), the class `Counter` from `collections` module comes useful. `Counter` is a special type of dictionary, and first of all, we must declare to Python our intention to use it:

[43]:
```
from collections import Counter
```

Suppose we want to count how many different characters there are in this list:

[44]:
```
my_seq = ['t', 'e', 'm', 'p', 'e', 'r', 'a', 'm', 'e', 'n', 't']
```

We can initialize `Counter` like this:

[45]:
```
histogram = Counter(my_seq)
```

If we print it, we see that the first elements are the most frequent:

[46]:
```
print(histogram)
```

```
Counter({'e': 3, 'm': 2, 't': 2, 'n': 1, 'a': 1, 'p': 1, 'r': 1})
```

WARNING: IF WE DON'T USE `print` JUPYTER WILL PRINT IN ALPHABETICAL ORDER!

[47]:
```
histogram    # careful !
```

[47]:
```
Counter({'a': 1, 'e': 3, 'm': 2, 'n': 1, 'p': 1, 'r': 1, 't': 2})
```

We can obtain a list with the `n` most frequent items by using the method `most_common`, which returns a list of tuples:

[48]:
```
histogram.most_common(5)
```

[48]:
```
[('e', 3), ('m', 2), ('t', 2), ('n', 1), ('a', 1)]
```

`Counter` can be initialized with any sequence, for example with tuples:

```
[49]: ct = Counter((50,70,40,60,40,50,40,70,50,50,50,60,50,30,50,30,40,50,60,70))
print(ct)
```

```
Counter({50: 8, 40: 4, 60: 3, 70: 3, 30: 2})
```

or strings:

```
[50]: cs = Counter('condonation')
```

```
[51]: print(cs)
```

```
Counter({'n': 3, 'o': 3, 'a': 1, 'c': 1, 'd': 1, 't': 1, 'i': 1})
```

For other methods we refer to Python documentation[245]

5.13.13 Exercise - saddened

Given a string `s`, write some code which prints:

- the most frequent character
- the least frequent character
- how many and which different frequencies there are
- Your code must work with *any* string `s`
- Ignore the possibility there could be ties among the most/least frequent items
- remember to import `Counter` from `collections`

Example - given:

```
s = 'saddened'
```

your code must print:

```
Among the most frequent ones we find ('d', 3)
Among the least frequent ones we find ('a', 1)
There are 3 different frequencies: {1, 2, 3}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[52]:

s = 'saddened'

# write here
from collections import Counter

c = Counter(s)

print("Among the most frequent ones we find", c.most_common()[0])
print("Among the least frequent ones we find", c.most_common()[-1])
print("There are", len(set(c.values())), "different frequencies:", set(c.values()))
```

[245] https://docs.python.org/3/library/collections.html#collections.Counter

```
Among the most frequent ones we find ('d', 3)
Among the least frequent ones we find ('s', 1)
There are 3 different frequencies: {1, 2, 3}
```

</div>

[52]:

```python
s = 'saddened'

# write here
```

```
Among the most frequent ones we find ('d', 3)
Among the least frequent ones we find ('s', 1)
There are 3 different frequencies: {1, 2, 3}
```

5.13.14 Continue

Go on with Dictionaries 4[246]

[ ]:

```python
```

5.14 Conditionals - if else

5.14.1 Download exercises zip

Browse online files[247]

We can use the conditional command `if` every time the computer must take a decision according to the value of some condition. If the condition is evaluated as true (that is, the boolean `True`), then a code block will be executed, otherwise execution will pass to another one

5.14.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
control-flow
    flow1-if.ipynb
    flow1-if-sol.ipynb
    flow2-for.ipynb
    flow2-for-sol.ipynb
    flow3-while.ipynb
    flow3-while-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

[246] https://en.softpython.org/dictionaries/dictionaries4-sol.html
[247] https://github.com/DavidLeoni/softpython-en/tree/master/control-flow

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `flow1-if.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.14.3 The basic command `if else`

Let's see a small program which takes different decisions according to the value of a variable `sweets`:

[2]:
```
sweets = 20

if sweets > 10 :
    print('We found...')
    print('Many sweets!')
else:
    print("Alas there are.. ")
    print('few sweets!')

print()
print("Let's find other sweets!")
```

```
We found…
Many sweets!

Let's find other sweets!
```

The condition here is `sweets > 10`

[3]:
```
sweets > 10
```

[3]:
```
True
```

WARNING: Right after the condition you must place a colon `:`

```
if sweets > 10:
```

Since in the example above `sweets` is valued `20`, the condition gets evalued to `True` and so the code block following the `if` row gets executed.

Let's try instead to place a small number, like `sweets = 5`:

[4]:
```
sweets = 5

if sweets > 10 :
    print('We found...')
    print('Many sweets!')
```

(continues on next page)

(continued from previous page)

```
else:
    print("Alas there are.. ")
    print('Few sweets!')

print()
print("Let's find other sweets!")
```

```
Alas there are..
Few sweets!

Let's find other sweets!
```

In this case, the code block after the `else:` row got executed

> **WARNING: Careful about block indentation!**
>
> As all code blocks in Python, they are preceded by spaces. Usually there are 4 spaces (in some Python projects you can find only 2, but official Python guidelines recommend 4)

5.14.4 `else` is optional

It is not mandatory to use `else`. If we omit it and the condition becomes `False`, the control directly pass to commands with the same indentation level of `if` (without errors):

[5]:
```
sweets = 5

if sweets > 10 :
    print('We found...')
    print('Many sweets!')

print()
print("Let's find other sweets!")
```

```

Let's find other sweets!
```

QUESTION: Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1.
```
x = 3
if x > 2  and if x < 4:
    print('ABBA')
```

2.
```
x = 3
if x > 2 and x < 4
    print('ABBA')
```

3.
```
x = 3
if x > 2 and x < 4:
    print('ABBA')
```

4.
```
x = 2
if x > 1:
    print(x+1, x):
```

5.
```python
x = 3
if x > 5 or x:
    print('ACDC')
```

6.
```python
x = 7
if x == 7:
print('GLAM')
```

7.
```python
x = 7
if x < 1:
    print('BIM')
else:
    print('BUM')
print('BAM')
```

8.
```python
x = 30
if x > 8:
    print('DOH')
if x > 10:
    print('DUFF')
if x > 20:
    print('BURP')
```

9.
```python
if not True:
    print('upside down')
else:
    print('down upside')
```

10.
```python
if False:
else:
    print('ZORB')
```

11.
```python
if False:
    pass
else:
    print('ZORB')
```

12.
```python
if 0:
    print('Brandy')
else:
    print('Rum')
```

13.
```python
if False:
    print('illustrious')
else:
    print('distinguished')
else:
    print('excellent')
```

14.
```python
if 2 != 2:
    'BE'
else:
    'CAREFUL'
```

15.
```
if 2 != 2:
    print('BE')
else:
    print('CAREFUL')
```

16.
```
x = [1,2,3]
if 4 in x:
    x.append(4)
else:
    x.remove(3)
print(x)
```

17.
```
if 'False':
    print('WATCH OUT FOR THE STRING!')
else:
    print('CRUEL')
```

5.14.5 Exercise - no fuel

You want to do a car trip for which you need at least `30` litres of fuel. Write some code that:

- if the `fuel` variable is less than `30`, prints `'Not enough fuel, I must fill up'` and increments `fuel` of `20` litres
- Otherwise, prints `Enough fuel!`
- In any case, prints at the end `'We depart with '` followed by the final quantity of fuel

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[6]:
```
fuel = 5
#fuel = 30

# write here

if fuel < 30:
    print('Not enough fuel, I must fill up')
    fuel += 20
else:
    print('Enough fuel!')

print('We depart with', fuel, 'litres')
```

```
Not enough fuel, I must fill up
We depart with 25 litres
```

</div>

[6]:
```
fuel = 5
#fuel = 30

# write here
```

```
Not enough fuel, I must fill up
We depart with 25 litres
```

5.14.6 The command *if - elif - else*

By examining the little sweets program we just saw, you may have wondered what it should print when there are no sweets at all. To handle many conditions, we could chain them with the command `elif` (abbreviation of *else if):*

[7]:
```
sweets = 0  # WE PUT ZERO

if sweets > 10:
    print('We found...')
    print('Many sweets!')
elif sweets > 0:
    print("Alas there are.. ")
    print('Few sweets!')
else:
    print("Too bad!")
    print('There are no sweets!')

print()
print("Let's find other sweets!")
```

```
Too bad!
There are no sweets!

Let's find other sweets!
```

EXERCISE: Try changing the values of `sweets` in the above cell and see what happens

The little program behaves exacly like the previous ones and when no condition is satisfied the last code block after the `else` is executed:

We can add as many `elif` as we want, so we could even put a specific `elif x == 0:` and handle in the `else` all other cases, even the unforeseen or absurd ones like for example placing a negative number of sweets. Why should we do it? Accidents can always happen, you surely found a good deal of *bugged* programs in your daily life… (we will see how to better handle these situations in the tutorial Errors handling and testing[248])

[8]:
```
sweets = -2   # LET'S TRY A NEGATIVE NUMBER

if sweets > 10:
    print('We found...')
    print('Many sweets!')
elif sweets > 0:
    print("Alas there are.. ")
    print('Few sweets!')
elif sweets == 0:
    print("Too bad! ")
    print('There are no sweets!')
else:
    print('Something went VERY WRONG! We found', sweets, 'sweets')

print()
print("Let's find other sweets!")
```

[248] https://it.softpython.org/errors-and-testing/errors-and-testing-sol.html

```
Something went VERY WRONG! We found -2 sweets

Let's find other sweets!
```

EXERCISE: Try changing the values of `sweets` in the cell above and see what happens

Questions

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1. ```python
   y = 2
   if y < 3:
       print('bingo')
   elif y <= 2:
       print('bango')
   ```

2. ```python
   z = 'q'
   if not 'quando'.startswith(z):
       print('BAR')
   elif not 'spqr'[2] == z:
       print('WAR')
   else:
       print('ZAR')
   ```

3. ```python
   x = 1
   if x < 5:
       print('SHIPS')
       elif x < 3:
           print('RAFTS')
           else:
               print('LIFEBOATS')
   ```

4. ```python
   x = 5
   if x < 3:
       print('GOLD')
   else if x >= 3:
       print('SILVER')
   ```

5. ```python
   if 0:
       print(0)
   elif 1:
       print(1)
   ```

5.14.7 Questions - Are they equivalent?

Look at the following code fragments: each contains two parts, A and B. For each value of the variables they depend on, try guessing whether part A will print exactly the same result printed by code in part B

- **FIRST** think about the answer
- **THEN** try executing with each of the values of suggested variables
   ```

Are they equivalent? - strawberries

Try changing the value of `strawberries` by removing the comments

[9]:
```
strawberries = 5
#strawberries = 2
#strawberries = 10

print('strawberries =', strawberries)
print('A:')
if strawberries > 5:
    print("The strawberries are > 5")
elif strawberries > 5:
    print("I said the strawberries are > 5!")
else:
    print("The strawberries are <= 5")

print('B:')
if strawberries > 5:
    print("The strawberries are > 5")
if strawberries > 5:
    print("I said the strawberries are > 5!")
if strawberries <= 5:
    print("The strawberries are <= 5")
```

```
strawberries = 5
A:
The strawberries are <= 5
B:
The strawberries are <= 5
```

Are they equivalent? - max

```
x, y = 3, 5
#x, y = 5, 3
#x, y = 3, 3

print('x =', x)
print('y =', y)

print('A:')
if x > y:
    print(x)
else:
    print(y)

print('B:')
print(max(x,y))
```

Are they equivalent? - min

```
x, y = 3, 5
#x, y = 5, 3
#x, y = 3, 3

print('x =',x)
print('y =', y)

print('A:')
if x < y:
    print(y)
else:
    print(x)

print('B:')
print(min(x,y))
```

Are they equivalent? - big small

```
x = 2
#x = 4
#x = 3

print('x =',x)

print('A:')
if x > 3:
    print('big')
else:
    print('small')

print('B:')
if x < 3:
    print('small')
else:
    print('big')
```

Are they equivalent? - Cippirillo

```
x = 3
#x = 10
#x = 11
#x = 15

print('x =', x)

print('A:')
if x % 5 == 0:
    print('cippirillo')
if x % 3 == 0:
    print('cippirillo')

```

(continues on next page)

(continued from previous page)

```
print('B:')
if x % 3 == 0 or x % 5 == 0:
    print('cippirillo')
```

Exercise - farm

Given a string `s`, write some code which prints `'BARK!'` if the string ends with `dog`, prints `'CROAK!'` if the string ends with `'frog'` and prints `'???'` in all other cases

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:
```
s = 'bulldog'
#s = 'bullfrog'
#s = 'frogbull'

print(s)

# write here

if s.endswith('dog'):
    print('BAU')
elif s.endswith('frog'):
    print('CROAK!')
else:
    print('???')
```

```
bulldog
BAU
```

</div>

[10]:
```
s = 'bulldog'
#s = 'bullfrog'
#s = 'frogbull'

print(s)

# write here
```

```
bulldog
BAU
```

Exercise - accents

Write some code which prints whether a `word` ends or not with an accented character.

- To determine if a character is accented, use the strings of accents `acute` and `grave`
- Your code must work with any `word`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[11]:
```
acute = "áéíóú"
grave = "àèìòù"

word = 'urrà'           # ends with an accent
#word = 'martello'     # does not end with an accent
#word = 'ahó'          # ends with an accent
#word = 'però'         # ends with an accent
#word = 'capitaneria' # does not end with an accent
#word = 'viceré'       # ends with an accent
#word = 'cioè'         # ends with an accent
#word = 'chéto'        # does not end with an accent
#word = 'Chi dice che la verità è una sòla?'  # does not end with an accent

# write here

if word[-1] in acute or word[-1] in grave:
    print(word,'ends with an accent!')
else:
    print(word, 'does not end with an accent')
```

```
urrà ends with an accent!
```

</div>

[11]:
```
acute = "áéíóú"
grave = "àèìòù"

word = 'urrà'           # ends with an accent
#word = 'martello'     # does not end with an accent
#word = 'ahó'          # ends with an accent
#word = 'però'         # ends with an accent
#word = 'capitaneria' # does not end with an accent
#word = 'viceré'       # ends with an accent
#word = 'cioè'         # ends with an accent
#word = 'chéto'        # does not end with an accent
#word = 'Chi dice che la verità è una sòla?'  # does not end with an accent

# write here
```

```
urrà ends with an accent!
```

Exercise - Arcana

Given an arcana `x` expressed as a string and a list of `majors` and `minors` arcanas, print to which category `x` belongs. If `x` does not belong to any category, prints `is a Mistery`.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```
x = 'Wheel of Fortune'   # major
#x = 'The Tower'         # major
#x = 'Ace of Swords'     # minor
#x = 'Two of Coins'      # minor
#x = 'Coding'            # mistery
```

(continues on next page)

(continued from previous page)

```
majors  = ['Wheel of Fortune','The Chariot', 'The Tower']
minors = ['Ace of Swords', 'Two of Coins', 'Queen of Cups']

# write here
if x in majors:
    print(x, 'is a Major Arcana')
elif x in minors:
    print(x, 'is a Minor Arcana')
else:
    print(x, 'is a Mistery')
```

```
Wheel of Fortune is a Major Arcana
```

</div>

[12]:

```
x = 'Wheel of Fortune'   # major
#x = 'The Tower'         # major
#x = 'Ace of Swords'     # minor
#x = 'Two of Coins'      # minor
#x = 'Coding'            # mistery

majors  = ['Wheel of Fortune','The Chariot', 'The Tower']
minors = ['Ace of Swords', 'Two of Coins', 'Queen of Cups']

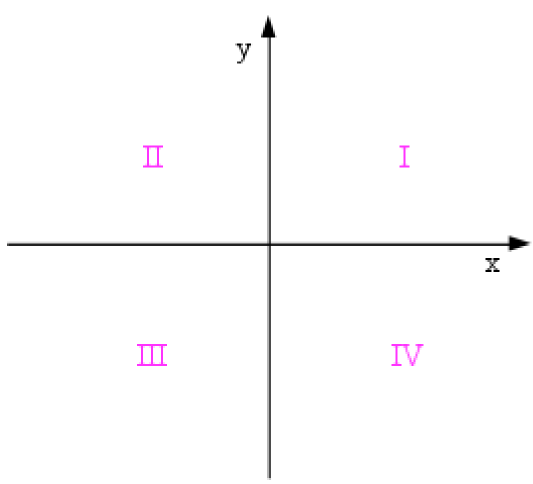
# write here
```

```
Wheel of Fortune is a Major Arcana
```

5.14.8 Nested ifs

`if` commands are *blocks* so they can be nested as any other block.

Let's make an example. Suppose you have a point at coordinates `x` and `y` and you want to know in which quadrant it lies:



You might write something like this:

[13]:
```
x,y = 5,9
#x,y = -5,9
#x,y = -5,-9
#x,y = 5,-9

print('x =',x,'y =', y)

if x >= 0:
    if y >= 0:
        print('first quadrant')
    else:
        print('fourth quadrant')
else:
    if y >= 0:
        print('second quadrant')
    else:
        print('third quadrant')
```

```
x = 5 y = 9
first quadrant
```

EXERCISE: try the various couples of suggested points by removing the comments and convince yourself the code is working as expected.

NOTE: Sometime the nested `if` can be avoided by writing sequences of `elif` with boolean expressions which verify two conditions at a time:

[14]:
```
x,y = 5,9
#x,y = -5,9
#x,y = -5,-9
#x,y = 5,-9

print('x =',x,'y =', y)

if x >= 0 and y >= 0:
    print('first quadrant')
elif x >= 0 and y < 0:
    print('fourth quadrant')
elif x < 0 and y >= 0:
    print('second quadrant')
elif x < 0 and y < 0:
    print('third quadrant')
```

```
x = 5 y = 9
first quadrant
```

Exercise - abscissae and ordinates 1

The code above is not very precise, as doesn't consider the case of points which lie on axes. In these cases instead of the quadrant number it should print:

- 'origin' when `x` and `y` are equal to `0`
- 'ascissae' when `y` is `0`
- 'ordinate' when `x` is `0`

Write down here a modified version **of the code with nested ifs** which takes into account also these cases, then test it by removing the comments from the various suggested point coordinates.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[15]:
```python
x,y = 0,0      # origin
#x,y = 0,5     # ordinate
#x,y = 5,0     # abscissa
#x,y = 5,9     # first
#x,y = -5,9    # second
#x,y = -5,-9   # third
#x,y = 5,-9    # fourth

print('x =',x,'y =', y)

# write here
if x == 0 and y == 0:
    print('origin')
elif x == 0:
    print('ordinate')
elif x > 0:
    if y == 0:
        print('abscissa')
    elif y > 0:
        print('first quadrant')
    else:
        print('fourth quadrant')
else:
    if y == 0:
        print('abscissa')
    elif y > 0:
        print('second quadrant')
    else:
        print('third quadrant')
```

```
x = 0 y = 0
origin
```

</div>

[15]:
```python
x,y = 0,0      # origin
#x,y = 0,5     # ordinate
#x,y = 5,0     # abscissa
#x,y = 5,9     # first
#x,y = -5,9    # second
#x,y = -5,-9   # third
#x,y = 5,-9    # fourth

print('x =',x,'y =', y)

# write here
```

```
x = 0 y = 0
origin
```

Esercise - abscissae and ordinates 2

If we wanted to be even more specific, instead of a generic 'absissa' or 'ordinate', we might print:

- 'abscissa between the first and fourth quadrant'
- 'abscissa between the second and third quadrant'
- 'ordinate between the first and the second quadrant'
- 'ordinate between the third and the fourth quadrant'

Copy the code from the previous exercise, and modify it to also consider such cases.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[16]:
```python
x,y = 0,0      # origin
#x,y = 0,5     # ordinate between the first and the second quadrant
#x,y = 0,-5    # ordinate between the third and the fourth quadrant
#x,y = 5,0     # abscissa between the first and the fourth quadrant
#x,y = -5,0    # abscissa between the second and the third quadrant
#x,y = 5,9     # first
#x,y = -5,9    # second
#x,y = -5,-9   # third
#x,y = 5,-9    # fourth

print('x =',x,'y =', y)

# write here
if x == 0 and y == 0:
    print('origin')
elif x == 0:
    if y > 0:
        print('ordinate between the first and the second quadrant')
    else:
        print('ordinate between the third and the fourth quadrant')
elif x > 0:
    if y == 0:
        print('abscissa between the first and the fourth quadrant')
    elif y > 0:
        print('first quadrant')
    else:
        print('fourth quadrant')
else:
    if y == 0:
        print('abscissa between the second and the third quadrant')
    elif y > 0:
        print('second quadrant')
    else:
        print('third quadrant')
```

```
x = 0 y = 0
origin
```

</div>

[16]:
```python
x,y = 0,0      # origin
#x,y = 0,5     # ordinate between the first and the second quadrant
#x,y = 0,-5    # ordinate between the third and the fourth quadrant
```

(continues on next page)

(continued from previous page)

```
#x,y = 5,0    # abscissa between the first and the fourth quadrant
#x,y = -5,0   # abscissa between the second and the third quadrant
#x,y = 5,9    # first
#x,y = -5,9   # second
#x,y = -5,-9  # third
#x,y = 5,-9   # fourth

print('x =',x,'y =', y)

# write here


```

```
x = 0 y = 0
origin
```

Exercise - bus

You must catch the bus, and only have few minutes left. To do the trip:

- you need the backpack, otherwise you remain at home
- you also need money for the ticket or the transport card or both, otherwise you remain at home.

Write some code which given three variables `backpack`, `money` and `card`, prints what you see in the comments according to the various cases. Once you're done writing the code, test the results by removing comments from the assignments.

- **HINT**: to keep track of the found objects, try creating a list of strings which holds the objects

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[17]:

```

backpack, money, card = True, False, True
# I have no money !
# I've found: backpack,card
# I can go !

#backpack, money, card = False, False, True
# I don't have the backpack, I can't go !

#backpack, money, card = True, True, False
# I have no card !
# I've found: backpack,money
# I can go !

#backpack, money, card = True, True, True
# I've found: backpack,money,card
# I can go !

#backpack, money, card = True, False, False
# I have no money !
# I have no card !
# I don't have the card nor the money, I can't go !


```

(continues on next page)

(continued from previous page)

```
# write here

found = []
if backpack:
    found.append('backpack')
    if money:
        found.append('money')
    else:
        print('I have no money !')
    if card:
        found.append('card')
    else:
        print('I have no card !')
    if money or card:
        print("I've found:", ','.join(found))
        print('I can go !')
    else:
        print("I don't have the card nor the money, I can't go !")
else:
    print("I don't have the backpack, I can't go !")
```

```
I have no money !
I've found: backpack,card
I can go !
```

</div>

[17]:

```
backpack, money, card = True, False, True
# I have no money !
# I've found: backpack,card
# I can go !

#backpack, money, card = False, False, True
# I don't have the backpack, I can't go !

#backpack, money, card = True, True, False
# I have no card !
# I've found: backpack,money
# I can go !

#backpack, money, card = True, True, True
# I've found: backpack,money,card
# I can go !

#backpack, money, card = True, False, False
# I have no money !
# I have no card !
# I don't have the card nor the money, I can't go !


# write here
```

```
I have no money !
I've found: backpack,card
I can go !
```

Exercise - chronometer

A chronometer is counting the hours, minutes and seconds since the midnight of a certain day in a string `chronometer`, in which the numbers of hours, minutes and seconds are separated by colon **:**

Write some code which prints the day phase according to the number of passed hours:

- from 6:00 included to 12:00 excluded: prints `morning`
- from 12:00 included to 18:00 excluded: prints `afternoon`
- from 18:00 included to 21:00 excluded: prints `evening`
- from 21:00 included to 6:00 excluded: prints `night`
- **USE** `elif` with multiple boolean expressions
- Your code **MUST** work even if the chronometer goes beyond 23:59:59, see examples
- **HINT**: use the modulo operator `%` for having hours which only go from 0 to 23

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```python
chronometer = '10:23:43'   # morning
#chronometer = '12:00:00'  # afternoon
#chronometer = '15:56:02'  # afternoon
#chronometer = '19:23:27'  # evening
#chronometer = '21:45:15'  # night
#chronometer = '02:45:15'  # night
#chronometer = '27:45:30'  # night
#chronometer = '32:28:30'  # morning

# write here
hour = int(chronometer.split(':')[0]) % 24

if hour >= 6 and hour < 12:
    print('morning')
elif hour >=12 and hour < 18:
    print('afternoon')
elif hour >=18 and hour < 21:
    print('evening')
else:
    print('night')
```

```
morning
```

</div>

[18]:
```python
chronometer = '10:23:43'   # morning
#chronometer = '12:00:00'  # afternoon
#chronometer = '15:56:02'  # afternoon
#chronometer = '19:23:27'  # evening
#chronometer = '21:45:15'  # night
#chronometer = '02:45:15'  # night
```

(continues on next page)

(continued from previous page)

```
#chronometer = '27:45:30'  # night
#chronometer = '32:28:30'  # morning

# write here
```

```
morning
```

5.14.9 Questions - Are they equivalent?

Look at the following code fragments: each contains two parts, A and B. For each value of `x`, try guessing whether part A will print exactly the same result printed by code in part B

- **FIRST** think about the answer
- **THEN** try executing with each of the suggested values of `x`

Are they equivalent? - inside outside 1

```
x = 3
#x = 4
#x = 5

print('x =',x)

print('A:')
if x > 3:
    if x < 5:
        print('inside')
    else:
        print('outside')
else:
    print('outside')

print('B:')
if x > 3 and x < 5:
    print('inside')
else:
    print('outside')
```

Are they equivalent? - stars planets

```
x = 2
#x = 3
#x = 4

print('x =', x)

print('A:')
if not x > 3:
    print('stars')
```

(continues on next page)

(continued from previous page)

```
else:
    print('planets')

print('B:')
if x > 3:
    print('planets')
else:
    print('stars')
```

Are they equivalent? - green red

```
x = 10
#x = 5
#x = 0

print('x =',x)

print('A:')
if x >= 5:
    print('green')
    if x >= 10:
        print('red')


print('B:')
if x >= 10:
    if x >= 5:
        print('green')
    print('red')
```

Are they equivalent? - circles squares

```
x = 4
#x = 3
#x = 2
#x = 1
#x = 0

print('x =', x)

print('A:')
if x > 3:
    print('circles')
else:
    if x > 1:
        print('squares')
    else:
        print('triangles')

print('B:')
if x <= 1:
    print('triangles')
elif x <= 3:
```

(continues on next page)

(continued from previous page)

```
    print('squares')
else:
    print('circles')
```

Are they equivalent? - inside outside 2

```
x = 7
#x = 0
#x = 15

print('x =', x)

print('A:')
if x > 5:
    if x < 10:
        print('inside')
    else:
        print('outside')
else:
    print('outside')

print('B:')
if not x > 5 and not x < 10:
    print('outside')
else:
    print('inside')
```

Are they equivalent? - Ciabanga

```
x = 4
#x = 5
#x = 6
#x = 9
#x = 10
#x = 11

print('x =', x)

print('A:')
if x < 6:
    print('Ciabanga!')
else:
    if x >= 10:
        print('Ciabanga!')

print('B:')
if x <= 5 or not x < 10:
    print('Ciabanga!')
```

Exercise - The maximum

Write some code which prints the maximum value among the numbers `x`, `y` and `z`

- use **nested ifs**
- **DO NOT** use the function `max`
- **DO NOT** create variables called `max` (it would violate the V Commandment[249]: you shall never ever redefine system functions)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[19]:
```python
x,y,z = 1,2,3
#x,y,z = 1,3,2
#x,y,z = 2,1,3
#x,y,z = 2,3,1
#x,y,z = 3,1,2
#x,y,z = 3,2,1

# write here

if x > y:
    if x > z:
        print(x)
    else:
        print(z)
elif y > z:
    print(y)
else:
    print(z)
```

```
3
```

</div>

[19]:
```python
x,y,z = 1,2,3
#x,y,z = 1,3,2
#x,y,z = 2,1,3
#x,y,z = 2,3,1
#x,y,z = 3,1,2
#x,y,z = 3,2,1

# write here

```

```
3
```

[249] https://en.softpython.org/commandments.html#V-COMMANDMENT

5.14.10 Ternary operator

In some cases, initializing a variable with different values according to a condition may result convenient.

Example:

The discount which is applied to a purchase depends on the purchased quantity. Create a variable `discount` by setting its value to `0` if the variable `expense` is less than 100€, or `10%` if it is greater.

[20]:
```python
expense = 200
discount = 0

if expense > 100:
    discount = 0.1
else:
    discount = 0 # not necessary

print("expense:", expense, "  discount:", discount)
```

```
expense: 200   discount: 0.1
```

The previous code can be written more concisely like this:

[21]:
```python
expense = 200
discount = 0.1 if expense > 100 else 0
print("expense:", expense, "  discount:", discount)
```

```
expense: 200   discount: 0.1
```

The syntax of the ternary operator is:

```python
VARIABLE = VALUE if CONDITION else ANOTHER_VALUE
```

which means that `VARIABLE` is initialized to `VALUE` if `CONDITION` is `True`, otherwise it is initialized to `OTHER_VALUE`

Questions ternary ifs

QUESTION: Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1. ```python
   y = 3
   x = 8 if y < 2 else 9
   print(x)
   ```

2. ```python
   y = 1
   z = 2 if y < 3
   ```

3. ```python
   y = 10
   z = 2 if y < 3 elif y > 5 9
   ```
   ```

Exercise - shoes

Write some code which given the numerical variable `shoes`, if `shoes` is less than 10 it gets incremented by `1`, otherwise it is decremented by `1`

- **USE ONLY** the **ternary if**
- Your code must work for *any* value of `shoes`

Example 1 - given:

```
shoes = 2
```

After your code, it must result:

```
>>> print(shoes)
3
```

Example 2 - given:

```
shoes = 16
```

After your code, it must result:

```
>>> print(shoes)
15
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```
shoes = 2
#shoes = 16

# write here

shoes = shoes + 1 if shoes < 10 else shoes - 1
print('shoes =', shoes)
```

```
shoes = 3
```

</div>

[22]:
```
shoes = 2
#shoes = 16

# write here
```

```
shoes = 3
```

Exercise - the little train

Write some code which given 3 strings `sa`, `sb` and `sc` assigns the string `CHOO CHOO` to variable `x` if it is possible to compose `sa`, `sb` and `sc` to obtain the writing `'the little train'`, otherwise assigns the string `':-('`

- **USE** a ternay if
- your code must work for **any** triplet of strings
- **NOTE**: we are only interested to know IF it is possible to compose writings like `'the little train'`, we are NOT interested in which order they will get composed
- **HINT**: you are allowed to create a helper list

Example 1 - given:

```
sa,sb,sc = "little","train","the"
```

after your code, it must result:

```
>>> print(x)
CHOO CHOO
```

Example 2 - given:

```
sa,sb,sc = "quattro","ni","no"
```

after your code, it must result:

```
>>> print(x)
:-(
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```
sa,sb,sc = "little","train","the"    # CHOO CHOO
#sa,sb,sc = "little","the","train"   # CHOO CHOO
#sa,sb,sc = "a","little","train"     # :-(
#sa,sb,sc = "train","no","no"        # :-(


# write here
words = [sa,sb,sc]
x = 'CHOO CHOO' if 'the' in words and 'little' in words and 'train' in words else ':-(
↪'

print(x)
```

```
CHOO CHOO
```

</div>

[23]:
```
sa,sb,sc = "little","train","the"    # CHOO CHOO
#sa,sb,sc = "little","the","train"   # CHOO CHOO
#sa,sb,sc = "a","little","train"     # :-(
#sa,sb,sc = "train","no","no"        # :-(


# write here
```

(continues on next page)

(continued from previous page)

```

```

```
CHOO CHOO
```

5.14.11 References

- SoftPython - Basics - booleans[250]
- W3Resources - Conditional statements and loops[251] (many exercises)

[ ]:

```

```

5.15 Conditionals - for loops

5.15.1 Download exercises zip

Browse online files[252]

If we want to perform some actions for each element of a collection, we will need the so-called `for` loop, which allows to *iterate* any sequence.

5.15.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
control-flow
    flow1-if.ipynb
    flow1-if-sol.ipynb
    flow2-for.ipynb
    flow2-for-sol.ipynb
    flow3-while.ipynb
    flow3-while-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `flow2-for.ipynb`

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`

[250] https://en.softpython.org/basics/basics-sol.html#Booleans
[251] https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php
[252] https://github.com/DavidLeoni/softpython-en/tree/master/control-flow

- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.15.3 Iteration by element

If we have a sequence like this list:

```
[2]: sports = ['volleyball', 'tennis', 'soccer', 'swimming']
```

and we want ito use every element of the list in some way (for exemple to print them), we can go through them (more precisely, *iterate)* with a `for` cycle:

```
[3]: for element in sports:
         print('Found an element!')
         print(element)

     print('Done!')
```

```
Found an element!
volleyball
Found an element!
tennis
Found an element!
soccer
Found an element!
swimming
Done!
```

Let's see what happens in Python Tutor:

```
[4]: # WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
     #          (it's sufficient to execute it only once)

     import jupman
```

```
[5]: sports = ['volleyball', 'tennis', 'soccer', 'swimming']
     for element in sports:
         print('Found an element!')
         print(element)

     print('Done!')

     jupman.pytut()
```

```
Found an element!
volleyball
Found an element!
tennis
Found an element!
soccer
Found an element!
swimming
Done!
```

```
[5]: <IPython.core.display.HTML object>
```

Names of variables in `for`

At each iteration, an element of the list is assigned to the variable `element`.

As variable name we can choose whatever we like, for example this code is totally equivalent to the previous one:

[6]:
```
sports = ['volleyball', 'tennis', 'soccer', 'swimming']
for name in sports:
    print('Found an element!')
    print(name)

print('Done!')
```

```
Found an element!
volleyball
Found an element!
tennis
Found an element!
soccer
Found an element!
swimming
Done!
```

We need to be careful about one thing:

> II COMMANDMENT[253]: **Whenever you insert a variable in a `for` cycle, such variables must be new**

If you defined the variable before, you shall not reintroduce it in a `for`, as this would bring confusion in the readers' mind.

For example:

[7]:
```
sports = ['volleyball', 'tennis', 'soccer', 'swimming']
my_var = 'hello'

for my_var in sports:  # you lose the original variable
    print(my_var)

print(my_var) # prints 'swimming' instead of 'hello'
```

```
volleyball
tennis
soccer
swimming
swimming
```

[253] https://en.softpython.org/commandments.html#II-COMMANDMENT

Iterating strings

Strings are sequences of characters, so we can iterate them with `for`:

[8]:
```python
for character in "hello":
    print(character)
```

```
h
e
l
l
o
```

Iterating tuples

Tuples are also sequences so we can iterate them:

[9]:
```python
for word in ("I'm", 'visiting', 'a', 'tuple'):
    print(word)
```

```
I'm
visiting
a
tuple
```

Questions - iteration

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1.
```python
for i in [1,2,3]:
    print(i)
```

2.
```python
for x in 7:
    print(x)
```

3.
```python
for x in [7]:
    print(x)
```

4.
```python
for x in ['a','b','c']:
    x
```

5.
```python
for i in []:
    print('GURB')
```

6.
```python
for i in [1,2,3]:
    print(type(i))
```

7.
```python
for i in '123':
    print(type(i))
```

8.
```python
for i in 'abc':
    print(i)
```

9.
```
for x in ((4,5,6)):
    print(x)
```

10.
```
for x in [[1],[2,3],[4,5,6]]:
    print(x)
```

11.
```
x = 5
for x in ['a','b','c']:
    print(x)
print(x)
```

12.
```
for x in ['a','b','c']:
    pass
print(x)
```

13.
```
for x in [1,2,3,4,5,6,7,8]:
    if x % 2 == 0:
        print(x)
```

14.
```
la = [4,5,6]
for x in la:
    print(x)
la.reverse()
for x in la[1:]:
    print(x)
```

Exercise - magic carpet

⊛ Months ago you bought a carpet from a pitchman. After some time, after a particularly stressful day, you say 'I wish I went on vacation to some exotic places, like say, *Marrakesh!'* To your astonishment, the carpet jumps in the air and answers: 'I hear and obey!'

Write some code which given the lists of places `trip1` and `trip2` prints all the visited stops.

Example - given:

```
trip1 = ['Marrakesh','Fez','Bazaar','Kasbah']
trip2 = ['Koutoubia', 'El Badii', 'Chellah']
```

Prints:

```
The first trip starts
      You: Let's go to Marrakesh !
   Carpet: I hear and obey
      You: Let's go to Fez !
   Carpet: I hear and obey
      You: Let's go to Bazaar !
   Carpet: I hear and obey
      You: Let's go to Kasbah !
   Carpet: I hear and obey
End of second trip

The second trip starts
      You: Let's go to Koutoubia !
```

(continues on next page)

(continued from previous page)

```
    Carpet: I hear and obey
      You: Let's go to El Badii !
    Carpet: I hear and obey
      You: Let's go to Chellah !
    Carpet: I hear and obey
End of second trip
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:
```python
trip1 = ['Marrakesh','Fez','Bazaar','Kasbah']
trip2 = ['Koutoubia', 'El Badii', 'Chellah']

# write here
print('The first trip starts')
for place in trip1:
    print("      You: Let's go to",place,'!')
    print('   Carpet: I hear and obey')
print('End of second trip')

print()

print('The second trip starts')
for place in trip2:
    print("      You: Let's go to",place,'!')
    print('    Carpet: I hear and obey')
print('End of second trip')
```

```
The first trip starts
      You: Let's go to Marrakesh !
   Carpet: I hear and obey
      You: Let's go to Fez !
   Carpet: I hear and obey
      You: Let's go to Bazaar !
   Carpet: I hear and obey
      You: Let's go to Kasbah !
   Carpet: I hear and obey
End of second trip

The second trip starts
      You: Let's go to Koutoubia !
    Carpet: I hear and obey
      You: Let's go to El Badii !
    Carpet: I hear and obey
      You: Let's go to Chellah !
    Carpet: I hear and obey
End of second trip
```

</div>

[10]:
```python
trip1 = ['Marrakesh','Fez','Bazaar','Kasbah']
trip2 = ['Koutoubia', 'El Badii', 'Chellah']

# write here
```

```
The first trip starts
      You: Let's go to Marrakesh !
   Carpet: I hear and obey
      You: Let's go to Fez !
   Carpet: I hear and obey
      You: Let's go to Bazaar !
   Carpet: I hear and obey
      You: Let's go to Kasbah !
   Carpet: I hear and obey
End of second trip

The second trip starts
      You: Let's go to Koutoubia !
    Carpet: I hear and obey
      You: Let's go to El Badii !
    Carpet: I hear and obey
      You: Let's go to Chellah !
    Carpet: I hear and obey
End of second trip
```

Esercise - evensum

⊛ Given the list `numbers`, write some code which calculates and prints the sum of the even **elements** (**not** the elements at even indexes !)

Example - given:

```
numbers = [3,4,1,5,12,7,9]
```

finds `4` and `12` so it must print:

```
16
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[11]:
```python
numbers = [3,4,1,5,12,7,9]

# write here
s = 0
for x in numbers:
    if x % 2 == 0:
        s += x
print(s)
```

```
16
```

</div>

[11]:
```python
numbers = [3,4,1,5,12,7,9]

# write here
```

```
16
```

Exercise - birbantello

⊛ Given a string in lowercase, write some code which prints each character in uppercase followed by the character as lowercase.

- **HINT**: to obtain uppercase characters use the `.upper()` method

Example - given:

```
s = "birbantello"
```

Prints:

```
B b
I i
R r
B b
A a
N n
T t
E e
L l
L l
O o
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```
s = "birbantello"

# write here
for x in s:
    print(x.upper(), x)
```

```
B b
I i
R r
B b
A a
N n
T t
E e
L l
L l
O o
```

</div>

[12]:
```
s = "birbantello"

# write here
```

```
B b
I i
R r
B b
A a
```

(continues on next page)

(continued from previous page)

```
N n
T t
E e
L l
L l
O o
```

Exercise - articulate

⊕ A new word is taught to a kid. He knows a lot of characters from the alphabet, but not all of them. To remember the known ones, he treats them as they where actors divided in three categories: the `good`, `bad` ad `ugly`. Write some code which given a `word` prints all the characters and for each of them tells whether it is good, bad or ugly. If a character is not recognized by the kid, prints 'not interesting'.

Example - given:

```
word = 'articulate'

good = 'abcde'
bad = 'ru'
ugly = 'ijklmn'
```

Prints:

```
a is good
r is bad
t is not interesting
i is ugly
c is good
u is bad
l is ugly
a is good
t is not interesting
e is good
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[13]:
```
word = 'articulate'

good = 'abcde'
bad = 'ru'
ugly = 'ijklmn'

# write here

for c in word:
    if c in good:
        print(c,'is good')
    elif c in bad:
        print(c, 'is bad')
    elif c in ugly:
        print(c, 'is ugly')
```

(continues on next page)

(continued from previous page)

```
    else:
        print(c, 'is not interesting')
```

```
a is good
r is bad
t is not interesting
i is ugly
c is good
u is bad
l is ugly
a is good
t is not interesting
e is good
```

</div>

[13]:

```
word = 'articulate'

good = 'abcde'
bad = 'ru'
ugly = 'ijklmn'

# write here
```

```
a is good
r is bad
t is not interesting
i is ugly
c is good
u is bad
l is ugly
a is good
t is not interesting
e is good
```

Exercise - gala

⊕ At a gala event high-society people are invited. At the beginning of the evening, doors are opened and guests enter a `queue`. Unfortunately, during these occasions uninvited guests always show up, so the concierge in the atrium is given a list of `unwelcome` ones. Whenever a guest is recognized as unwelcome, he will be taken care by the strong hands of Ferruccio the bouncer. Illustrious guests will be written instead in the list `admitted`.

Write some code which prints the various passages of the event.

Example - given:

```
queue = ['Consul','Notary','Skeleton','Dean','Goblin','Vampire', 'Jeweller']
unwelcome = {'Vampire','Goblin','Skeleton'}
admitted = []
```

Prints:

```
Open the doors!

Good evening Mr Consul
  This way, Your Excellence
  Next in line !
Good evening Mr Notary
  This way, Your Excellence
  Next in line !
Good evening Mr Skeleton
  Ferruccio, would you please take care of Mr Skeleton ?
  Next in line !
Good evening Mr Dean
  This way, Your Excellence
  Next in line !
Good evening Mr Goblin
  Ferruccio, would you please take care of Mr Goblin ?
  Next in line !
Good evening Mr Vampire
  Ferruccio, would you please take care of Mr Vampire ?
  Next in line !
Good evening Mr Jeweller
  This way, Your Excellence
  Next in line !

These guests were admitted: Consul, Notary, Dean, Jeweller
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[14]:
```python
queue = ['Consul','Notary','Skeleton','Dean','Goblin','Vampire', 'Jeweller']
unwelcome = {'Vampire','Goblin','Skeleton'}
admitted = []


# write here
print('Open the doors!')
print()
for guest in queue:

    print('Good evening Mr', guest)
    if guest in unwelcome:
        print("  Ferruccio, would you please take care of Mr", guest, '?')
    else:
        print("  This way, Your Excellence")
        admitted.append(guest)
    print('  Next in line !')

print()
print('These guests were admitted:', ', '.join(admitted))
```

```
Open the doors!

Good evening Mr Consul
  This way, Your Excellence
  Next in line !
Good evening Mr Notary
  This way, Your Excellence
  Next in line !
```

(continues on next page)

(continued from previous page)

```
Good evening Mr Skeleton
  Ferruccio, would you please take care of Mr Skeleton ?
  Next in line !
Good evening Mr Dean
  This way, Your Excellence
  Next in line !
Good evening Mr Goblin
  Ferruccio, would you please take care of Mr Goblin ?
  Next in line !
Good evening Mr Vampire
  Ferruccio, would you please take care of Mr Vampire ?
  Next in line !
Good evening Mr Jeweller
  This way, Your Excellence
  Next in line !

These guests were admitted: Consul, Notary, Dean, Jeweller
```

</div>

[14]:
```
queue = ['Consul','Notary','Skeleton','Dean','Goblin','Vampire', 'Jeweller']
unwelcome = {'Vampire','Goblin','Skeleton'}
admitted = []


# write here

```

```
Open the doors!

Good evening Mr Consul
  This way, Your Excellence
  Next in line !
Good evening Mr Notary
  This way, Your Excellence
  Next in line !
Good evening Mr Skeleton
  Ferruccio, would you please take care of Mr Skeleton ?
  Next in line !
Good evening Mr Dean
  This way, Your Excellence
  Next in line !
Good evening Mr Goblin
  Ferruccio, would you please take care of Mr Goblin ?
  Next in line !
Good evening Mr Vampire
  Ferruccio, would you please take care of Mr Vampire ?
  Next in line !
Good evening Mr Jeweller
  This way, Your Excellence
  Next in line !

These guests were admitted: Consul, Notary, Dean, Jeweller
```

Exercise - balance

⊛⊛ A crop of seeds has been harvested, and seeds will be poured in a certain number of `bags` of a given `capacity` each (i.e. `15` kilograms).

The seeds arrive in `containers` of variable capacity. Each container is placed on a weight scale and its content is poured in the current bag. As soon as the quantity `capacity` is reached, the scale weight is emptied, the bag is substituted with a new one which starts being filled from what remains from the previous fill. Write some code which prints the procedure.

Example - given:

```
containers = [5,1,7,4,3,9,5,2,7,3]
capacity = 15
```

Prints:

```
Take 5 kg
The scale weight shows 5 kg
Take 1 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 4 kg
The scale weight shows 17 kg
We reached the capacity of 15 kg, there remain 2 kg

Take 3 kg
The scale weight shows 5 kg
Take 9 kg
The scale weight shows 14 kg
Take 5 kg
The scale weight shows 19 kg
We reached the capacity of 15 kg, there remain 4 kg

Take 2 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 3 kg
The scale weight shows 16 kg
We reached the capacity of 15 kg, there remain 1 kg

We filled 3 bags
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[15]:
```
containers = [5,1,7,4,3,9,5,2,7,3]
capacity = 15

# write here
bags = 0
k = 0
for n in containers:
    k += n
    print('Take', n, 'kg')
    print('The scale weight shows', k,'kg')
    if k >= capacity:
```

(continues on next page)

(continued from previous page)

```
        print('We reached the capacity of',capacity,'kg, there remain', k - capacity,
↪'kg')
        print()
        k = k - capacity
        bags += 1

print('We filled', bags, 'bags')
```

```
Take 5 kg
The scale weight shows 5 kg
Take 1 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 4 kg
The scale weight shows 17 kg
We reached the capacity of 15 kg, there remain 2 kg

Take 3 kg
The scale weight shows 5 kg
Take 9 kg
The scale weight shows 14 kg
Take 5 kg
The scale weight shows 19 kg
We reached the capacity of 15 kg, there remain 4 kg

Take 2 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 3 kg
The scale weight shows 16 kg
We reached the capacity of 15 kg, there remain 1 kg

We filled 3 bags
```

</div>

[15]:
```
containers = [5,1,7,4,3,9,5,2,7,3]
capacity = 15

# write here
```

```
Take 5 kg
The scale weight shows 5 kg
Take 1 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 4 kg
The scale weight shows 17 kg
We reached the capacity of 15 kg, there remain 2 kg

Take 3 kg
The scale weight shows 5 kg
```

(continues on next page)

(continued from previous page)

```
Take 9 kg
The scale weight shows 14 kg
Take 5 kg
The scale weight shows 19 kg
We reached the capacity of 15 kg, there remain 4 kg

Take 2 kg
The scale weight shows 6 kg
Take 7 kg
The scale weight shows 13 kg
Take 3 kg
The scale weight shows 16 kg
We reached the capacity of 15 kg, there remain 1 kg

We filled 3 bags
```

5.15.4 Counting with range

If we need to keep track of the iteration number, we can use the iterable sequence `range`, which produces a series of integer numbers from `0` INCLUDED until the specified number EXCLUDED:

[16]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Note it *did not* print the limit `5`

When we call `range` we can also specify the starting index, which is INCLUDED in the generated sequence, while the arrival index is always EXCLUDED:

[17]:
```
for i in range(3,7):
    print(i)
```

```
3
4
5
6
```

Counting intervals: we can specify the increment to apply to the counter at each iteration by passing a third parameter, for example here we specify an increment of `2` (note the final `18` index is EXCLUDED from the sequence):

[18]:
```
for i in range(4,18,2):
    print(i)
```

```
4
6
8
10
12
14
16
```

Reverse order: we can count in reverse by using a negative increment:

[19]:
```
for i in range(5,0,-1):
    print(i)
```

```
5
4
3
2
1
```

Note how the limit `0` *was not* reached, in order to arrive there we need to write

[20]:
```
for i in range(5,-1,-1):
    print(i)
```

```
5
4
3
2
1
0
```

Questions - range

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1.
```
for x in range(1):
    print(x)
```

2.
```
for i in range(3):
    i
```

3.
```
for i in range(3):
print(i)
```

4.
```
for x in range(-1):
    print(x)
```

5.
```
for 'm' in range(3):
    print('m')
```

6.
```
for i in range(3):
    i-1
```

7.
```
for x in range(6,4,-1):
    print(x)
```

8.
```
for x in range(1,0,-1):
    print(x)
```

9.
```
for x in range(3,-3,-2):
    print(x)
```

10.
```
for x in 3:
    print(x)
```

11.
```
x = 3
for i in range(x):
    print(i)
for i in range(x,2*x):
    print(i)
```

12.
```
for x in range(range(3)):
    print(x)
```

Exercise - printdoubles

⊛ Given a positive number `n` (i.e. `n=4`) write some code which prints:

```
The double of 0 is 0
The double of 1 is 2
The double of 2 is 4
The double of 3 is 6
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[21]:
```
n = 4
# write here
for i in range(n):
    print('The double of', i, 'is', i*2)
```

```
The double of 0 is 0
The double of 1 is 2
The double of 2 is 4
The double of 3 is 6
```

</div>

[21]:
```
n = 4
# write here
```

```
The double of 0 is 0
The double of 1 is 2
The double of 2 is 4
The double of 3 is 6
```

Exercise - multiples or not

⊛⊛ Write some code which given two integer positive numbers `k` and `b`:

- first prints all the numbers from `k` INCLUDED to `b` INCLUDED which are multiples of `k`
- the prints all the numbers from `k` EXCLUDED to `b` EXCLUDED which are NOT multiples of `k`

Example - given:

```
k,b = 3,15
```

it prints:

```
Multiples of 3
3
6
9
12
15

Not divisible by 3
4
5
7
8
10
11
13
14
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```python
k,b = 3,15

# write here
print('Multiples of', k)
for i in range(k,b+1,k):
    print(i)
print()
print('Not divisible by', k)
for i in range(k+1,b):
    if i % k != 0:
        print(i)
```

```
Multiples of 3
3
6
9
12
15

Not divisible by 3
4
5
7
8
10
```

(continues on next page)

(continued from previous page)

```
11
13
14
```

</div>

[22]:
```python
k,b = 3,15

# write here
```

```
Multiples of 3
3
6
9
12
15

Not divisible by 3
4
5
7
8
10
11
13
14
```

Exercise - ab interval

⊛⊛ Given two integers `a` and `b` greater or equal than zero, write some code which prints all the integer numbers among the two bounds INCLUDED.

- **NOTE**: `a` may be greater, equal or less than `b`, your code must handle all the cases.

Example 1 - given:

```python
a,b = 5,9
```

it must print:

```
5
6
7
8
9
```

Example 2 - given:

```python
a,b = 8,3
```

it must print:

```
3
4
```

(continues on next page)

(continued from previous page)

```
5
6
7
8
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```python
a,b = 5,9   # 5 6 7 8 9
#a,b = 8,3  # 3 4 5 6 7 8
#a,b = 6,6  # 6

# write here

mn = min(a,b)
mx = max(a,b)

for x in range(mn, mx + 1):
    print(x)
```

```
5
6
7
8
9
```

</div>

[23]:
```python
a,b = 5,9   # 5 6 7 8 9
#a,b = 8,3  # 3 4 5 6 7 8
#a,b = 6,6  # 6

# write here
```

```
5
6
7
8
9
```

5.15.5 Iterating by index

If we have a sequence like a list, sometimes during the iteration it is necessary to know in which cell position we are. We can generate the indexes with `range`, and use them to access a list:

[24]:
```python
sports = ['volleyball', 'tennis', 'soccer', 'swimming']

for i in range(len(sports)):
    print('position', i)
    print(sports[i])
```

```
position 0
volleyball
position 1
```

(continues on next page)

(continued from previous page)

```
tennis
position 2
soccer
position 3
swimming
```

Note we passed to `range` the dimension of the list obrained with `len`.

Exercise - kitchen

⊕ Write some code which given a list of an even number of strings `kitchen`, prints the couples of elements we can find in sequences, one row at a time

Example - given:

```
kitchen = ['oil', 'soup', 'eggs', 'pie', 'tomato sauce', 'pasta', 'meat sauce',
↪'lasagna']
```

Prints:

```
oil, soup
eggs, pie
tomato sauce, pasta
meat sauce, lasagna
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[25]: kitchen = ['oil', 'soup', 'eggs', 'pie', 'tomato sauce', 'pasta', 'meat sauce',
↪'lasagna']

# write here
for i in range(0,len(kitchen)-1, 2):
    print(kitchen[i] +',', kitchen[i+1])
```

```
oil, soup
eggs, pie
tomato sauce, pasta
meat sauce, lasagna
```

</div>

```
[25]: kitchen = ['oil', 'soup', 'eggs', 'pie', 'tomato sauce', 'pasta', 'meat sauce',
↪'lasagna']

# write here
```

```
oil, soup
eggs, pie
tomato sauce, pasta
meat sauce, lasagna
```

Exercise - neon

⊕ Given two lists `la` and `lb` of *equal length* n, write some code which prints their characters separated by a space on n rows

Example - given:

```
la = ['n','e','o','n']
lb = ['s','h','o','w']
```

prints:

```
n s
e h
o o
n w
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[26]:

```python
la = ['n','e','o','n']
lb = ['s','h','o','w']

# write here

for i in range(len(la)):
    print(la[i],lb[i])
```

```
n s
e h
o o
n w
```

</div>

[26]:

```python
la = ['n','e','o','n']
lb = ['s','h','o','w']

# write here
```

```
n s
e h
o o
n w
```

Exercise - emotions

⊕ Given the list of strings `emotions` and another one `grade` containing the numbers `-1` and `1`, write some code which prints the emotions followed with 'positive' if their corresponding grade is a number greater than zero or 'negative' otherwise

Example - given:

```
emotions = ['Fear', 'Anger','Sadness','Joy','Disgust','Ecstasy']
grade =    [-1,       -1,     -1,       1,     -1,        1]
```

prints:

```
Fear : negative
Anger : negative
Sadness : negative
Joy : positive
Disgust : negative
Ecstasy : positive
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[27]:
```python
emotions = ['Fear', 'Anger','Sadness','Joy','Disgust','Ecstasy']
grade =    [-1,       -1,     -1,       1,     -1,        1]

# write here
for i in range(len(emotions)):
    if grade[i] > 0:
        print(emotions[i], ': positive')
    else:
        print(emotions[i], ': negative')
```

```
Fear : negative
Anger : negative
Sadness : negative
Joy : positive
Disgust : negative
Ecstasy : positive
```

</div>

[27]:
```python
emotions = ['Fear', 'Anger','Sadness','Joy','Disgust','Ecstasy']
grade =    [-1,       -1,     -1,       1,     -1,        1]

# write here
```

```
Fear : negative
Anger : negative
Sadness : negative
Joy : positive
Disgust : negative
Ecstasy : positive
```

Exercise - organetto

⊛ Given a string `s`, write some code which prints all the substrings you can obtain from the position of the character `'n'` and which terminates with the last character of `s`.

Example - given:

```
s = 'organetto'
```

Prints:

```
netto
etto
tto
to
o
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[28]:
```
s = 'organetto'

# write here
for i in range(s.index('n'), len(s)):
    print(s[i:])
```

```
netto
etto
tto
to
o
```

</div>

[28]:
```
s = 'organetto'

# write here
```

```
netto
etto
tto
to
o
```

Exercise - sghiribizzo

Write some code which given the string `s` prints all the possible combinations of row couples such that a row begins with the first characters of `s` and the successive continues with the following characters.

Example - given:

```
s = 'sghiribizzo'
```

prints:

```
s
 ghiribizzo
sg
  hiribizzo
sgh
   iribizzo
sghi
    ribizzo
sghir
     ibizzo
sghiri
      bizzo
sghirib
       izzo
sghiribi
        zzo
sghiribiz
         zo
sghiribizz
          o
sghiribizzo
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[29]:
```
s = 'sghiribizzo'
# write here
for i in range(len(s)):
    print(s[:i+1])
    print(' '*i,s[i+1:])
```

```
s
 ghiribizzo
sg
  hiribizzo
sgh
   iribizzo
sghi
    ribizzo
sghir
     ibizzo
sghiri
      bizzo
sghirib
       izzo
sghiribi
        zzo
sghiribiz
         zo
sghiribizz
          o
sghiribizzo
```

</div>

[29]:
```
s = 'sghiribizzo'
# write here
```

(continues on next page)

(continued from previous page)

```
s
 ghiribizzo
sg
  hiribizzo
sgh
   iribizzo
sghi
    ribizzo
sghir
     ibizzo
sghiri
      bizzo
sghirib
       izzo
sghiribi
        zzo
sghiribiz
         zo
sghiribizz
          o
sghiribizzo
```

Exercise - dna

Given two DNA strings `s1` and `s2` of equal length, write some code which prints among the first and second string another string made by spaces `` and pipe | where equal characters are found.

- **HINT**: create a list containing the characters space or the character |, and only at the end convert the string by using strings `join` method (doing so is much more efficient than keep generating strings with + operator)

Example - given:

```
s1 = "ATACATATAGGGCCAATTATTATAAGTCAC"
s2 = "CGCCACTTAAGCGCCCTGTATTAAAGTCGC"
```

Prints:

```
ATACATATAGGGCCAATTATTATAAGTCAC
   ||  || |  |  |   |  ||||| |
CGCCACTTAAGCGCCCTGTATTAAAGTCGC
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[30]:

```python
s1 = "ATACATATAGGGCCAATTATTATAAGTCAC"
s2 = "CGCCACTTAAGCGCCCTGTATTAAAGTCGC"

# write here
lst = []
for i in range(len(s1)):
    if(s1[i] == s2[i]):
```

(continues on next page)

(continued from previous page)

```
        lst.append('|')
    else:
        lst.append(' ')

bars = ''.join(lst)

print(s1)
print(bars)
print(s2)
```

```
ATACATATAGGGCCAATTATTATAAGTCAC
   ||   || |  |  |   |  ||||| |
CGCCACTTAAGCGCCCTGTATTAAAGTCGC
```

</div>

[30]:

```
s1 = "ATACATATAGGGCCAATTATTATAAGTCAC"
s2 = "CGCCACTTAAGCGCCCTGTATTAAAGTCGC"

# write here
```

```
ATACATATAGGGCCAATTATTATAAGTCAC
   ||   || |  |  |   |  ||||| |
CGCCACTTAAGCGCCCTGTATTAAAGTCGC
```

Exercise - sportello

✪✪ Given a string `s`, prints the first half of the characters as lowercase and the following half as uppercase.

- if the string is of odd length, the first half must have one character *more* than the second string.

Example - given:

```
s = 'sportello'
```

Your code must print:

```
s
p
o
r
t
E
L
L
O
```

(note that 'sportello' has odd length and there are *five* characters in the first half and *four* in the second

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[31]: s =  'sportello'   # sportELLO
      #s = 'maglia'      # magLIA

      # write here

      if len(s) % 2 == 1:
          midpoint = (len(s) // 2) + 1
      else:
          midpoint = (len(s) // 2)

      for i in range(midpoint):
          print(s[i])

      for i in range(midpoint, len(s)):
          print(s[i].upper())
```

```
s
p
o
r
t
E
L
L
O
```

</div>

```
[31]: s =  'sportello'   # sportELLO
      #s = 'maglia'      # magLIA

      # write here
```

```
s
p
o
r
t
E
L
L
O
```

Exercise - farm

⊛⊛ Given a dictionary `sounds` which associates animal names to the sounds they produce, and a list `rooms` of tuples of 2 elements containing the animal names, write some code that for each room prints the sounds you hear while passing in front of it.

- **NOTE**: the rooms to print are numbered **from 1**

Example - given:

```
sounds = {'dog':'Bark!',
          'cat':'Mew!',
```

(continues on next page)

(continued from previous page)

```
           'cow':'Moo!',
           'sheep':'Bleat!'}

rooms = [('dog', 'sheep'),
         ('cat','cow'),
         ('cow', 'dog')]
```

Prints:

```
In the room 1 we hear Bark! and Bleat!
In the room 2 we hear Mew! and Moo!
In the room 3 we hear Moo! and Bark!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[32]:

```
sounds = {'dog':'Bark!',
          'cat':'Mew!',
          'cow':'Moo!',
          'sheep':'Bleat!'}

rooms = [('dog', 'sheep'),
         ('cat','cow'),
         ('cow', 'dog')]

# write here

for i in range(len(rooms)):
    room = rooms[i]
    print('In the room',i+1,'we hear',sounds[room[0]], 'and', sounds[room[1]])
```

```
In the room 1 we hear Bark! and Bleat!
In the room 2 we hear Mew! and Moo!
In the room 3 we hear Moo! and Bark!
```

</div>

[32]:

```
sounds = {'dog':'Bark!',
          'cat':'Mew!',
          'cow':'Moo!',
          'sheep':'Bleat!'}

rooms = [('dog', 'sheep'),
         ('cat','cow'),
         ('cow', 'dog')]

# write here
```

```
In the room 1 we hear Bark! and Bleat!
In the room 2 we hear Mew! and Moo!
In the room 3 we hear Moo! and Bark!
```

Exercise - pokemon

⊛⊛⊛ Given a list `pokemon` and a number `g` of groups, write some code which prints `g` rows showing all the group components. Group the pokemons in the order you find them in the list.

- **HINT 1**: To obtain the number of group components you should use integer division `//`
- **HINT 2**: to print group components use the method `join` of strings

Example 1 - given:

```
#                0         1          2          3            4          5
pokemon = ['Charizard','Gengar','Arcanine','Bulbasaur','Blaziken','Umbreon',
#                6         7          8          9            10         11
           'Lucario','Gardevoir','Eevee','Dragonite', 'Volcarona', 'Sylveon' ]
g = 3
```

prints:

```
group 1 : Charizard and Gengar and Arcanine and Bulbasaur
group 2 : Blaziken and Umbreon and Lucario and Gardevoir
group 3 : Eevee and Dragonite and Volcarona and Sylveon
```

Example 2 - given:

```
#                0         1          2          3            4          5
pokemon = ['Charizard','Gengar','Arcanine','Bulbasaur','Blaziken','Umbreon',
#                6         7          8          9            10         11
           'Lucario','Gardevoir','Eevee','Dragonite', 'Volcarona', 'Sylveon' ]

g = 4
```

prints:

```
group 1 : Charizard and Gengar and Arcanine
group 2 : Bulbasaur and Blaziken and Umbreon
group 3 : Lucario and Gardevoir and Eevee
group 4 : Dragonite and Volcarona and Sylveon
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[33]:
```python
#                0         1          2          3            4          5
pokemon = ['Charizard','Gengar','Arcanine','Bulbasaur','Blaziken','Umbreon',
#                6         7          8          9            10         11
           'Lucario','Gardevoir','Eevee', 'Dragonite', 'Volcarona', 'Sylveon' ]
g = 3
#g = 4

# write here
k = len(pokemon) // g  # pokemon in a group

for i in range(0, g):
    print('group',i+1,':', ' and '.join(pokemon[i*k:(i+1)*k]))
```

```
group 1 : Charizard and Gengar and Arcanine and Bulbasaur
group 2 : Blaziken and Umbreon and Lucario and Gardevoir
group 3 : Eevee and Dragonite and Volcarona and Sylveon
```

</div>

[33]:
```
#              0          1          2           3            4          5
pokemon = ['Charizard','Gengar','Arcanine','Bulbasaur','Blaziken','Umbreon',
#              6          7          8          9            10          11
           'Lucario','Gardevoir','Eevee', 'Dragonite', 'Volcarona', 'Sylveon' ]
g = 3
#g = 4

# write here
```

```
group 1 : Charizard and Gengar and Arcanine and Bulbasaur
group 2 : Blaziken and Umbreon and Lucario and Gardevoir
group 3 : Eevee and Dragonite and Volcarona and Sylveon
```

5.15.6 Modifying during iteration

Suppose you have a list `la` containing characters, and you are asked to duplicate all the elements, for example if you have

```
lst = ['a','b','c']
```

after your code it must result

```
>>> print(lst)
['a','b','c','a','b','c']
```

Since you gained such great knowledge about iteration,you might be tempted to write something like this:

```
for char in lst:
    lst.append(char)     # WARNING !
```

QUESTION: Do you see any problem?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: if we go through the list and in *the meanwhile* we keep adding pieces, there is a concrete risk we will never terminate examining the list! Read carefully what follows:

</div>

X COMMANDMENT[254]: **You shall never ever add nor remove elements from a sequence you are iterating with a** `for` **!**

Falling into such temptations **would produce totally unpredictable behaviours** (do you know the expression *pulling the rug out from under your feet ?*)

What about removing? We've seen that adding is dangerous, but so is removing. Suppose you have to eliminate all the elements from a list, you might be tempted to write something like this:

[34]:
```
my_list = ['a','b','c','d','e']

for el in my_list:
    my_list.remove(el)    # VERY BAD IDEA
```

[254] https://en.softpython.org/commandments.html#X-COMMANDMENT

Have a close look at the code. Do you think we removed everything, uh?

```
[35]: my_list
```

```
[35]: ['b', 'd']
```

`O_o'` The absurd result is given by the internal implementation of Python, our version of Pyhton gives this result, yours might give a completely different one. **So be careful!**

If you really need to remove elements from a sequence you are iterating, use a while cycle[255] or duplicate first a copy of the original sequence.

Exercise - duplicate

⊛ Try writing some code which MODIFIES a list `la` by duplicating the elements

- use a `for` cycle
- **DO NOT** use list multiplication

Example - given:

```
la = ['a','b','c']
```

after your code, it must result:

```
>>> la
['a','b','c','a','b','c']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[36]: la = ['a','b','c']

# write here
for element in list(la):   # with list we create a *copy* of the original list,  which␣
↪remains stable
    la.append(element)
print(la)
```

```
['a', 'b', 'c', 'a', 'b', 'c']
```

</div>

```
[36]: la = ['a','b','c']

# write here
```

```
['a', 'b', 'c', 'a', 'b', 'c']
```

[255] https://en.softpython.org/control-flow/flow3-while-sol.html

Exercise - hammers

⊛ Given a list of characters `la`, MODIFY the list by changing all the characters at even indeces with the character `z`

Example - given:

```
la = ['h', 'a', 'm', 'm', 'e', 'r', 's']
```

after your code, it must result:

```
>>> print(la)
['z', 'a', 'z', 'm', 'z', 'r', 'z']
```

- **NOTE**: here we *are not* adding nor removing cells from the list

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[37]:
```
la = ['h', 'a', 'm', 'm', 'e', 'r', 's']

# write here
for i in range(len(la)):
    if i % 2 == 0:
        la[i] = 'z'
print(la)
```

```
['z', 'a', 'z', 'm', 'z', 'r', 'z']
```

</div>

[37]:
```
la = ['h', 'a', 'm', 'm', 'e', 'r', 's']

# write here
```

```
['z', 'a', 'z', 'm', 'z', 'r', 'z']
```

Exercise - Orangutan

⊛⊛ Given two strings `sa` and `sb`, write some code which places in the string `sc` a string composed by alternating all the characters in `sa` and `sb`.

- if a string is shorter than the other one, at the end of `sc` put all the remaining characters from the other string.
- **HINT**: even if it is possible to augment a string a character at a time at each iteration, each time you do so a new string is created (because strings are immutable). So it's more efficient to keep augmenting a list, and then convert to string only at the very end.

Example - given:

```
sa,sb = 'gibbon', 'ORANGUTAN'
```

after your code it must result:

```
>>> print(sc)
gOiRbAbNoGnUTAN
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[38]:
```
sa,sb = 'gibbon', 'ORANGUTAN'    # gOiRbAbNoGnUTAN
#sa,sb = 'cruise ship', 'BOAT'   # cBrOuAiTse ship

# write here
temp = []

for i in range(len(sa)):
    temp.append(sa[i])
    if i < len(sb):
        temp.append(sb[i])

if i < len(sb):
    temp.extend(sb[i+1:])

sc = ''.join(temp)
print(sc)
```

```
gOiRbAbNoGnUTAN
```

</div>

[38]:
```
sa,sb = 'gibbon', 'ORANGUTAN'    # gOiRbAbNoGnUTAN
#sa,sb = 'cruise ship', 'BOAT'   # cBrOuAiTse ship

# write here
```

```
gOiRbAbNoGnUTAN
```

Exercise - basket

⊛⊛⊛ There is a `basket` full of fruits, which we represent as a list of strings. We want to take all the fruits and put them in a plate, in the same order we find them in the basket. We must take only the fruits contained in the set `preferences`.

- The `basket` may contain duplicates, if they are in the preferences you must take them all
- the fruits are to be taken **in the same order** in which they were found

Example - given:

```
basket = ['strawberry', 'melon', 'cherry', 'watermelon', 'apple', 'melon','watermelon
↪', 'apple', ]
preferences = {'cherry', 'apple', 'strawberry'}
plate = []
```

after your code, it must result:

```
>>> print(basket)
['melon', 'watermelon', 'melon', 'watermelon']
>>> print(plate)
['strawberry', 'cherry', 'apple', 'apple']
```

You can solve the problem in two ways:

- Way 1 (simple and recommended): create a list `new_basket` and finally assign the variable `basket` to it

- Way 2 (hard, slow, not recommended but instructive): MODIFY the original `basket` list, using the pop method[256] and without ever reassigning `basket`, so no rows beginning with `basket =`

Try solving the exercise in both ways.

> **WARNING: Either way, always remember the sacred X COMMANDMENT[257]: You shall never ever add nor remove elements from a sequence you are iterating with a `for` !**

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[39]:
```python
# WAY 1

basket = ['strawberry', 'melon', 'cherry', 'watermelon', 'apple', 'melon','watermelon
↪', 'apple', ]
preferences = {'cherry', 'apple', 'strawberry'}
plate = []


# write here
new_basket = []
for fruit in basket:
    if fruit in preferences:
        plate.append(fruit)
    else:
        new_basket.append(fruit)


basket = new_basket    # we substitute the original list
print('basket:',basket)
print('plate:',plate)
```

```
basket: ['melon', 'watermelon', 'melon', 'watermelon']
plate: ['strawberry', 'cherry', 'apple', 'apple']
```

</div>

[39]:
```python
# WAY 1

basket = ['strawberry', 'melon', 'cherry', 'watermelon', 'apple', 'melon','watermelon
↪', 'apple', ]
preferences = {'cherry', 'apple', 'strawberry'}
plate = []


# write here
```

```
basket: ['melon', 'watermelon', 'melon', 'watermelon']
plate: ['strawberry', 'cherry', 'apple', 'apple']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[256] https://en.softpython.org/lists/lists3-sol.html#pop-method
[257] https://en.softpython.org/commandments.html#X-COMMANDMENT

[40]:
```python
# WAY 2

basket = ['strawberry', 'melon', 'cherry', 'watermelon', 'apple', 'melon','watermelon
↪', 'apple', ]
preferences = {'cherry', 'apple', 'strawberry'}
plate = []

# write here
copy = list(basket)
j = 0
# so we're sure to iterate on a different sequence from the one we're modifying
for i in range(len(copy)):
    fruit = copy[i]
    if fruit in preferences:
        plate.append(fruit)
        basket.pop(j)
    else:
        j += 1

print('basket:',basket)
print('plate:',plate)
```

```
basket: ['melon', 'watermelon', 'melon', 'watermelon']
plate: ['strawberry', 'cherry', 'apple', 'apple']
```

</div>

[40]:
```python
# WAY 2

basket = ['strawberry', 'melon', 'cherry', 'watermelon', 'apple', 'melon','watermelon
↪', 'apple', ]
preferences = {'cherry', 'apple', 'strawberry'}
plate = []

# write here
```

```
basket: ['melon', 'watermelon', 'melon', 'watermelon']
plate: ['strawberry', 'cherry', 'apple', 'apple']
```

5.15.7 Iterating a set

Given a set, we can examinate the element sequence with a `for` cycle.

> **WARNING**: sets iteration order is **not** predictable !
>
> To better understand why, you can see again the tutorial on sets[258]

[41]:
```python
for word in {'this', 'is', 'a', 'set'}:
    print(word)
```

[258] https://en.softpython.org/sets/sets-sol.html#Creating-a-set

```
this
is
set
a
```

[42]:
```python
s = set()
s.add('pan')
s.add('de')
s.add('mo')
s.add('nium')
print(s)
```

```
{'de', 'mo', 'pan', 'nium'}
```

Questions - sets

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1.
```python
s = set()
s.add('pan')
s.add('de')
s.add('mo')
s.add('nium')
print(s)
```

2.
```python
for x in {'a',12,'34',56,34}[2:4]:
    print(x)
```

3.
```python
for x in set(['a']) | set(['b']):
    print(x)
```

4.
```python
for x in set(['a']) & set(['b']):
    print(x)
```

5.15.8 Iterating a dictionary

Given a dictionary, we can examinate the sequence of its keys, values or both with a `for` cycle.

WARNING: keys iteration order is **not** predictable !

We can go through the keys:

[43]:
```python
pastries = {
    'cream puff':5,
    'brioche':8,
    'donut':2
}
```

[44]:
```python
for key in pastries:
    print('Found key    :', key)
    print('  with value:', pastries[key])
```

```
Found key    : donut
  with value: 2
Found key    : cream puff
  with value: 5
Found key    : brioche
  with value: 8
```

At each iteration, the declared variable `key` is assigned to a key taken from the dictionary, in an order we cannot predict.

We can also directly obtain both the key and the associated value with this notation:

[45]:
```python
for key, value in pastries.items():
    print('Found key   :', key)
    print(' with value:', pastries[key])
```

```
Found key   : donut
 with value: 2
Found key   : cream puff
 with value: 5
Found key   : brioche
 with value: 8
```

`.items()` return a list of key/value couples, and during each iteration a couple is assigned to the variable `key` and `value`.

Dictionary iteration - questions

Look at the following code fragments , and for each try guessing the result it produces (or if it gives an error):

WARNING: Remember the order is IMPOSSIBLE to foresee, so the important bit is to guess all the printed stuff

1. ```python
   for x in {'a':1,'b':2,'c':3}:
       print(x)
   ```

2. ```python
   for x in {1:'a',2:'b',3:'c'}:
       print(x)
   ```

3. ```python
   diz = {'a':1,'b':2,'c':3}
   for x in diz:
       print(x[diz])
   ```

4. ```python
   diz = {'a':1,'b':2,'c':3}
   for x in diz:
       print(diz[x])
   ```

5. ```python
   diz = {'a':1,'b':2,'c':3}
   for x in diz:
       if x == 'b':
           print(diz[x])
   ```

6. ```python
   for k,v in {1:'a',2:'b',3:'c'}:
       print(k,v)
   ```

7.
```
for x in {1:'a',2:'b',3:'c'}.values():
    print(x)
```

8.
```
for x in {1:'a',2:'b',3:'c'}.keys():
    print(x)
```

9.
```
for x in {1:'a',2:'b',3:'c'}.items():
    print(x)
```

10.
```
for x,y in {1:'a',2:'b',3:'c'}.items():
    print(x,y)
```

Questions - Are they equivalent?

Look at the following code fragments: each contains two parts, A and B. For each fragment, try guessing whether part A will print exactly the same result printed by code in part B

- **FIRST** think about the answer
- **THEN** try executing

Are they equivalent ? postin

```
diz = {
    'p':'t',
    'o':'i',
    's':'n',
}

print('A:')
for x in diz.keys():
    print(x)

print('\nB:')
for y in diz:
    print(y)
```

Are they equivalent ? cortel

```
diz = {
    'c':'t',
    'o':'e',
    'r':'l',
}

print('A:')
for p,q in diz.items():
    print(q)

print('\nB:')
for x in diz.values():
    print(x)
```

Are they equivalent ? - gel

```
diz = {
    'g':'l',
    'e':'e',
    'l':'g',
}

print('A:')
for x in diz.values():
    print(x)

print('\nB:')
for z in diz.items():
    print(z[0])
```

Are they equivalent ? - giri

```
diz = {
    'p':'g',
    'e':'i',
    'r':'r',
    'i':'i',
}

print('A:')
for p,q in diz.items():
    if p == q:
        print(p)

print('\nB:')
for x in diz:
    if x == diz[x]:
        print(x)
```

Are they equivalent - Found

First think if they are equivalent, then check with all the proposed values of `k`.

> **Be very careful about this exercise !**
>
> Getting this means having *really* understood dictionaries ;-)

```
k = 'w'
#k = 'h'
#k = 'y'
#k = 'z'

dct = {
    'w':'s',
    'h':'o',
    'y':'?',
```

(continues on next page)

(continued from previous page)

```
}

print('A:')
for x in dct:
    if x == k:
        print('Found', dct[x])

print('\nB:')
if k in dct:
    print('Found', dct[k])
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The two codes reported above are equivalent, with an important difference: code A will be executed in a time proportional to the dimension of `dct` (because it needs to go through all the dictionary), code B instead will be always executed in a short constant time which does *not* depend on the dimension of `dct`. Both the command `if k in dct` that the expression `dct[k]` which retrieves the value associated to key `k` are extremely fast.

> **WARNING: be sure to fully understand this point!**
>
> So many people write code as in part A, losing the main feature of dictionaries which is fast access. As long as data is small you may not notice, but when we have several megabytes of key/value couples you start feeling the time lost in pointless loops! For more you can read (or review) the section Fast disorder[259] in the dictionaries tutorial.

</div>

Exercise - color of hearts

⊛ Write some code which given a dictionary `suits`, for each suits prints its color.

Example - given:

```
suits = {
    'hearts':'red',
    'spades':'black',
    'diamonds':'red',
    'clubs':'black'
}
```

Prints:

> **WARNING**: do not care about the order in which values are printed!
>
> On your computer you might see different results, the important bit is that all rows get printed.

```
The color of spades is black
The color of diamonds is red
The color of hearts is red
The color of clubs is black
```

[259] https://en.softpython.org/dictionaries/dictionaries2-sol.html#Fast-disorder

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[46]:
```python
suits = {
    'hearts':'red',
    'spades':'black',
    'diamonds':'red',
    'clubs':'black'
}

# write here

for k in suits.keys():
    print('The color of', k, 'is', suits[k])
```

```
The color of spades is black
The color of clubs is black
The color of diamonds is red
The color of hearts is red
```

</div>

[46]:
```python
suits = {
    'hearts':'red',
    'spades':'black',
    'diamonds':'red',
    'clubs':'black'
}

# write here

```

```
The color of spades is black
The color of clubs is black
The color of diamonds is red
The color of hearts is red
```

Exercise - jewels

⊗ In the dictionary `jewels` some keys are equal to the respective values. Write some code which find such keys and prints them all.

Example - given:

```python
jewels = {
    'rubies': 'jade',
    'opals':'topazes',
    'gems':'gems',
    'diamonds': 'gems',
    'rubies':'rubies'
}
```

prints:

```
couple of equal elements: gems and gems
couple of equal elements: rubies and rubies
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[47]:
```python
jewels = {
    'rubies': 'jade',
    'opals':'topazes',
    'gems':'gems',
    'diamonds': 'gems',
    'rubies':'rubies'
}

# write here
for k,v in jewels.items():
    if k == v:
        print('couple of equal elements:',k, 'and', v)
```

```
couple of equal elements: rubies and rubies
couple of equal elements: gems and gems
```

</div>

[47]:
```python
jewels = {
    'rubies': 'jade',
    'opals':'topazes',
    'gems':'gems',
    'diamonds': 'gems',
    'rubies':'rubies'
}

# write here
```

```
couple of equal elements: rubies and rubies
couple of equal elements: gems and gems
```

Exercise - powers

⊕ Given a number `n`, write some code which creates a NEW dictionary `d` containing as keys the numbers from 1 a `n` INCLUDED, by associating keys to their squares.

Example - given:

```python
n = 5
```

after your code, it must result:

```python
>>> print(d)
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[48]:
```python
n = 5

# write here
```

(continues on next page)

(continued from previous page)

```
d = {}
for i in range(1,n+1):
    d[i] = i*i

print(d)
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

</div>

```
[48]: n = 5

# write here
```

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Exercise - flowers

⊕ Given a list `flowers`, write some code which creates a NEW dictionary `is_cap` which associates to each flower `True` if the flower name is written all uppercase, and `False` otherwise

- **HINT**: to verify whether a string is all uppercase, use `.isupper()` method

```
flowers = ['sunflower','GILLYFLOWER', 'tulip', 'PASSION FLOWER', 'ROSE', 'violet']
```

prints (they are in alphabetical order because we print with `pprint`):

```
>>> from pprint import pprint
>>> pprint(is_cap)
{'GILLYFLOWER': True,
 'PASSION FLOWER': True,
 'ROSE': True,
 'sunflower': False,
 'tulip': False,
 'violet': False}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[49]:
flowers = ['sunflower','GILLYFLOWER', 'tulip', 'PASSION FLOWER', 'ROSE', 'violet']

# write here

is_cap = {}
for el in flowers:
    is_cap[el] = el.isupper()

from pprint import pprint
pprint(is_cap)
```

```
{'GILLYFLOWER': True,
 'PASSION FLOWER': True,
 'ROSE': True,
```

(continues on next page)

(continued from previous page)

```
 'sunflower': False,
 'tulip': False,
 'violet': False}
```

</div>

[49]:

```
flowers = ['sunflower','GILLYFLOWER', 'tulip', 'PASSION FLOWER', 'ROSE', 'violet']

# write here
```

```
{'GILLYFLOWER': True,
 'PASSION FLOWER': True,
 'ROSE': True,
 'sunflower': False,
 'tulip': False,
 'violet': False}
```

Exercise - art

⊛ An artist painted a series of works with different techiques. In the dictionary `prices` he writes the price of each technique. The artist intend to promote a series of exhibitions, and in each of them he will present a particular technique. Supposing for each technique he produced `q` paintings, show how much he will learn in each exhibition (suppose he sells everything).

Example - given:

```
q = 20

exhibitions = ['watercolor', 'oil', 'mural', 'tempera', 'charcoal','ink']

prices = {'watercolor': 3000,
          'oil'       : 6000,
          'mural'     : 2000,
          'tempera'   : 4000,
          'charcoal'  : 7000,
          'ink'       : 1000
}
```

Prints - **this time order matters!!**

```
Expected Income:
  exhibition watercolor : 60000 €
  exhibition oil : 120000 €
  exhibition mural : 40000 €
  exhibition tempera : 80000 €
  exhibition charcoal : 140000 €
  exhibition ink : 20000 €
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```python
q = 20

exhibitions = ['watercolor', 'oil', 'mural', 'tempera', 'charcoal','ink']

prices = {'watercolor': 3000,
          'oil'       : 6000,
          'mural'     : 2000,
          'tempera'   : 4000,
          'charcoal'  : 7000,
          'ink'       : 1000
}

# write here

print('Expected Income:')
for i in range(len(exhibitions)):
    technique = exhibitions[i]
    print('  exhibition', technique, ":", prices[technique]*q,  '€')
```

```
Expected Income:
  exhibition watercolor : 60000 €
  exhibition oil : 120000 €
  exhibition mural : 40000 €
  exhibition tempera : 80000 €
  exhibition charcoal : 140000 €
  exhibition ink : 20000 €
```

</div>

[50]:
```python
q = 20

exhibitions = ['watercolor', 'oil', 'mural', 'tempera', 'charcoal','ink']

prices = {'watercolor': 3000,
          'oil'       : 6000,
          'mural'     : 2000,
          'tempera'   : 4000,
          'charcoal'  : 7000,
          'ink'       : 1000
}

# write here

```

```
Expected Income:
  exhibition watercolor : 60000 €
  exhibition oil : 120000 €
  exhibition mural : 40000 €
  exhibition tempera : 80000 €
  exhibition charcoal : 140000 €
  exhibition ink : 20000 €
```

Exercise - stationery stores

⊕ An owner of two stationery shops, in order to reorganize the stores wants to know the materials which are in common among the shops. Given two dictionaries `store1` and `store2` which associates objects to their quantity, write some code which finds all the keys in common and for each prints the sum of the found quantities.

Example - given:

```
store1 = {'pens':10,
          'folders':20,
          'papers':30,
          'scissors':40}

store2 = {'pens':80,
          'folders':90,
          'goniometer':130,
          'scissors':110,
          'rulers':120,
          }
```

prints (order is **not** important):

```
materials in common:
   pens : 90
   folders : 110
   scissors : 150
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[51]:
```
store1 = {'pens':10,
          'folders':20,
          'papers':30,
          'scissors':40}

store2 = {'pens':80,
          'folders':90,
          'goniometer':130,
          'scissors':110,
          'rulers':120,
          }
# write here

print('materials in common:')
for k in store1:
    if k in store2:
        print('  ',k, ':', store1[k] + store2[k])
```

```
materials in common:
   pens : 90
   folders : 110
   scissors : 150
```

</div>

[51]:
```
store1 = {'pens':10,
          'folders':20,
          'papers':30,
```

(continues on next page)

(continued from previous page)

```
          'scissors':40}

store2 = {'pens':80,
          'folders':90,
          'goniometer':130,
          'scissors':110,
          'rulers':120,
          }
# write here
```

```
materials in common:
   pens : 90
   folders : 110
   scissors : 150
```

Exercise - legumes

⊛ A store has numbered shelves, each containing a number of legumes expressed in kilograms. We represent `store` as a list. There is also a `registry` available as a dictionary which associates to legume names the shelves number in which they are contained.

Write some code which given a list of legume names, shows the sum of kilograms in the store for those legumes.

Example - given:

```
legumes = ['lentils', 'soy']

#         0  1  2  3  4  5
store = [50,90,70,10,20,50]

registry = {'peas':3,
            'soy':1,
            'chickpeas':5,
            'lentils':4,
            'broad beans':2,
            'beans':0,
}
```

after your code, it must print (order does **not** matter):

```
Searching for lentils and soy ...
Found 20 kg of lentils
Found 90 kg of soy
Total: 110 kg
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[52]:

```
legumes = ['lentils', 'soy']     # 110
#legumes = ['beans', 'broad beans', 'chickpeas']    # 170

#         0  1  2  3  4  5
store = [50,90,70,10,20,50]
```

(continues on next page)

(continued from previous page)

```
registry = {'peas':3,
            'soy':1,
            'chickpeas':5,
            'lentils':4,
            'broad beans':2,
            'beans':0,
}
# write here
print('Searching for', ' and '.join(legumes), '...')
s = 0
for leg in legumes:
    print('Found', store[registry[leg]], 'kg of', leg)
    s += store[registry[leg]]

print('Total:',s, 'kg')
```

```
Searching for lentils and soy …
Found 20 kg of lentils
Found 90 kg of soy
Total: 110 kg
```

</div>

[52]:

```
legumes = ['lentils', 'soy']     # 110
#legumes = ['beans', 'broad beans', 'chickpeas']   # 170

#         0  1  2  3  4  5
store = [50,90,70,10,20,50]

registry = {'peas':3,
            'soy':1,
            'chickpeas':5,
            'lentils':4,
            'broad beans':2,
            'beans':0,
}
# write here
```

```
Searching for lentils and soy …
Found 20 kg of lentils
Found 90 kg of soy
Total: 110 kg
```

Exercise - sports

⊛⊛ Write some code which given a dictionary `sports` in which people are associated to the favourite sport, create a NEW dictionari `counts` in which associates each sport to the number of people that prefer it

Exemple - given:

```
sports = {
    'Gianni':'soccer',
    'Paolo':'tennis',
    'Sara':'volleyball',
    'Elena':'tennis',
    'Roberto':'soccer',
    'Carla':'soccer',
}
```

After your code, it must result:

```
>>> print(counts)
{'tennis': 2, 'soccer': 3, 'volleyball': 1}
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[53]:
```
sports = {
    'Gianni':'soccer',
    'Paolo':'tennis',
    'Sara':'volleyball',
    'Elena':'tennis',
    'Roberto':'soccer',
    'Carla':'soccer',
}

# write here

counts = {}

for k,v in sports.items():
    if v in counts:
        counts[v] += 1
    else:
        counts[v] = 1

print(counts)
```

```
{'tennis': 2, 'volleyball': 1, 'soccer': 3}
```

</div>

[53]:
```
sports = {
    'Gianni':'soccer',
    'Paolo':'tennis',
    'Sara':'volleyball',
    'Elena':'tennis',
    'Roberto':'soccer',
    'Carla':'soccer',
}
```

(continues on next page)

(continued from previous page)

```
# write here
```

```
{'tennis': 2, 'volleyball': 1, 'soccer': 3}
```

Modifying a dictionary during iteration

Suppose you have a dictionary of provinces:

```
provinces = {
    'tn': 'Trento',
    'mi':'Milano',
    'na':'Napoli',
}
```

and you want to MODIFY it so that after your code the acronyms are added as capitalized:

```
>>> print(provinces)
{'tn': 'Trento',
 'mi':'Milano',
 'na':'Napoli',
 'TN': 'Trento',
 'MI':'Milano',
 'NA':'Napoli',
}
```

You might think to write something like this:

```
for key in provinces:
    provinces[key.upper()] = provinces[key]     # WARNING !
```

QUESTION: Do you see any problem?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: if you go through a dictionary and in *the meanwhile* you keep adding pieces, there is a concrete risk we will never terminate examining the keys !

So carefully read what follows:

</div>

X COMMANDMENT[260]: **You shall never ever add nor remove elements from a dictionary you are iterating with a `for` !**

In this case, if we try executing the code, we will get an explicit error:

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-26-9b20900057e8> in <module>()
----> 1 for key in provinces:
```

(continues on next page)

[260] https://en.softpython.org/commandments.html#X-COMMANDMENT

(continued from previous page)

```
      2     provinces['chiave'.upper()] = provinces[key]     # WARNING !

RuntimeError: dictionary changed size during iteration
```

but in other cases (like for example lists) modifying stuff **may produce totally unpredictable behaviours** (do you know the expression *pulling the rug out from under your feet ?*)

What about removing? We've seen adding is dangerous, but so is removing.

Suppose we want to remove any couple having as value `'Trento'`

```
provinces = {
    'tn': 'Trento',
    'mi':'Milano',
    'na':'Napoli',
}
```

to obtain:

```
>>> print(provinces)
{'mi':'Milano',
 'na':'Napoli'}
```

If we try executing something like this Python notices and raises an exception:

```
provinces = {
    'tn': 'Trento',
    'mi':'Milano',
    'na':'Napoli',
}

for key in provinces:
    if provinces[key] == 'Trento':
        del provinces[key]    # VERY BAD IDEA
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-23-5df0fd659120> in <module>()
      5     'na':'Napoli'
      6 }
----> 7 for key in provinces:
      8     if provinces[key] == 'Trento':
      9         del provinces[key]    # VERY BAD IDEA

RuntimeError: dictionary changed size during iteration
```

If you really need to remove elements from the sequence in which you are iterating, use a while cycle[261] or first copy the original sequence.

[261] https://en.softpython.org/control-flow/flow3-while-sol.html

Exercise - zazb

⊛⊛ Write some code which given a dictionary `chars` with characters as keys, MODIFY the dictionary so to add keys like the existing ones prefixed with character `'z'` - new keys should be associated with the constant integer `10`

Example - given:

```
chars = {
    'a':3,
    'b':8,
    'c':4
}
```

after your code, `chars` should result MODIFIED like this:

```
>>> chars
{   'a':3,
    'b':8,
    'c':4,
    'za':10,
    'zb':10,
    'zc':10
}
```

QUESTION: Is it desirable to write a solution like the following one? Read carefully!

```
chars = {
    'a':3,
    'b':8,
    'c':4
}

for key in chars:
    chars['z'+key] = 10     # WARNING !! TROUBLE AHEAD !!
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Absolutely not - in this case we're lucky and we will obtain an explicit error, in other cases we might obtain infinite loops or incomprehensible results:

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-36-550c4c302120> in <module>()
      5 }
      6
----> 7 for key in chars:
      8     chars['z'+key] = 10

RuntimeError: dictionary changed size during iteration
```

Do something better: try now rewriting a version of the program without this bug:

</div>

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[54]:
```python
chars = {
    'a':3,
    'b':8,
    'c':4
}

# write here

for el in list(chars.keys()):  # list 'takes a picture' of the current keys state
    chars['z'+el] = 10

#chars
```

</div>

[54]:
```python
chars = {
    'a':3,
    'b':8,
    'c':4
}

# write here
```

5.15.9 Nested `for`

Pay attention when using names of variables in nested `for` loops:

II COMMANDMENT[262] **Whenever you insert a variable in a `for` cycle, such variable must be new**

If you defined a variable in an external `for`, you shall not reintroduce it in an internal `for`, because this would bring a lot of confusion. For example here `s` is introduced both in the external and in the internal loop:

[55]:
```python
for s in ['volleyball', 'tennis', 'soccer', 'swimming']:

    for s in range(3):  # debugging hell, you lose the external cycle s
        print(s)

    print(s)  # prints 2 instead of a sport!
```

```
0
1
2
2
0
1
2
2
0
1
```

(continues on next page)

262 https://en.softpython.org/commandments.html#II-COMMANDMENT

(continued from previous page)

```
2
2
0
1
2
2
```

Questions - nested for

Look at the following code fragments , and for each try guessing the result it produces (or if it gives an error):

1. ```python
   for y in for x in range(3):
       print(x,y)
   ```

2. ```python
   for y in for x in range(2) in range(3):
       print(x,y)
   ```

3. ```python
   for y in range(3):
       for x in range(2):
           print(x,y)
   ```

4. ```python
   for x in range(2):
       for x in range(3):
           print(x)
       print(x)
   ```

5. ```python
   for x in range(2):
       for y in range(3):
           print(x,y)
       print(x,y)
   ```

6. ```python
   for x in range(1):
       for y in range(1):
           print(x,y)
   ```

7. ```python
   for x in range(2):
       for y in range(3):
           print(x,y)
   ```

8. ```python
   la = 'abc'
   for x in la:
       for y in la:
           print(x)
   ```

9. ```python
   for x in 'ab':
       for y in 'cd':
           print(x,y)
       for y in 'ef':
           print(x,y)
   ```
   ```

10.
```
for x in 'abc':
    for y in 'abc':
        if x == y:
            print(x)
```

11.
```
for x in 'abc':
    for y in 'abc':
        if x != y:
            print(x,y)
```

12.
```
lista = []
for x in 'a':
    for y in 'bc':
        lista.append(x)
        lista.append(y)
print(lista)
```

13.
```
lista = []
for x in 'abc':
    for y in 'de':
        lista.append('z')
print(len(lista))
```

14.
```
c = 1
for x in range(1,4):
    s = ''
    for y in range(1,4):
        s = s + str(c)
        c += 1
    print(s)
```

Exercise - casting

⊕ A new USA-Japanese videocultural production is going to be launched, so actors are called for casting. The director wants to try a scene with all the possible couples which can be formed among actors and actresses. Write some code which prints all the couples, also putting introduction messages.

- **NOTE**: the number of actors and actresses may be different

Exemple - given:

```
actresses = ['Leela','Wilma']
actors = ['Capitan Harlock', 'Lupin', 'Kenshiro']
```

prints:

```
Leela enters the scene!
   Capitan Harlock enters the scene!
      Leela and Capitan Harlock get ready ... ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Leela and Lupin get ready ... ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Leela and Kenshiro get ready ... ACTION!
```

(continues on next page)

(continued from previous page)

```
    Thanks Kenshiro - next one !
Thanks Leela - next one !
Wilma enters the scene!
   Capitan Harlock enters the scene!
      Wilma and Capitan Harlock get ready ... ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Wilma and Lupin get ready ... ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Wilma and Kenshiro get ready ... ACTION!
    Thanks Kenshiro - next one !
Thanks Wilma - next one !

Casting is over for today!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[56]:

```python
actresses = ['Leela','Wilma']
actors = ['Capitan Harlock', 'Lupin', 'Kenshiro']


# write here
for actress in actresses:
    print(actress, 'enters the scene!')
    for actor in actors:
        print('  ',actor, 'enters the scene!')

        print('     ',actress, 'and', actor, 'get ready ... ACTION!')
        print('    Thanks', actor, '- next one !')
    print('Thanks', actress, '- next one !')
print()
print('Casting is over for today!')
```

```
Leela enters the scene!
   Capitan Harlock enters the scene!
      Leela and Capitan Harlock get ready … ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Leela and Lupin get ready … ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Leela and Kenshiro get ready … ACTION!
    Thanks Kenshiro - next one !
Thanks Leela - next one !
Wilma enters the scene!
   Capitan Harlock enters the scene!
      Wilma and Capitan Harlock get ready … ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Wilma and Lupin get ready … ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Wilma and Kenshiro get ready … ACTION!
    Thanks Kenshiro - next one !
Thanks Wilma - next one !
```

(continues on next page)

(continued from previous page)

```
Casting is over for today!
```

</div>

[56]:

```python
actresses = ['Leela','Wilma']
actors = ['Capitan Harlock', 'Lupin', 'Kenshiro']


# write here
```

```
Leela enters the scene!
   Capitan Harlock enters the scene!
      Leela and Capitan Harlock get ready … ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Leela and Lupin get ready … ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Leela and Kenshiro get ready … ACTION!
    Thanks Kenshiro - next one !
Thanks Leela - next one !
Wilma enters the scene!
   Capitan Harlock enters the scene!
      Wilma and Capitan Harlock get ready … ACTION!
    Thanks Capitan Harlock - next one !
   Lupin enters the scene!
      Wilma and Lupin get ready … ACTION!
    Thanks Lupin - next one !
   Kenshiro enters the scene!
      Wilma and Kenshiro get ready … ACTION!
    Thanks Kenshiro - next one !
Thanks Wilma - next one !

Casting is over for today!
```

Exercise - cover the plane

⊛ Given the integers `a` and `b`, write some code which prints all the possible couples of numbers x and y such that $1 \leq x \leq a$ and $1 \leq y \leq b$

For example, given:

```python
a,b = 5,3
```

it must print:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
```

(continues on next page)

(continued from previous page)

```
3 2
3 3
4 1
4 2
4 3
5 1
5 2
5 3
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[57]:
```python
a,b = 5,3

# write here
for x in range(1,a+1):
    for y in range(1,b+1):
        print(x,y)
```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
5 1
5 2
5 3
```

</div>

[57]:
```python
a,b = 5,3

# write here
```

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
4 1
4 2
4 3
5 1
5 2
5 3
```

Exercise - triangular

⊕ Given the integer `a`, write some code which prints all the possible couples of numbers x and y such that $0 \leq x \leq y < a$

For example, for

```
a = 5
```

it must print:

```
0 0
0 1
0 2
0 3
0 4
1 1
1 2
1 3
1 4
2 2
2 3
2 4
3 3
3 4
4 4
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[58]:
```
a = 5
# write here
for x in range(a):
    for y in range(x,a):
        print(x,y)
```

```
0 0
0 1
0 2
0 3
0 4
1 1
1 2
1 3
1 4
2 2
2 3
2 4
3 3
3 4
4 4
```

</div>

[58]:
```
a = 5
# write here
```

```
0 0
0 1
0 2
0 3
0 4
1 1
1 2
1 3
1 4
2 2
2 3
2 4
3 3
3 4
4 4
```

Exercise - port

⊛ Write some code which given a list `words` and a list `characters`, for each word calculates how many characters it contains

- **ONLY** count the characters present in `characters`
- **ONLY** print the result if the number is greater than zero

Example - given:

```
words = ['ships','pier','oar','fish trap','sails','trawling net']
characters = ['n','i','s']
```

prints:

```
ships contains 1 i
ships contains 2 s
pier contains 1 i
fish trap contains 1 i
fish trap contains 1 s
sails contains 1 i
sails contains 2 s
trawling net contains 2 n
trawling net contains 1 i
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[59]:
```
words = ['ships','pier','oar','fish trap','sails','trawling net']
characters = ['n','i','s']


# write here

for x in words:
    for y in characters:
        if y in x:
            print(x,'contains',x.count(y),y)
```

```
ships contains 1 i
ships contains 2 s
pier contains 1 i
fish trap contains 1 i
fish trap contains 1 s
sails contains 1 i
sails contains 2 s
trawling net contains 2 n
trawling net contains 1 i
```

</div>

[59]:
```python
words = ['ships','pier','oar','fish trap','sails','trawling net']
characters = ['n','i','s']


# write here
```

```
ships contains 1 i
ships contains 2 s
pier contains 1 i
fish trap contains 1 i
fish trap contains 1 s
sails contains 1 i
sails contains 2 s
trawling net contains 2 n
trawling net contains 1 i
```

Exercise - polygons

⊛⊛ Given a list `polygons` with polygon names ordered by sides number starting from a triangle, write some code which prints all the possible questions we can form regarding the number of sides. Start from a minimum of 3 sides until a maximum corresponding to the number of sides of the last polygon (remember names are ordered by number of sides!)

Example - given:

```python
#                0           1          2            3
polygons = ["triangle","square","pentagon",  "hexagon"]
```

prints:

```
Does the triangle have 3 sides? True
Does the triangle have 4 sides? False
Does the triangle have 5 sides? False
Does the triangle have 6 sides? False
Does the square have 3 sides? False
Does the square have 4 sides? True
Does the square have 5 sides? False
Does the square have 6 sides? False
Does the pentagon have 3 sides? False
Does the pentagon have 4 sides? False
Does the pentagon have 5 sides? True
Does the pentagon have 6 sides? False
Does the hexagon have 3 sides? False
```

(continues on next page)

(continued from previous page)

```
Does the hexagon have 4 sides? False
Does the hexagon have 5 sides? False
Does the hexagon have 6 sides? True
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[60]:
```python
#                 0              1              2              3
polygons = ["triangle","square","pentagon", "hexagon"]

# write here
for i in range(len(polygons)):
    for j in range(len(polygons)):
        print('Does the', polygons[i],'have', j+3, 'sides?', i+3 == j+3 )
```

```
Does the triangle have 3 sides? True
Does the triangle have 4 sides? False
Does the triangle have 5 sides? False
Does the triangle have 6 sides? False
Does the square have 3 sides? False
Does the square have 4 sides? True
Does the square have 5 sides? False
Does the square have 6 sides? False
Does the pentagon have 3 sides? False
Does the pentagon have 4 sides? False
Does the pentagon have 5 sides? True
Does the pentagon have 6 sides? False
Does the hexagon have 3 sides? False
Does the hexagon have 4 sides? False
Does the hexagon have 5 sides? False
Does the hexagon have 6 sides? True
```

</div>

[60]:
```python
#                 0              1              2              3
polygons = ["triangle","square","pentagon", "hexagon"]

# write here
```

```
Does the triangle have 3 sides? True
Does the triangle have 4 sides? False
Does the triangle have 5 sides? False
Does the triangle have 6 sides? False
Does the square have 3 sides? False
Does the square have 4 sides? True
Does the square have 5 sides? False
Does the square have 6 sides? False
Does the pentagon have 3 sides? False
Does the pentagon have 4 sides? False
Does the pentagon have 5 sides? True
Does the pentagon have 6 sides? False
Does the hexagon have 3 sides? False
Does the hexagon have 4 sides? False
Does the hexagon have 5 sides? False
Does the hexagon have 6 sides? True
```

Exercise - bon jour

⊛⊛⊛ Given two strings `sa` and `sb` in lowercase, write some code which prints single letters from `sa` as upper case, followed by all possible combinations of `sb` where ONLY ONE character is uppercase.

Example - given:

```
sa = 'bon'
sb = 'jour'
```

Must print:

```
B Jour
B jOur
B joUr
B jouR
O Jour
O jOur
O joUr
O jouR
N Jour
N jOur
N joUr
N jouR
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[61]:
```python
sa = 'bon'
sb = 'jour'

# write here

for c1 in sa:
    for i in range(len(sb)):
        print(c1.upper() + ' ' + sb[:i] + sb[i].upper() + sb[i+1:])
```

```
B Jour
B jOur
B joUr
B jouR
O Jour
O jOur
O joUr
O jouR
N Jour
N jOur
N joUr
N jouR
```

</div>

[61]:
```python
sa = 'bon'
sb = 'jour'

# write here
```

```
B Jour
B jOur
B joUr
B jouR
O Jour
O jOur
O joUr
O jouR
N Jour
N jOur
N joUr
N jouR
```

5.15.10 `break` and `continue` commands

We can use the commands `break` and `continue` to have even more control on loop execution.

NOTE: Please use sparingly!

When there is a lot of code in the cycle it's easy to 'forget' about their presence and introduce hard-to-discover bugs. On the other hand, in some selected cases these commands *may* increase code readability, so as everything use your judgement.

Terminate with `break`

To immediately exit a cycle you can use the `break` command:

[62]:
```python
for x in 'PARADE':

    if x == 'D':
        print('break, exits the loop!')
        break
        print('After the break')

    print(x)

print('Loop is over !')
```

```
P
A
R
A
break, exits the loop!
Loop is over !
```

Note how the instruction which prints `'After the break'` was *not* executed

Jumping with `continue`

By calling `continue` execution is immediately brough to the next iteration , so we jump to the next element in the sequence without executing the instructions after the `continue`.

[63]:
```python
i = 1
for x in 'PARADE':
    if x == 'A':
        print("continue, jumps to next element")
        continue
    print(x)
print('Loop is over !')
```

```
P
continue, jumps to next element
R
continue, jumps to next element
D
E
Loop is over !
```

Combining `break` and `continue`

Let's see both in Python Tutor:

[64]:
```python
i = 1
for x in 'PARADE':
    if x == 'A':
        print("continue, jumps to next element")
        continue
    if x == 'D':
        print('break, exits loop!')
        break
    print(x)

print('Loop is over !')

jupman.pytut()
```

```
P
continue, jumps to next element
R
continue, jumps to next element
break, exits loop!
Loop is over !
```

[64]:
```
<IPython.core.display.HTML object>
```

Questions - break and continue

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1. ```
   for x in ['a','b','c']:
       print(x)
       break
   ```

2. ```
   for x in ['a','b','c']:
       print(x)
       break
       print('GLAM')
   ```

3. ```
   for x in ['a','b','c']:
       print(x)
       break
       break
   ```

4. ```
   for x in ['a','b','c']:
       break
       print(x)
   ```

5. ```
   break
   for x in ['a','b','c']:
       print(x)
   ```

6. ```
   for x in ['a','b','c']:
       print(x)
   break
   ```

7. ```
   for x in ['a','b','c']:
       continue
       print(x)
   ```

8. ```
   for x in ['a','b','c']:
       print(x)
       continue
   ```

9. ```
   for x in ['a','b','c']:
       print(x)
       continue
       print('BAM')
   ```

10. ```
    continue
    for x in ['a','b','c']:
        print(x)
    ```

11. ```
    for x in ['a','b','c']:
        print(x)
    continue
    ```

12. ```
    for x in ['a','b','c']:
        break
    ```

(continues on next page)

(continued from previous page)

```
    1/0
print('BAD KARMA')
```

13.
```
for x in ['a','b','c']:
    1/0
    break
print('BAD KARMA')
```

14.
```
for x in range(8):
    if x < 4:
        continue
    print('ZAM', x)
```

15.
```
for x in range(8):
    if x >= 4:
        break
    print('ZUM', x)
```

16.
```
for x in range(6):
    if x % 2 == 0:
        continue
    print(x)
```

17.
```
for x in ['M','C','M']:
    print(x)
    for y in ['S','P','Q','R']:
        print(y)
        break
```

18.
```
for x in ['M','C','M']:
    print(x)
    break
    for y in ['S','P','Q','R']:
        print(y)
```

19.
```
for x in ['M','C','M']:
    print(x)
    for y in ['S','P','Q','R']:
        print(y)
        continue
```

20.
```
for x in ['M','C','M']:
    print(x)
    continue
    for y in ['S','P','Q','R']:
        print(y)
```

Exercise - autonomous walking

⊕ Write some code which given a string `phrase`, prints all the characters *except* the vocals.

Example - given:

```
phrase = 'autonomous walking'
```

prints:

```
t
n
m
s

w
l
k
n
g
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[65]:
```python
phrase = 'autonomous walking'
#phrase='continuous'

# write here
for x in phrase:
    if x in 'aeiou':
        continue
    else:
        print(x)
```

```
t
n
m
s

w
l
k
n
g
```

</div>

[65]:
```python
phrase = 'autonomous walking'
#phrase='continuous'

# write here
```

```
t
n
m
s
```

(continues on next page)

(continued from previous page)

```
w
l
k
n
g
```

Exercise - breaking bad

⊕ Write some code which prints all the charactes from `string` until it finds the string `'bad'`.

Example - given:

```python
string = 'cascapirillabadgnippobadzarpogno'
```

prints

```
c
a
s
c
a
p
i
r
i
l
l
a
```

[66]:
```python
string = 'cascapirillabadgnippobadzarpogno'   # cascapirilla
#string = 'sobad'   # 'so'
#string = 'bad'   # ''
#string = 'badso'   # ''

for i in range(len(string)):
    if string[i:i+3] == 'bad':
        break
    else:
        print(string[i])
```

```
c
a
s
c
a
p
i
r
i
l
l
a
```

Exercise - breaking point

⊛⊛ Given a phrase, prints all the words one per row *until* it finds a dot, and in that case it stops.

- **DO NOT** use `phrase.split('.')`. Splits on other characters are allowed.

Example - given:

```
phrase = 'At some point you must stop. Never go beyond the limit.
```

prints:

```
At
some
point
you
must
stop
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[67]:
```python
phrase = 'At some point you must stop. Never go beyond the limit.'
#phrase = "Respect the halt. Do you want to have us arrested?"
#phrase = 'Stop.'
#phrase = 'No stop'

# write here

for word in phrase.split():
    if '.' in word:
        print(word[:-1])
        break
    else:
        print(word)
```

```
At
some
point
you
must
stop
```

</div>

[67]:
```python
phrase = 'At some point you must stop. Never go beyond the limit.'
#phrase = "Respect the halt. Do you want to have us arrested?"
#phrase = 'Stop.'
#phrase = 'No stop'

# write here

```

```
At
some
point
you
must
stop
```

Exercise - breakdance

⊛⊛ As a skilled breakdancer, you're given `music` as a list of sounds. You will have to perform a couple of dances:

- during the first one, you will have to repeat the music sounds until you find exactly `3` sounds `'pa'`, then you will shout `BREAKDANCE!`.
- during the second one, you will have to repeat the music sounds *in reverse* until you find exactly `3` sounds `'pa'`, then you will shout `BREAKDANCE!`
- **DO NOT** modify `music`, so no `music.reverse()`

Example - given:

```
music = ['unz','pa','pa','tud','unz','pa','pa','tud','unz','boom','boom','tud']
```

Prints:

```
unz
pa
pa
tud
unz
pa
BREAKDANCE!

tud
boom
boom
unz
tud
pa
pa
unz
tud
pa
BREAKDANCE!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[68]:
```
music = ['unz','pa','pa','tud','unz','pa','pa','tud','unz','boom','boom','tud']

# write here

k = 0
for x in music:
    print(x)
    if x == 'pa':
        k += 1
    if k == 3:
        print('BREAKDANCE!')
        print()
        break

k = 0
for x in range(len(music)-1, -1, -1):
    print(music[x])
    if music[x] == 'pa':
```

(continues on next page)

(continued from previous page)

```
        k += 1
    if k == 3:
        print('BREAKDANCE!')
        break
```

```
unz
pa
pa
tud
unz
pa
BREAKDANCE!

tud
boom
boom
unz
tud
pa
pa
unz
tud
pa
BREAKDANCE!
```

</div>

[68]:
```
music = ['unz','pa','pa','tud','unz','pa','pa','tud','unz','boom','boom','tud']

# write here

```

```
unz
pa
pa
tud
unz
pa
BREAKDANCE!

tud
boom
boom
unz
tud
pa
pa
unz
tud
pa
BREAKDANCE!
```

[ ]:
```

```

5.16 Control flow - while loop

5.16.1 Download exercises zip

Browse online files[263]

Let's see how to repeat instructions by executing them inside `while` loops.

The main feature of `while` loop is allowing to explicitly control when the loop should end. Typically, such loops are used when we must *iterate* on a sequence of which we don't know the dimension, or it can vary over time, or several conditions might determine the cycle stop.

5.16.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
control-flow
    flow1-if.ipynb
    flow1-if-sol.ipynb
    flow2-for.ipynb
    flow2-for-sol.ipynb
    flow3-while.ipynb
    flow3-while-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `flow3-while.ipynb`
3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ⊛ to ⊛⊛⊛⊛ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.16.3 Introduction

A `while` cycle is a code block which is executed when a certain boolean condition is verified. The code block is repeatedly executed as long as the condition is true.

Let's see an example:

[263] https://github.com/DavidLeoni/softpython-en/tree/master/control-flow

[2]:
```python
i = 1

while i < 4:
    print('Counted', i)
    i += 1

print('Loop is over!')
```

```
Counted 1
Counted 2
Counted 3
Loop is over!
```

In the example, the boolean condition is

```python
i < 4
```

the block to repeatedly executed is

```python
print('Counted', i)
i += 1
```

Like any Python code blocks, the block is indented with spaces (usually 4).

Have a better look at the execution in Python Tutor and read the following comment.

[3]:
```python
# WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#           (it's sufficient to execute it only once)

import jupman
```

[4]:
```python
i = 1
while i < 4:
    print('Counted', i)
    i += 1

print('Loop is over !')

jupman.pytut()
```

```
Counted 1
Counted 2
Counted 3
Loop is over !
```

[4]:
```
<IPython.core.display.HTML object>
```

In the example we used a variable we called `i` and we initialized it to zero.

At the beginning of the cycle `i` is valued `1`, so the boolean expression `i < 4` is evaluated as `True`. Since it's `True`, execution continues inside the block with the `print` and finally MODIFIES `i` by incrementing `i += 1`.

Now the execution goes to `while` row, and condition `i < 4` is evaluated again. At this second iteration `i` is valued `2`, so the boolean expression `i < 4` is again evaluated to `True` and the execution remains inside the block. A new print is done and `i` gets incremented.

Another loop is done until `i` is valued `4`. A that point `i < 4` produces `False` so in that moment execution *exits* the `while` block and goes on with the commands at the same indentation level as the `while`

Terminating `while`

When we have a `while` cycle, typically sooner or later we want it to terminate (programs which hang aren't users' favourites …). To guarantee termination, we need:

1. initializing a variable outside the cycle
2. a condition after the `while` command which evaluates that variable (and optionally other things)
3. at least one instruction in the internal block which MODIFIES the variable, so that sooner or later is going to satisfy condition 2

If any of these points is omitted, we will have problems. Let's try forgetting them on purpose:

Error 1: omit initialization. As in those cases in Python where we forgot to initialize a variable (let's try `j` in this case), the execution is interrupted as soon we try using the variable:

```python
print("About to enter the cycle ..")
while j < 4:
    print('Counted', j)
    j += 1

print('Loop is over !')
```

```
About to enter the cycle ..
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-277-3f311955204d> in <module>()
      1 print("About to enter the cycle ..")
----> 2 while j < 4:
      3     print('Counted', j)
      4     j += 1
      5

NameError: name 'j' is not defined
```

Error 2: omit using the variable in the condition. If we forget to evaluate the variable, for example using another one by mistake (say `x`), the loop will never stop:

```python
i = 1
x = 1
print('About to enter the cycle ..')
while x < 4:   # evalutes x instead of i
    print('Counted', i)
    i += 1

print('Loop is over !')
```

```
About to enter the cycle ..
Counted 1
Counted 2
Counted 3
Counted 4
Counted 5
Counted 6
.
.
```

Error 3: Omit to MODIFY the variable in the internal block. If we forget to place at least one instruction which MODIFIES the variable used in the condition, whenever the condition is evaluated it will always produce the same boolean value `False` preventing a cycle exit:

```
i = 1
print('About to enter the cycle ..')
while i < 4:
    print('Counted', i)

print('Loop is over !')
```

```
About to enter the cycle ..
Counted 1
Counted 1
Counted 1
Counted 1
Counted 1
.
.
```

Non terminating `while`

QUESTION: Can you imagine a program which *never* terminates?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: if you live nearby a hydropower or nuclear plant, what happens if the program regulating the water stops?

Or: suppose you are inside an airplane and the program which checks the fuel flux to the engine suddenly stops. Could this be a problem?

All programs if well written must foresee termination, but some software are executed for such a long time that termination is to be considered an exceptional event.

</div>

Questions

QUESTION: Look at the following code fragments , and for each try guessing the result it produces (or if it gives an error):

1. ```
   i = 0
   while i < 3:
   print(i)
   ```

2. ```
   k = 0
   while k < 5:
       print(k)
       k + 1
   ```

3. ```
   i = 0
   while i < 3:
       print(i)
   i += 1
   ```
   ```

4.
```python
i = 0
while False:
    print(i)
    i += 1
print('Done !')
```

5.
```python
i = 0
while i < 3:
    print(i)
    i += 1
```

6.
```python
k = 0
while k < 2
    print(i)
    k += 1
```

7.
```python
i = 0
while i < 3:
    print('GAM')
    i = i + 1
```

8.
```python
while zanza < 2
    print('ZANZA')
    zanza += 1
```

9.
```python
i = 0
while False:
    print(i)
    i = i + 1
print('DARK')
```

10.
```python
i = 0
while True:
    print(i)
    i = i + 1
print('LIGHT')
```

11.
```python
while 2 + 3:
    print('z')
print('')
```

12.
```python
i = 10
while i > 0:
    if i > 5:
        print(i)
        i -= 1
print('WAM')
```

13.
```python
i = 10
while i > 0:
    if i > 5:
        print(i)
    i -= 1
print('MAW')
```

14.
```
import random
x = 0
while  x < 7:
    x = random.randint(1,10)
    print(x)

print('LUCK')
```

15.
```
x,y = 0,0
while x + y < 4:
    x += 1
    y += 1
    print(x,y)
```

16.
```
x,y = 0,3
while x < y:
    print(x,y)
    x += 1
    y -= 1
```

Esercises - introduction

Exercise - printeven

⊕ Write some code to print all the odd numbers from `1` to `k` in a `while` cycle

- for `k<1` prints nothing

Example - given:

```
k = 5
```

after your code it must print:

```
1
3
5
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[5]:
```
k = 5    # 1 3 5
#k = 1   # 1
#k = 0   # no print

# write here
i = 1
while i <= k:
    if i % 2 == 1:
        print(i)
    i += 1
```

```
1
3
5
```

</div>

[5]:
```
k = 5   # 1 3 5
#k = 1  # 1
#k = 0  # no print

# write here
```

```
1
3
5
```

Exercise - average

⊕ Write some code that given a list `numbers`, calculates the average of values using a `while` and then prints it.

- if the list is not empty, the average is supposed to be `0.0`
- **DO NOT** use the function `sum`
- DO NOT create variables called `sum` (would violate the V COMMANDMENT[264]: you shall never ever redefine system functions)

Example - given:

```
numbers = [8,6,5,9]
```

prints

```
7.0
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[6]:
```
numbers = [8,6,5,9] # 7.0
#numbers = [3,1,2]   # 2.0
#numbers = []        # 0

# write here
s = 0.0
i = 0
while i < len(numbers):
    s += numbers[i]
    i += 1

if len(numbers) > 0:
    print(s / len(numbers))
else:
    print(0.0)
```

```
7.0
```

</div>

264 https://en.softpython.org/commandments.html#V-COMMANDMENT

[6]:
```python
numbers = [8,6,5,9] # 7.0
#numbers = [3,1,2]  # 2.0
#numbers = []       # 0

# write here

```

```
7.0
```

5.16.4 `break` and `continue` commands

For getting even more control on cycle execution we can use the commands `break` and `continue`

NOTE: Use them sparingly!

When there is a lot of code in the cycle it's easy to 'forget' about their presence and introduce hard-to-discover bugs. On the other hand, in some selected cases these commands may increase code readability, so as everything use your judgement.

Terminate with a `break`

The scheme we've seen to have a terminating `while` is the recommended one, but if we have a condition which does NOT evaluate the variable we are incrementing (like for example the constant expression `True`), as an alternative to immediatly exit the cycle we can use the command `break`:

[7]:
```python
i = 1
while True:

    print('Counted', i)

    if i > 3:
        print('break! Exiting the loop!')
        break
        print('After the break')

    i += 1

print('Loop is over !')
```

```
Counted 1
Counted 2
Counted 3
Counted 4
break! Exiting the loop!
Loop is over !
```

Note `After the break` is *not* shown.

Jumping with `continue`

We can bring the execution immediately to the next iteration by calling `continue`, which directly jumps to the condition check without executing the instructions after the `continue`.

> **WARNING: `continue` instructions if used carelessly can cause infinite loops !**
>
> When using `continue` ensure it doesn't jump the instriction which modifies the variable used in the termination condition (or it doesn't jump a `break` needed for exiting the cycle)!

To avoid problems here we incremented `i` before the `if` with a `continue`:

[8]:
```python
i = 1
while i < 5:
    print('Counted', i)

    i += 1

    if i % 2 == 1:
        print('continue, jumping to condition check')
        continue
        print('After the continue')

    print('arrived till the end')

print('Loop is over !')
```

```
Counted 1
arrived till the end
Counted 2
continue, jumping to condition check
Counted 3
arrived till the end
Counted 4
continue, jumping to condition check
Loop is over !
```

Let's try combining `break` and `continue`, and see what happens in Python Tutor:

[9]:
```python
i = 1
while i < 5:
    print('Counted', i)
    if i > 3:
        print('break! Exiting the cycle!')
        break
        print('After the break')
    i += 1
    if i % 2 == 1:
        print('continue, jumping to next condition check')
        continue
        print('After the continue')
    print('arrived till the end')

print('Loop is over !')

jupman.pytut()
```

```
Counted 1
arrived till the end
Counted 2
continue, jumping to next condition check
Counted 3
arrived till the end
Counted 4
break! Exiting the cycle!
Loop is over !
```

[9]:
```
<IPython.core.display.HTML object>
```

Questions about `break` and `continue`

QUESTION: Look at the following code fragments , and for each try guessing the result it produces (or if it gives an error):

1.
```python
i = 1
while i < 4:
    print('Counted', i)
    i += 1
    continue

print('Loop is over !')
```

2.
```python
i = 1
while i < 4:
    print('Counted', i)
    continue
    i += 1

print('Loop is over !')
```

3.
```python
i = 3
while i > 0:
    print('Counted', i)
    if i == 2:
        print('continue, jumping to condition check')
        continue
    i -= 1
    print('arrived till the end')

print('Loop is over !')
```

4.
```python
i = 0
while True:
    i += 1
    print(i)
    if i > 3:
        break

print('BONG')
```

5.
```python
i = 0
while True:
```

(continues on next page)

(continued from previous page)

```
    if i < 3:
        continue
    else:
        break
    i += 1

print('ZONG')
```

6.
```
i = 0
while True:
    i += 1
    if i < 3:
        continue
    else:
        break

print('ZANG')
```

Exercise - findchar

⊛⊛ Write some code that using a `while` searches the character list `la` for a character specified in `car` variable. As soon as the FIRST character occurrence is found, it stops and prints the index where it was found.

- if it doesn't find the character, prints a Not found message.

Example 1 - given:

```
car = 'z'
la = ['b','a','f','g','z','h','z','r']
```

after your code it must print:

```
Found first z at index 4
```

Example 2 - given:

```
car = 'z'
la = ['b','a','f','g','h','r']
```

must print:

```
Didn't find z
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:
```

car = 'z'

#       0   1   2   3   4   5   6   7
la = ['b','a','f','g','z','h','z','r']    # Found first z at index 4
#la = ['b','a','f','g','h','r']           # Didn't find z


# write here
```

(continues on next page)

(continued from previous page)

```
i = 0
while i < len(la):
    if la[i] == car:
        print("Found first", car, "at index", i)
        break
    i += 1
if i == len(la):
    print("Didn't find", car)
```

```
Found first z at index 4
```

</div>

[10]:

```
car = 'z'

#        0   1   2   3   4   5   6   7
la = ['b','a','f','g','z','h','z','r']   # Found first z at index 4
#la = ['b','a','f','g','h','r']          # Didn't find z


# write here
```

```
Found first z at index 4
```

5.16.5 Questions - Are they equivalent?

Look at the following code fragments: each contains two parts, A and B. For each value of the variables they depend on, try guessing whether part A will print exactly the same result printed by code in part B

- **FIRST** think about the answer
- **THEN** try executing with each of the values of suggested variables

Are they equivalent? - BORG

```
print('A:')
while True:
    print('BORG')
    break

print('\nB:')
while False:
    pass
print('BORG')
```

Are they equivalent? - until 3

```
print('A:')
x = 0
while x < 3:
    print(x)
    x += 1

print('\nB:')
x = 1
while x <= 3:
    print(x-1)
    x += 1
```

Are they equivalent? - by chance

Remember `randint(a, b)` gives back a random integer `N` such that `a <= N <= b`

```
print('A:')
x = 0
while x < 3:
    x += 1
print(x)

print('\nB:')
x = 0
import random
while x != 3:
    x = random.randint(1,5)
print(x)
```

Are they equivalent? - until six

```
print('A:')
i = 0
while i < 3:
    print(i)
    i += 1
while i < 6:
    print(i)
    i += 1

print('\nB:')
i = 0
while i < 6:
    print(i)
    i += 1
```

Are they equivalent? - countdown 1

```
print('A:')
i = 2
print(i)
while i > 0:
    i -= 1
    print(i)

print('\nB:')
i = 2
while i > 0:
    print(i)
    i -= 1
```

Are they equivalent? - countdown 2

```
print('A:')
i = 2
print(i)
while i > 0:
    i -= 1
    print(i)

print('\nB:')
i = 2
while i > 0:
    print(i)
    i -= 1
print(i)
```

Are they equivalent? - sorcery

```
print('A:')
s = 'sorcery'
i = 0
while s[i] != 'g':
    i += 1
print(s[i:])

print('B:')
s = 'sorcery'
i = len(s)
while s[i] != 'g':
    i -= 1
print(s[i:])
```

Are they equivalent? - ping pong

```
print('A:')
ping,pong = 0,3
while ping < 3 or pong > 0:
    print(ping,pong)
    ping += 1
    pong -= 1

print('\nB:')
ping,pong = 0,3
while not(ping >= 3 and pong <= 0):
    print(ping,pong)
    ping += 1
    pong -= 1
```

Are they equivalent? - zanna

```
print('A:')
n,i,s = 0,0,'zanna'
while i < len(s):
    if s[i] == 'n':
        n += 1
    i += 1
print(n)

print('\nB:')
n,i,s = 0,0,'zanna'
while i < len(s):
    i += 1
    if s[i-1] == 'n':
        n += 1
print(n)
```

Are they equivalent? - pasticcio

```
print('A:')
c,i,s = 0,0,'pasticcio'
while i < len(s):
    if s[i] == 'c':
        c += 1
    i += 1
print(c)

print('\nB:')
no,k,s = 0,0,'pasticcio'
while k < len(s):
    if s[k] != 'c':
        no += 1
    else:
        k += 1
print(len(s) - no)
```

5.16.6 Exercises - counters

Exercise - don't break 1

⊕ Look at the following code, and write in the following cell some code which produces the same result with a `while` and **without using** `break`

[11]:
```
x = 3
while True:
    print(x)
    if x == 0:
        break
    x -= 1
```

```
3
2
1
0
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```
x = 3

# write here

while x >= 0:
    print(x)
    x -= 1
```

```
3
2
1
0
```

</div>

[12]:
```
x = 3

# write here

```

```
3
2
1
0
```

Exercise - don't break 2

⊕ Look at the following code, and write in the following cell some code which produces the same result with a `while` and **without using** `break`

[13]:
```
la = [2,3,7,5,6]
k = 7    # 2 3 7
#k = 5   # 2 3 7 5 6
#k = 13 # 2 3 7 5 6

i = 0
while True:
    print(la[i])
    if i >= len(la)-1 or la[i] == k:
        break
    else:
        i += 1
```

```
2
3
7
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[14]:
```
la = [2,3,7,5,6]
k = 7    # 2 3 7
#k = 6   # 2 3 7 5 6
#k = 13 # 2 3 7 5 6

i = 0

# write here

while i < len(la) and la[i] != k:
    print(la[i])
    i += 1
if i < len(la) and la[i] == k:
    print(la[i])
```

```
2
3
7
```

</div>

[14]:
```
la = [2,3,7,5,6]
k = 7    # 2 3 7
#k = 6   # 2 3 7 5 6
#k = 13 # 2 3 7 5 6

i = 0

# write here
```

```
2
3
```

(continues on next page)

(continued from previous page)

```
7
```

Exercise - Give me a break

⊕ Look at the following code, and write in the next cell some code which produces the same result with a `while` **this time using a** `break`

[15]:

```python
x,y = 1,5     # (1,5)   (2,4)
#x,y = 2,8    # (2, 8) (3, 7) (4, 6)

while x < y or x == 4:
    print((x,y))
    x += 1
    y -= 1
```

```
(1, 5)
(2, 4)
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[16]:

```python
x,y = 1,5     # (1,5)   (2,4)
#x,y = 2,8    # (2, 8) (3, 7) (4, 6)

# write here
while True:

    if x >= y or x == 4:
        break
    else:
        print((x,y))
        x += 1
        y -= 1

if x < y or x == 4:
    print((x,y))
```

```
(1, 5)
(2, 4)
```

</div>

[16]:

```python
x,y = 1,5     # (1,5)   (2,4)
#x,y = 2,8    # (2, 8) (3, 7) (4, 6)

# write here
```

```
(1, 5)
(2, 4)
```

Exercise - paperboard

⊛ Prints integer numbers from `0` to `k` INCLUDED using a `while`, and for each number prints to its side one among the strings `'PA'`, `'PER'` and `'BOARD'` alternating them

Ex - for `k=8` prints

```
0 PA
1 PER
2 BOARD
3 PA
4 PER
5 BOARD
6 PA
7 PER
8 BOARD
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[17]:
```python
k = 8

# write here
x = 0
while x <= k:
    if x % 3 == 0:
        print(x, 'PA')
    elif x % 3 == 1:
        print(x, 'PER')
    else:
        print(x, 'BOARD')
    x += 1
```

```
0 PA
1 PER
2 BOARD
3 PA
4 PER
5 BOARD
6 PA
7 PER
8 BOARD
```

</div>

[17]:
```python
k = 8

# write here
```

```
0 PA
1 PER
2 BOARD
3 PA
4 PER
5 BOARD
6 PA
```

(continues on next page)

(continued from previous page)

```
7 PER
8 BOARD
```

Exercise - until ten

⊕ Given two numbers `x` and `y`, write some code with a `while` which prints and increments the numbers, stopping as soon as one of them reaches ten.

```
x,y = 5,7
```

after your code it must result:

```
5 7
6 8
7 9
8 10
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```python
x,y = 5,7
#x,y = 8,4

# write here
while x <= 10 and y <= 10:
    print(x,y)
    x += 1
    y += 1
```

```
5 7
6 8
7 9
8 10
```

</div>

[18]:
```python
x,y = 5,7
#x,y = 8,4

# write here
```

```
5 7
6 8
7 9
8 10
```

Exercise - cccc

⊕ Write some code using a `while` which given a number `y`, prints `y` rows containing the character `c` as many times as the row number.

Example - given:

```
y = 4
```

Prints:

```
c
cc
ccc
cccc
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[19]:
```python

y = 4

# write here
x = 0
while x <= y:
    print('c'*x)
    x += 1
```

```

c
cc
ccc
cccc
```

</div>

[19]:
```python

y = 4

# write here

```

```

c
cc
ccc
cccc
```

Exercise - converge

⊕ Given two numbers `x` and `k`, using a `while` modify and print `x` until it reaches `k` included

- **NOTE**: `k` can either be greater or lesser than `x`, you must handle both cases

Example 1 - given:

```
x,k = 3,5
```

prints:

```
3
4
5
```

Example 2 - given:

```
x,k = 6,2
```

prints:

```
6
5
4
3
2
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[20]:

```
x,k = 3,5     # 3 4 5
#x,k = 6,2    # 6 5 4 3 2
#x,k = 4,4    # 4

# write here

while x != k:
    print(x)
    if x < k:
        x += 1
    else:
        x -= 1
print(x)
```

```
3
4
5
```

</div>

[20]:

```
x,k = 3,5     # 3 4 5
#x,k = 6,2    # 6 5 4 3 2
#x,k = 4,4    # 4

# write here
```

(continues on next page)

(continued from previous page)

```
3
4
5
```

Exercise - wow

⊛ Given a list of strings `la`, write some code that searches inside the list for the first occurrence of a string beginning with the character `2` (for example `'wow'`. As soon as it is found, the program stops and prints `Found wow`. Otherwise, prints `Not found!` Use a `while` cycle.

Example 1 - given:

```
la = ['a','d','g','wow','f','wonder','r']
```

Prints:

```
examined a
examined d
examined g
Found wow
```

Example 2 - given:

```
la = ['d','v','q','c','e']
```

Prints:

```
examined d
examined v
examined q
examined c
examined e
Not found!
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[21]:
```python
la = ['a','d','g','wow','f','wonder','r']       # a d g Found wow
#la = ['a','d','g','f','wonder','r', 'woman']  # a d g f Found wonder
#la = ['d','v','q','c','e']                    # d v q c e Not found!

# write here

i = 0
while i < len(la) and la[i][0] != 'w':
    print('examined',la[i])
    i += 1

if i < len(la) and la[i][0] == 'w':
    print('Found', la[i])
else:
    print('Not found!')
```

```
examined a
examined d
examined g
Found wow
```

</div>

[21]:
```
la = ['a','d','g','wow','f','wonder','r']       # a d g Found wow
#la = ['a','d','g','f','wonder','r', 'woman']  # a d g f Found wonder
#la = ['d','v','q','c','e']                     # d v q c e Not found!

# write here


```

```
examined a
examined d
examined g
Found wow
```

Exercise - Wild West

⊛⊛ The two outlaws Carson and Butch agreed to bury a treasure in the jolly town of Tombstone, ma now each of them wants to take back the treasure without sharing anything with the partner.

- For arriving to the treasure there is a `road` from `Santa Fe` until `Tombstone` which we represent as a list of strings
- we use two indexes `butch` and `carson` to represent where the outlaws are on the road
- each outlaw starts from a different town
- at each turn Carson moves of **one** city
- at each turn Butch moves of **two** cities, because he has a fast Mustang horse

Write some code which prints the run and terminates as soon as one them arrives to the last city, telling who got the treasure.

- In the case both outlaws arrive to the last city at the same time, prints `Final duel in Tombstone!`
- your code must work for *any* `road` and initial position `carson` and `butch`

Example - 1 given:

```
#            0          1           2            3             4                5
road = ['Santa Fe','Denver','Dodge City', 'Silverton', 'Agua Caliente', 'Tombstone']
carson,butch  = 3, 0
```

it must print:

```
Carson starts from Silverton
Butch starts from Santa Fe
Carson reaches Agua Caliente
Butch reaches Dodge City
Carson reaches Tombstone
Butch reaches Agua Caliente

Carson takes the treasure in  Tombstone !
```

Example 2 - given:

```
#             0          1           2            3              4               5
road = ['Santa Fe','Denver','Dodge City', 'Silverton', 'Agua Caliente', 'Tombstone']
carson,butch  = 3, 2
```

it must print:

```
Carson starts from Silverton
Butch starts from Dodge City
Carson reaches Agua Caliente
Butch reaches Agua Caliente
Carson reaches Tombstone
Butch reaches Tombstone

Final duel in  Tombstone !
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:

```
#            0          1          2              3              4               5
road = ['Santa Fe','Denver','Dodge City', 'Silverton', 'Agua Caliente', 'Tombstone']

carson,butch  = 3, 0    #  Carson takes the treasure in Tombstone !
#carson,butch  = 0, 0   #  Butch takes the treasure in Tombstone !
#carson,butch  = 3, 2   #  Final duel in Tombstone !

# write here

print('Carson starts from', road[carson])
print('Butch starts from', road[butch])

while carson < len(road)-1 and butch < len(road)-1:
    carson = min(len(road)-1, carson + 1)
    butch = min(len(road)-1, butch + 2)
    print('Carson reaches', road[carson])
    print('Butch reaches', road[butch])


print()
if carson == len(road)-1 and butch == len(road)-1:
    print('Final duel in ', road[-1], '!')
elif carson == len(road)-1:
    print('Carson takes the treasure in ', road[-1], '!')
else:
    print('Butch takes the treasure in ', road[-1], '!')
```

```
Carson starts from Silverton
Butch starts from Santa Fe
Carson reaches Agua Caliente
Butch reaches Dodge City
Carson reaches Tombstone
Butch reaches Agua Caliente

Carson takes the treasure in  Tombstone !
```

</div>

[22]:
```python
#            0          1         2              3              4              5
road = ['Santa Fe','Denver','Dodge City', 'Silverton', 'Agua Caliente', 'Tombstone']

carson,butch  = 3, 0    #  Carson takes the treasure in Tombstone !
#carson,butch  = 0, 0   #  Butch takes the treasure in Tombstone !
#carson,butch  = 3, 2   #  Final duel in Tombstone !

# write here
```

```
Carson starts from Silverton
Butch starts from Santa Fe
Carson reaches Agua Caliente
Butch reaches Dodge City
Carson reaches Tombstone
Butch reaches Agua Caliente

Carson takes the treasure in  Tombstone !
```

5.16.7 Modifying sequences

In the tutorial on `for` loops we've seen an important warning we repeat here:

X COMMANDMENT[265]: **You shall never ever add or remove elements from a sequence you are iterating with a** `for` **!**

Falling into such temptations **would produce totally unpredictable behaviours** (do you know the expression *pulling the rug out from under your feet ?*)

If you really need to remove elements from a sequence you are iterating, use a while cycle or duplicate first a copy of the original sequence.

Note the advice is only about `for` **cycles**. In case of necessity, at the end suggests to adopt `while` loops. Let's see when and how ot use them.

Stack - Drawing from a card deck

Suppose having a deck of cards which we represent as a list of strings, and we want to draw all the cards, reading them one by one.

We can write a `while` that as long as the deck contains cards, keeps removing cards from the top with the pop method[266] and prints their name. Remember `pop` MODIFIES the list by removing the last element AND gives back the element as call result, which we can save in a variable we will call `card`:

[23]:
```python
deck = ['3 hearts',    # <---- bottom
        '2 spades',
        '9 hearts',
        '5 diamonds',
        '8 clubs']     # <---- top
```

(continues on next page)

[265] https://en.softpython.org/commandments.html#X-COMMANDMENT
[266] https://en.softpython.org/lists/lists3-sol.html#pop-method

(continued from previous page)

```
while len(deck) > 0:
    card = deck.pop()
    print('Drawn', card)

print('No more cards!')

jupman.pytut()
```

```
Drawn 8 clubs
Drawn 5 diamonds
Drawn 9 hearts
Drawn 2 spades
Drawn 3 hearts
No more cards!
```

[23]:
```
<IPython.core.display.HTML object>
```

Looking at the code, we can notice that:

1. the variable `deck` is initialized
2. we verify that `deck` dimension is greater than zero
3. at each step the list `deck` is MODIFIED by reducing its dimension
4. it returns to step 2

The first three points are the conditions which guarantee the `while` loop will sooner or later actually terminate.

Stack - Drawing until condition

Suppose now to continue drawing cards until we find a heart suit. The situation is more complicated, because now the cycle can terminate in two ways:

1. we find hearts, and interrupt the search
2. there aren't heart cards, and the deck is exhausted

In any case, in the end we must tell the user a result. To do so, it's convenient initializing `card` at the beginning like an empty string for handling the case when no hearts cards are found (or the deck is empty).

Let's try a first implementation which uses an internal `if` to verify whether we have found hearts, and in that case exits with a `break` command.

- Try executing the code by uncomment the second deck which has no hearts cards, and look at the different execution.

[24]:
```
deck = ['3 hearts','2 spades','9 hearts','5 diamonds','8 clubs']
#deck = ['8 spades','2 spades','5 diamonds','4 clubs']    # no hearts!

card = ''
while len(deck) > 0:
    card = deck.pop()
    print('Drawn', card)
    if 'hearts' in card:
        break

if 'hearts' in card:
    print('Found hearts!')
else:
    print("Didn't find hearts!")
```

(continues on next page)

(continued from previous page)

```
jupman.pytut()
```

```
Drawn 8 clubs
Drawn 5 diamonds
Drawn 9 hearts
Found hearts!
```

[24]:
```
<IPython.core.display.HTML object>
```

Exercise - Don't break my heart

⊛ Write some code which solves the same previous problem:

- this time **DO NOT** use `break`
- ensure the code works with a deck without hearts, and also with an empty deck
- **HINT**: put a multiple condition in the `while`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[25]:
```python
deck = ['3 hearts','2 spades','9 hearts','5 diamonds','8 clubs']
#deck = ['8 spades','2 spades','5 diamonds','4 clubs']   # no hearts!
#deck = []  #  no hearts !

card = ''

# write here

while len(deck) > 0 and 'hearts' not in card:
    card = deck.pop()
    print('Drawn', card)

if 'hearts' in card:
    print("Found hearts!")
else:
    print("Didn't find hearts!")
```

```
Drawn 8 clubs
Drawn 5 diamonds
Drawn 9 hearts
Found hearts!
```

</div>

[25]:
```python
deck = ['3 hearts','2 spades','9 hearts','5 diamonds','8 clubs']
#deck = ['8 spades','2 spades','5 diamonds','4 clubs']   # no hearts!
#deck = []  #  no hearts !

card = ''

# write here
```

```
Drawn 8 clubs
Drawn 5 diamonds
Drawn 9 hearts
Found hearts!
```

Questions - what happens?

QUESTION: Look at the following code fragments , and for each try guessing the result it produces (or if it gives an error):

1.
```python
while []:
    print('z')
print('BIG')
```

2.
```python
while ['a']:
    print('z')
print('BUG')
```

3.
```python
la = []
while len(la) < 3:
    la.append('x')
print(la)
```

4.
```python
la = ['x','y','z']
while len(la) > 0:
    print(la.pop())
```

5.
```python
la = ['x','y','z']
while la:
    print(la.pop(0))
```

6.
```python
la = [4,5,8,10]
while la.pop() % 2 == 0:
    print(la)
```

Questions - are they equivalent?

Look at the following code fragments: each contains two parts, A and B. For each value of the variables they depend on, try guessing whether part A will print exactly the same result printed by code in part B

- **FIRST** think about the answer
- **THEN** try executing with each of the values of suggested variables

Are they equivalent? - train

```
print('A:')
la = ['t','r','a','i','n']
while len(la) > 0:
    print(la.pop())

print('\nB:')
la = ['t','r','a','i','n']
la.reverse()
while len(la) > 0:
    print(la.pop(0))
```

Are they equivalent? - append nx

```
print('A:')
x,n,la = 2,0,[]
while x not in la:
    la.append(n)
    n += 1
print(la)

print('\nB:')
x,la = 2,[]
while len(la) < 3:
    la.append(x)
    x += 1
print(la)
```

5.16.8 Exercises - stack

Exercise - break sum

⊛ Look at the following code, and rewrite it in the following cell as `while`

- this time **use command** `break`

[26]:
```
lst = []
i = 0
k = 10
while sum(lst) < k:
    lst.append(i)
    i += 1
    print(lst)
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[27]:
```
lst = []
i = 0

# write here

while True:
    if sum(lst) >= k:
        break
    else:
        lst.append(i)
        i += 1
        print(lst)
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

</div>

[27]:
```
lst = []
i = 0

# write here
```

```
[0]
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
```

Exercise - travelbook

✪✪ Suppose you visited the attic and found a stack of books, which we represent as a list of strings. Each string is prefixed by a label of one character indicating the category (D for Detective story, T for Travel, H for History)

```
stack = ['H-Middle Ages',    # <---- bottom
         'T-Australia',
         'T-Scotland',
         'D-Suspects',
         'T-Caribbean']     # <---- top
```

Since we are passionate about travel books, we want to examine `stack` one book at a time to transfer books into another pile we call `travel`, which at the beginning is empty. We start from the top book in `stack`, and transfer into `travel` only the books starting with the label `T` like (`'T-Australia'`)

```
travel = []
```

Write some code that produces the following print:

```
At the beginning:
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects', 'T-Caribbean
↪']
```

(continues on next page)

(continued from previous page)

```
    travel: []
Taken T-Caribbean
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects']
    travel: ['T-Caribbean']
Discarded D-Suspects
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland']
    travel: ['T-Caribbean']
Taken T-Scotland
    stack:   ['H-Middle Ages', 'T-Australia']
    travel: ['T-Caribbean', 'T-Scotland']
Taken T-Australia
    stack:   ['H-Middle Ages']
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
Discarded H-Middle Ages
    stack:   []
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
```

- The non-travel books are not interesting and must be discarded
- Your code must work with *any* `stack` list

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[28]:

```python

stack = ['H-Middle Ages',    # <---- bottom
         'T-Australia',
         'T-Scotland',
         'D-Suspects',
         'T-Caribbean']     # <---- top


travel = []

# write here
print("At the beginning:")
print('    stack:  ', stack)
print('    travel:', travel)

while len(stack) > 0:
    book = stack.pop()
    if book.startswith('T'):
        print('Taken',book)
        travel.append(book)
    else:
        print('Discarded', book)
    print('    stack:  ', stack)
    print('    travel:', travel)
```

```
At the beginning:
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects', 'T-Caribbean
↪']
    travel: []
Taken T-Caribbean
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects']
    travel: ['T-Caribbean']
Discarded D-Suspects
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland']
```

(continues on next page)

(continued from previous page)

```
    travel: ['T-Caribbean']
Taken T-Scotland
    stack:   ['H-Middle Ages', 'T-Australia']
    travel: ['T-Caribbean', 'T-Scotland']
Taken T-Australia
    stack:   ['H-Middle Ages']
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
Discarded H-Middle Ages
    stack:   []
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
```

</div>

[28]:

```
stack = ['H-Middle Ages',    # <---- bottom
         'T-Australia',
         'T-Scotland',
         'D-Suspects',
         'T-Caribbean']     # <---- top


travel = []

# write here
```

```
At the beginning:
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects', 'T-Caribbean
↪']
    travel: []
Taken T-Caribbean
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland', 'D-Suspects']
    travel: ['T-Caribbean']
Discarded D-Suspects
    stack:   ['H-Middle Ages', 'T-Australia', 'T-Scotland']
    travel: ['T-Caribbean']
Taken T-Scotland
    stack:   ['H-Middle Ages', 'T-Australia']
    travel: ['T-Caribbean', 'T-Scotland']
Taken T-Australia
    stack:   ['H-Middle Ages']
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
Discarded H-Middle Ages
    stack:   []
    travel: ['T-Caribbean', 'T-Scotland', 'T-Australia']
```

Exercise - BANG !

⊛⊛ There are two stacks of objects `right_stack` and `left_stack` which we represent as lists of strings. As a pastime, a cowboy decides to shoot the objects at the top of the stacks, alternating the stack at each shoot. The cowboy is skilled and always hits the target, so each shot decreases a stack.

- Suppose the objects on top are the ones at the end of the list
- To keep track of which stack to hit, use a variable `shoot` holding either `'R'` or `'L'` character
- After each shot the cowboy if possible changes the stack , otherwise keeps shooting at the same stack until it's empty.
- your code must work for *any* stack and initial shot

Example - given:

```
left_stack = ['box','boot','horseshoe','bucket']
right_stack = ['bin','saddle','tin can']
shoot = 'R'
```

after your code, it must print:

```
Ready?
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle', 'tin can']
BANG! right:  tin can
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle']
BANG! left:   bucket
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin', 'saddle']
BANG! right:  saddle
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin']
BANG! left:   horseshoe
   left_stack: ['box', 'boot']
  right_stack: ['bin']
BANG! right:  bin
   left_stack: ['box', 'boot']
  right_stack: []
BANG! left:   boot
   left_stack: ['box']
  right_stack: []
Nothing to shoot on the right!
   left_stack: ['box']
  right_stack: []
BANG! left:   box
   left_stack: []
  right_stack: []
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[29]:
```
left_stack = ['box','boot','horseshoe','bucket']
right_stack = ['bin','saddle','tin can']
shoot = 'R'
#shoot = 'L'
#left_stack = ['bucket', 'box']
```

(continues on next page)

(continued from previous page)

```
# write here

print('Ready?')
print('   left_stack:', left_stack)
print('  right_stack:', right_stack)
while len(right_stack) > 0 or len(left_stack) > 0:
    if shoot == 'R':
        if len(right_stack) > 0:
            print('BANG! right: ', right_stack.pop())
        else:
            print('Nothing to shoot on the right!')
        shoot = 'L'
    else:
        if len(left_stack) > 0:
            print('BANG! left:  ', left_stack.pop())
        else:
            print('Nothing to shoot on the left!')
        shoot = 'R'

    print('   left_stack:', left_stack)
    print('  right_stack:', right_stack)
```

```
Ready?
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle', 'tin can']
BANG! right:  tin can
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle']
BANG! left:   bucket
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin', 'saddle']
BANG! right:  saddle
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin']
BANG! left:   horseshoe
   left_stack: ['box', 'boot']
  right_stack: ['bin']
BANG! right:  bin
   left_stack: ['box', 'boot']
  right_stack: []
BANG! left:   boot
   left_stack: ['box']
  right_stack: []
Nothing to shoot on the right!
   left_stack: ['box']
  right_stack: []
BANG! left:   box
   left_stack: []
  right_stack: []
```

</div>

[29]:
```
left_stack = ['box','boot','horseshoe','bucket']
right_stack = ['bin','saddle','tin can']
shoot = 'R'
#shoot = 'L'
```

(continues on next page)

(continued from previous page)

```
#left_stack = ['bucket', 'box']

# write here
```

```
Ready?
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle', 'tin can']
BANG! right:  tin can
   left_stack: ['box', 'boot', 'horseshoe', 'bucket']
  right_stack: ['bin', 'saddle']
BANG! left:   bucket
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin', 'saddle']
BANG! right:  saddle
   left_stack: ['box', 'boot', 'horseshoe']
  right_stack: ['bin']
BANG! left:   horseshoe
   left_stack: ['box', 'boot']
  right_stack: ['bin']
BANG! right:  bin
   left_stack: ['box', 'boot']
  right_stack: []
BANG! left:   boot
   left_stack: ['box']
  right_stack: []
Nothing to shoot on the right!
   left_stack: ['box']
  right_stack: []
BANG! left:   box
   left_stack: []
  right_stack: []
```

Exercise - Growing or degrowing?

⊛⊛ Write some code which given a list `la`, keeps MODIFYING the list according to this procedure:

- if the last element is odd (i.e. `7`), attaches a new number at the end of the list obtained by multiplying by two the last element (i.e. attaches `14`)
- if the last element is even, removes the last two elements
- **DO NOT** create a new list (so no rows starting with `la =`)
- **WARNING**: when we want both grow and degrow the sequence we are considering in a cycle, we must convince ourselves that sooner or later the termination condition will happen, it's easy to make mistakes and end up with an infinite cycle!
- **HINT**: to degrow the list, you can use the pop method[267]

Example - given:

```
la = [3,5,6,7]
```

Executing the code, it must print:

[267] https://en.softpython.org/lists/lists3-sol.html#pop-method

```
 Odd: attaching 14
      la becomes [3, 5, 6, 7, 14]
Even: removing 14
      removing 7
      la becomes [3, 5, 6]
Even: removing 6
      removing 5
      la becomes [3]
 Odd: attaching 6
      la becomes [3, 6]
Even: removing 6
      removing 3
      la becomes []
Done! la is []
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[30]:
```python
la = [3,5,6,7]

# write here

i = 0
while len(la) > 0:
    if la[-1] % 2 == 1:
        new = la[-1]*2
        la.append(new)
        print(' Odd: attaching', new)
    else:
        print('Even: removing',la.pop())
        print('      removing',la.pop())
    print('      la becomes', la)
    i += 1
print('Done! la is', la)
```

```
 Odd: attaching 14
      la becomes [3, 5, 6, 7, 14]
Even: removing 14
      removing 7
      la becomes [3, 5, 6]
Even: removing 6
      removing 5
      la becomes [3]
 Odd: attaching 6
      la becomes [3, 6]
Even: removing 6
      removing 3
      la becomes []
Done! la is []
```

</div>

[30]:
```python
la = [3,5,6,7]

# write here
```

```
 Odd: attaching 14
      la becomes [3, 5, 6, 7, 14]
Even: removing 14
      removing 7
      la becomes [3, 5, 6]
Even: removing 6
      removing 5
      la becomes [3]
 Odd: attaching 6
      la becomes [3, 6]
Even: removing 6
      removing 3
      la becomes []
Done! la is []
```

5.16.9 References

- Think Python, Chapter 7, Iteration[268]
- W3 Resources - Conditional statements and loop exercises[269]

[ ]:

```

```

5.17 Sequences and comprehensions

5.17.1 Download exercises zip

Browse online files[270]

We can write elegant and compact code with sequences. First we will see how to scan sequences with iterators, and then how to build them with comprehensions of lists.

5.17.2 What to do

1. Unzip exercises zip in a folder, you should obtain something like this:

```
sequences
    sequences.ipynb
    sequences-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

2. open Jupyter Notebook from that folder. Two things should open, first a console and then a browser. The browser should show a file list: navigate the list and open the notebook `sequences.ipynb`

[268] http://greenteapress.com/thinkpython2/html/thinkpython2008.html
[269] https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php
[270] https://github.com/DavidLeoni/softpython-en/tree/master/sequences

3. Go on reading the exercises file, sometimes you will find paragraphs marked **Exercises** graded from ✪ to ✪✪✪✪ which will ask to write Python commands in the following cells.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.17.3 Iterables - lists

When dealing with loops with often talked about *iterating* sequences, but what does it exactly mean for a sequence to be *iterable* ? Concretely, it means we can call the function`iter` on that sequence.

Let's try for example with familiar lists:

[2]:
```
iter(['a','b','c','d'])
```

[2]:
```
<list_iterator at 0x7fc564437b00>
```

We notice Python just created an object of type `list_iterator`.

NOTE: the list was not shown!

You can imagine an iterator as a sort of still machine, that each time is activated it produces an element from the sequence, one at a time

---

Typically, an iterator only knows its *position* inside the sequence, and can provide us with the sequence elements one by one if we keep asking with calls to the function `next`:

[3]:
```
iterator = iter(['a','b','c','d'])
```

[4]:
```
next(iterator)
```

[4]:
```
'a'
```

[5]:
```
next(iterator)
```

[5]:
```
'b'
```

[6]:
```
next(iterator)
```

[6]:
```
'c'
```

[7]:
```
next(iterator)
```

[7]:
```
'd'
```

Note how the iterator has a *state* to keep track of where it is in the sequence (in other words, it's *stateful)*. The state is changed at each call of function `next`.

If we try asking more elements of the available ones, Python raises the exception `StopIteration`:

```
next(iterator)

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-65-4518bd5da67f> in <module>()
----> 1 next(iterator)

StopIteration:
```

V COMMANDMENT[271] **You shall never ever redefine `next` and `iter` system functions.**

DO NOT use them as variables !!

5.17.4 iterables - range

We iterated a list, which is a completely materialized in memory sequence we scanned with the iterator object. There are also other peculiar sequences which are *not* materialized in memory, like for example `range`.

Previously we used `range` in for loops[272] to obtain a sequence of numbers, but exactly, what is `range` doing? Let's try calling it on its own:

```
[8]: range(4)
```

```
[8]: range(0, 4)
```

Maybe we expected a sequence of numbers, instead, Python is showing us an object of type `range` (with the lower range limit).

NOTE: No number sequence is currently present in memory

We only have a 'still' *iterable* object, which if we want can provide us with numbers

How can we ask for numbers?

We've seen we can use a `for` loop:

```
[9]: for x in range(4):
         print(x)
```

```
0
1
2
3
```

As an alternative, we can pass `range` to the function `iter` which produces an *iterator*.

WARNING: `range` is iterable but it is NOT an iterator !!

To obtain the iterator we must call the `iter` function on the `range` object

271 https://en.softpython.org/commandments.html#V-COMMANDMENT
272 https://en.softpython.org/control-flow/flow2-for-sol.html#Counting-with-range

[10]:
```
iterator = iter(range(4))
```

`iter` also produces a 'still' object, which hasn't materialized numbers in memory yet:

[11]:
```
iterator
```

[11]:
```
<range_iterator at 0x7fc56439eea0>
```

In order to ask we must use the function `next`:

[12]:
```
next(iterator)
```

[12]:
```
0
```

[13]:
```
next(iterator)
```

[13]:
```
1
```

[14]:
```
next(iterator)
```

[14]:
```
2
```

[15]:
```
next(iterator)
```

[15]:
```
3
```

Note the iterator has a *state,* which is changed at each `next` call to keep track of where it is in the sequence.

If we try asking for more elements than actually available, Python raises a `StopIteration` exception:

```
next(iterator)

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-65-4518bd5da67f> in <module>()
----> 1 next(iterator)

StopIteration:
```

5.17.5 Materializing a sequence

We said a `range` object does not physically materialize in memory all the numbers at the same time. We can get them one by one by only using the iterator. What if we wanted a list with all the numbers? In the tutorial on lists[273] we've seen that by passing a sequence to function `list`, a new list is created with all the sequence elements. We talked generically about a *sequence,* but the more correct term would have been *iterable.*

If we pass any *iterable* object to `list`, then a new list will be built - we've seen `range` is iterable so let's try:

[16]:
```
list(range(4))
```

[16]:
```
[0, 1, 2, 3]
```

Voilà ! Now the sequence is all physically present in memory.

[273] https://en.softpython.org/lists/lists1-sol.html#Convert-sequences-into-lists

WARNING: `list` consumes the iterator!

If you try calling twice `list` on the same iterator, you will get an empty list:

[17]:
```python

sequence = range(4)
iterator = iter(sequence)
```

[18]:
```python
new1 = list(iterator)
```

[19]:
```python
new1
```

[19]:
```
[0, 1, 2, 3]
```

[20]:
```python
new2 = list(iterator)
```

[21]:
```python
new2
```

[21]:
```
[]
```

What if we wanted to directly access a specific position in the sequence generated by the iterator? Let's try extracting the character at index 2:

[22]:
```python
sequence = range(4)
iterator = iter(sequence)
```

```python
iterator[2]
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-129-3c080cc9e700> in <module>()
      1 sequence = range(4)
      2 iterator = iter(sequence)
----> 3 iterator[3]

TypeError: 'range_iterator' object is not subscriptable
```

… sadly we get an error!

We are left with only two alternatives. Either:

a) First we convert to list and then use the squared brackets

b) We call `next` 4 times (remember indexes start from zero)

Option a) very often looks handy, but careful: **converting an iterator into a list creates a NEW list in memory**. If the list is very big and/or this operation is repeated many times, you risk occupying memory for nothing.

Let's see the example in Python Tutor again:

[23]:
```python
# WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#          (it's sufficient to execute it only once)

import jupman
```

[24]:
```python
sequence = range(4)
iterator = iter(sequence)
```

(continues on next page)

(continued from previous page)

```
new1 = list(iterator)
new2 = list(iterator)

jupman.pytut()
```

```
[24]: <IPython.core.display.HTML object>
```

QUESTION: Which object occupies more memory? `a` or `b`?

```
a = range(10)
b = range(10000000)
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: they both occupy the same amount of memory.

</div>

QUESTION: Which object occupies more memory? `a` or `b` ?

```
a = list(range(10))
b = list(range(10000000))
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `b` occupies more (the list is materialized)

</div>

Questions - `range`

Look at the following expressions, and for each try guessing the result (or if it gives an error):

1. `range(3)`
2. `range()`
3. `list(range(-3))`
4. `range(3,6)`
5. `list(range(5,4))`
6. `list(range(3,3))`
7. `range(3) + range(6)`
8. `list(range(3)) + list(range(6))`
9. `list(range(0,6,2))`

10. `list(range(9,6,-1))`

5.17.6 reversed

`reversed` is a *function* which takes a sequence as parameter and PRODUCES a NEW *iterator* which allows to run through the sequence in reverse order.

> **WARNING**: by calling `reversed` we directly obtain an *iterator* !
>
> So you do *not* need to make further calls to `iter` as done with `range`!

Let's have a better look with an example:

```
[25]: la = ['s','c','a','n']
```

```
[26]: reversed(la)
```

```
[26]: <list_reverseiterator at 0x7fc564263940>
```

We see `reversed` has produced an *iterator* as result (not a reversed list)

INFO: iterators occupy a small amount of memory

Creating an iterator from a sequence only creates a sort of pointer, it *does not* create new memory regions.

Furthermore , we see the original list associated to `la` was *not* changed:

```
[27]: print(la)
```

```
['s', 'c', 'a', 'n']
```

> **WARNING**: the function `reversed` is different from reverse method[274]
>
> Note the final **d**! If we tried to call it as a method we would get an error:

```
>>> la.reversed()

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-182-c8d1eec57fdd> in <module>
----> 1 la.reversed()

AttributeError: 'list' object has no attribute 'reversed'
```

[274] https://en.softpython.org/lists/lists3-sol.html#reverse-method

Iterating with `next`

How can we obtain a reversed list in memory? In other words, how can we actionate the iterator machine?

We can ask the iterator for one element at a time with the function `next`:

[28]:
```
la = ['a','b','c']
```

[29]:
```
iterator = reversed(la)
```

[30]:
```
next(iterator)
```

[30]:
```
'c'
```

[31]:
```
next(iterator)
```

[31]:
```
'b'
```

[32]:
```
next(iterator)
```

[32]:
```
'a'
```

Once the iterator is exhausted, by calling `next` again we will get an error:

```
next(iterator)

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-248-4518bd5da67f> in <module>
----> 1 next(iterator)

StopIteration:
```

Let's try manually creating a destination list `lb` and adding elements we obtain one by one:

[33]:
```
la = ['a','b','c']
iterator = reversed(la)
lb = []
lb.append(next(iterator))
lb.append(next(iterator))
lb.append(next(iterator))
print(lb)

jupman.pytut()
```

```
['c', 'b', 'a']
```

[33]:
```
<IPython.core.display.HTML object>
```

Exercise - sconcerto

Write some code which given a list of characters `la`, puts in a list `lb` all the characters at odd position taken from reversed list `la`.

- use `reversed` and `next`
- **DO NOT** modify `la`
- **DO NOT** use negative indexes
- **DO NOT** use `list`

Example - given:

```
#       8     7     6     5     4     3     2     1     0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []
```

After your code it must show:

```
>>> print(lb)
['t', 'e', 'n', 'c']
>>> print(la)
['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
```

We invite you to solve the problem in several ways:

WAY 1 - without cycle: Suppose the list length **is fixed**, and repeatedly call `next` *without using a loop*

WAY 2 - while: Suppose having a list of arbitrary length, and try generalizing previous code by *using a* `while` *cycle,* and calling `next` inside

- **HINT 1**: keep track of the position in which you are with a counter `i`
- **HINT 2**: you cannot call `len` on an iterator, so in the `while` conditions you will have to use the original list length

WAY 3 - for: this is the most elegant way. Suppose having a list of arbitrary length and *use a loop* like `for x in reversed(la)`

- **HINT**: you will still need to keep track of the position in which you are with an `i` counter

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[34]:
```
# WAY 1: MANUAL

#       8     7     6     5     4     3     2     1     0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here

iterator = reversed(la)

next(iterator)
lb.append(next(iterator))
next(iterator)
lb.append(next(iterator))
next(iterator)
lb.append(next(iterator))
next(iterator)
```

(continues on next page)

(continued from previous page)

```
lb.append(next(iterator))
print(lb)

#jupman.pytut()
```

```
['t', 'e', 'n', 'c']
```

</div>

[34]:
```
# WAY 1: MANUAL

#       8    7    6    5    4    3    2    1    0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here
```

```
['t', 'e', 'n', 'c']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[35]:
```
# WAY 2: WHILE

#       8    7    6    5    4    3    2    1    0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here

iterator = reversed(la)

i = 1
while i < len(la):
    if i % 2 == 1:
        next(iterator)
        lb.append(next(iterator))
    i += 2

print(lb)
```

```
['t', 'e', 'n', 'c']
```

</div>

[35]:
```
# WAY 2: WHILE

#       8    7    6    5    4    3    2    1    0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here
```

```
['t', 'e', 'n', 'c']
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[36]: # WAY 3: for

#      8    7    6    5    4    3    2    1    0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here
i = 0

for x in reversed(la):
    if i % 2 == 1:
        lb.append(x)
    i += 1
print(lb)
```

```
['t', 'e', 'n', 'c']
```

</div>

```
[36]: # WAY 3: for

#      8    7    6    5    4    3    2    1    0
la = ['s', 'c', 'o', 'n', 'c', 'e', 'r', 't', 'o']
lb = []

# write here
```

```
['t', 'e', 'n', 'c']
```

Materializing an iterator

Luckily enough, we can obtain a list from an iterator with a less laborious method.

We've seen that when we want to create a new list from a sequence, we can use `list` as if it were a function. We can also do it in this case, interpreting the iterator as if it were a sequence:

```
[37]: la = ['s', 'c', 'a', 'n']
list( reversed(la) )
```

```
[37]: ['n', 'a', 'c', 's']
```

Notice we generated a NEW list, the original one associated to `la` is always the same:

```
[38]: la
```

```
[38]: ['s', 'c', 'a', 'n']
```

Let's see what happens using Python Tutor (we created some extra variables to evidence relevant passages):

[39]:
```
la = ['s', 'c', 'a', 'n']
iterator = reversed(la)
new = list(iterator)
print("la is",la)
print("new is",new)

jupman.pytut()
```

```
la is ['s', 'c', 'a', 'n']
new is ['n', 'a', 'c', 's']
```

[39]:
```
<IPython.core.display.HTML object>
```

QUESTION Which effect is the following code producing?

```
la = ['b','r','i','d','g','e']
lb = list(reversed(reversed(la)))
```

5.17.7 sorted

The **function** `sorted` takes as parameter a sequence and returns a NEW sorted list.

> **WARNING**: `sorted` returns a LIST, not an iterator!

[40]:
```
sorted(['g','a','e','d','b'])
```

[40]:
```
['a', 'b', 'd', 'e', 'g']
```

> **WARNING**: `sorted` is a **function** different from sort metod[275] !
>
> Note the final **ed**! If we tried to call it with a different method we would get an error:

```
>>> la.sorted()

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-182-c8d1eec57fdd> in <module>
----> 1 la.reversed()

AttributeError: 'list' object has no attribute 'sorted'
```

[275] https://en.softpython.org/lists/lists3-sol.html#sort-method

Exercise - reversort

⊕ Given a list of names, write some code to produce a list sorted in reverse

There are at least a couple of ways to do it in a single line of code, find them both

- INPUT: `['Maria','Paolo','Giovanni','Alessia','Greta']`
- OUTPUT: `['Paolo', 'Maria', 'Greta', 'Giovanni', 'Alessia']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[41]:
```python
# write here

list(sorted(['Maria','Paolo','Giovanni','Alessia','Greta'], reverse=True))

# or
#list(reversed(sorted(['Maria','Paolo','Giovanni','Alessia','Greta'])))
```

[41]:
```
['Paolo', 'Maria', 'Greta', 'Giovanni', 'Alessia']
```

</div>

[41]:
```python
# write here
```

[41]:
```
['Paolo', 'Maria', 'Greta', 'Giovanni', 'Alessia']
```

5.17.8 zip

Suppose we have two lists `paintings` and `years`, with rispectively names of famous paintings and the dates in which they were painted:

[42]:
```python
paintings = ["The Mona Lisa", "The Birth of Venus", "Sunflowers"]
years = [1503, 1482, 1888]
```

We want to produce a new list which contains some tuples which associate each painting with the year it was made:

```
[('The Mona Lisa', 1503),
 ('The Birth of Venus', 1482),
 ('Sunflowers', 1888)]
```

There are various ways to do it but certainly the most elegant is by using the **function** `zip` which produces an **iterator**:

[43]:
```python
zip(paintings, years)
```

[43]:
```
<zip at 0x7fc564252948>
```

Even if you don't see written 'iterator' in the object name, we can still use it as such with `next`:

[44]:
```python
iterator = zip(paintings, years)
next(iterator)
```

[44]:
```
('The Mona Lisa', 1503)
```

[45]:
```python
next(iterator)
```

[45]:
```
('The Birth of Venus', 1482)
```

[46]:
```
next(iterator)
```

[46]:
```
('Sunflowers', 1888)
```

As done previously, we can convert everything to a list with `list`:

[47]:
```
paintings = ["The Mona Lisa", "The Birth of Venus", "Sunflowers"]
years = [1503, 1482, 1888]

list(zip(paintings,years))
```

[47]:
```
[('The Mona Lisa', 1503), ('The Birth of Venus', 1482), ('Sunflowers', 1888)]
```

If the lists have different length, the sequence produced by `zip` will be as long as the shortest input sequence:

[48]:
```
list(zip([1,2,3], ['a','b','c','d','e']))
```

[48]:
```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

If we will, we can pass an arbitrary number of sequences - for example, by passing three of them we will obtain triplets of values:

[49]:
```
songs = ['Imagine', 'Hey Jude', 'Satisfaction', 'Yesterday' ]
authors = ['John Lennon','The Beatles', 'The Rolling Stones', 'The Beatles']
years = [1971, 1968, 1965, 1965]
list(zip(songs, authors, years))
```

[49]:
```
[('Imagine', 'John Lennon', 1971),
 ('Hey Jude', 'The Beatles', 1968),
 ('Satisfaction', 'The Rolling Stones', 1965),
 ('Yesterday', 'The Beatles', 1965)]
```

Exercise - ladder

Given a number `n`, create a list of tuples that for each integer number x such that $0 \leq x \leq n$ associates the number $n - x$

- INPUT: `n=5`
- OUTPUT: `[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```
n = 5
# write here
list(zip(range(n), reversed(range(n))))
```

[50]:
```
[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]
```

</div>

[50]:
```
n = 5
# write here
```

[50]:
```
[(0, 4), (1, 3), (2, 2), (3, 1), (4, 0)]
```

5.17.9 List comprehensions

List comprehensions are handy when you need to generate a NEW list by executing the same operation on all the elements of a sequence. Comprehensions start and end with square brackets `[ ]` so theit syntax reminds lists, but inside they contain a special `for` to loop inside a sequence:

[51]:
```
numbers = [2,5,3,4]

doubled = [x*2 for x in numbers]

doubled
```

[51]:
```
[4, 10, 6, 8]
```

Note the variable `numbers` is still associated to the original list:

[52]:
```
numbers
```

[52]:
```
[2, 5, 3, 4]
```

What happened ? We wrote the name of a variable `x` we just invented, and we told Python to go through the list `numbers`: at each iteration, the variable `x` is associated to a different value of the list `numbers`. This value can be reused in the expression we wrote on left of the `for`, which in this case is `x*2`

As name for the variable we used `x`, but we could have used any other name, for example this code is equivalent to the previous one:

[53]:
```
numbers = [2,5,3,4]

doubled = [number * 2 for number in numbers]

doubled
```

[53]:
```
[4, 10, 6, 8]
```

On the left of the `for` we can write any expression which produces a value, for example here we write `x + 1` to increment all the numbers of the original list:

[54]:
```
numbers = [2,5,3,4]

augmented = [x + 1 for x in numbers]

augmented
```

[54]:
```
[3, 6, 4, 5]
```

QUESTION: What is this code going to produce? If we visualize it in Python Tutor, will `la` and `lb` point to different objects?

```
la = [7,5,6,9]
lb = [x for x in la]
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: When `[x for x in la]` is executed, during the first iteration `x` is valued `7`, during the second `5`, during the third one `6` and so on and so forth. In the expression on the left of the `for` we put only `x`, so as expression result we will get the same identical number taken from the original string.

The code will produce a NEW list `[7,5,6,9]` and it will be associated to the variable `lb`.

</div>

[55]:
```
la = [7,5,6,9]
lb = [x for x in la]

jupman.pytut()
```

[55]:
```
<IPython.core.display.HTML object>
```

List comprehensions on strings

QUESTION: What is this code going to produce?

```
[x for x in 'question']
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: It will produce `['q', 'u', 'e', 's', 't', 'i', 'o', 'n']`

Since `question` is a string, if we interpret it as a sequence each element of it is a character, so during the first iteration `x` is valued `'q'`, during the second `'u'`, during the third `'e'` and so on and so forth. In the expression on the left of the `for` we put only `x`, so as expression result we will obtain the same identical character taken from the original string.

</div>

Let's now suppose to have a list of `animals` and we want to produce another one with the same names as uppercase. We can do it in a compact way with a list comprehension like this:

[56]:
```
animals = ['dogs', 'cats', 'squirrels', 'elks']

new_list = [animal.upper() for animal in animals]
```

[57]:
```
new_list
```

[57]:
```
['DOGS', 'CATS', 'SQUIRRELS', 'ELKS']
```

In the left part reserved to the expression we used the method `.upper()` on the string variable `animal`. We know strings are immutable, so we're sure the method call produces a NEW string. Let's see what happened with Python Tutor:

[58]:
```

animals = ['dogs', 'cats', 'squirrels', 'elks']

new_list = [animal.upper() for animal in animals]

jupman.pytut()
```

[58]:
```
<IPython.core.display.HTML object>
```

⊛ **EXERCISE**: Try writing here a list comprehension to put all characters as lowercase (`.lower()` method)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[59]: animals = ['doGS', 'caTS', 'SQUIrreLs', 'ELks']

      # write here

      [animal.lower() for animal in animals]
```

```
[59]: ['dogs', 'cats', 'squirrels', 'elks']
```

</div>

```
[59]: animals = ['doGS', 'caTS', 'SQUIrreLs', 'ELks']

      # write here
```

```
[59]: ['dogs', 'cats', 'squirrels', 'elks']
```

5.17.10 Questions - List comprehensions

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1. ```
   [x for [4,2,5]]
   ```

2. ```
   x for x in range(3)
   ```

3. ```
   [x for y in 'cartoccio']
   ```

4. ```
   [for x in 'zappa']
   ```

5. ```
   [for [3,4,5]]
   ```

6. ```
   [k + 1 for k in 'bozza']
   ```

7. ```
   [k + 1 for k in range(5)]
   ```

8. ```
   [k > 3 for k in range(7)]
   ```

9. ```
   [s + s for s in ['lam','pa','da']]
   ```

10. ```
    la = ['x','z','z']
    [x for x in la] + [y for y in la]
    ```

11. ```
    [x.split('-') for x in ['a-b', 'c-d', 'e-f']]
    ```

12. ```
    ['@'.join(x) for x in [['a','b.com'],['c','d.org'],['e','f.net'] ]]
    ```

13. ```
    ['z' for y in 'borgo'].count('z') == len('borgo')
    ```

14. ```
    m = [['a','b'],['c','d'],['e','f'] ]
    la = [x.pop() for x in m]   # not advisable - why ?
    ```

(continues on next page)

(continued from previous page)

```
print(' m:', m)
print('la:',la)
```

5.17.11 Exercises - list comprehension

Exercise - power

⊕ Given a list of numbers, produce a list with the input numbers squared

- INPUT: `[4,5,9]`
- OUTPUT: `[16, 25, 81]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[60]: import math

      # write here
      [x*x for x in [4,5,9]]
```

```
[60]: [16, 25, 81]
```

</div>

```
[60]: import math

      # write here
```

```
[60]: [16, 25, 81]
```

Exercise - root

⊕ Given a list of numbers, produce a list with the square root of the input numbers

- INPUT: `[16,25,81]`
- OUTPUT: `[4.0, 5.0, 9.0]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[61]: import math

      # write here
      [math.sqrt(x) for x in [16,25,81]]
```

```
[61]: [4.0, 5.0, 9.0]
```

</div>

```
[61]: import math

      # write here
```

(continues on next page)

(continued from previous page)

```
[61]: [4.0, 5.0, 9.0]
```

Exercise - first chars

⊕ Given a list of strings, produce a list with the first characters of each string

- INPUT: `['When','The','Telephone','Rings']`
- OUTPUT: `['W', 'T', 'T', 'R']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[62]: # write here
[x[0] for x in ['When','The','Telephone','Rings']]
```

```
[62]: ['W', 'T', 'T', 'R']
```

</div>

```
[62]: # write here
```

```
[62]: ['W', 'T', 'T', 'R']
```

Exercise - don't worry

⊕ Given a list of strings, produce a list with the lengths of all the lists

- INPUT: `["don't", 'worry','and', 'be','happy']`
- OUTPUT: `[5, 5, 3, 2, 5]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[63]: # write here
[len(x) for x in ["don't", 'worry','and', 'be','happy']]
```

```
[63]: [5, 5, 3, 2, 5]
```

</div>

```
[63]: # write here
```

```
[63]: [5, 5, 3, 2, 5]
```

Exercise - greater than 3

⊕ Given a list of numbers, produce a list with `True` if the corresponding element is greater than `3`, `False` otherwise

- INPUT: `[4,1,0,5,0,9,1]`
- OUTPUT: `[True, False, False, True, False, True, False]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[64]:
```python
# write here
[x > 3 for x in [4,1,0,5,0,9,1]]
```

[64]:
```
[True, False, False, True, False, True, False]
```

</div>

[64]:
```python
# write here
```

[64]:
```
[True, False, False, True, False, True, False]
```

Exercise - even

⊕ Given a list of numbers, produce a list with `True` if the corresponding element is even

- INPUT: `[3,2,4,1,5,3,2,9]`
- OUTPUT: `[False, True, True, False, False, False, True, False]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[65]:
```python
# write here
[x % 2 == 0 for x in [3,2,4,1,5,3,2,9]]
```

[65]:
```
[False, True, True, False, False, False, True, False]
```

</div>

[65]:
```python
# write here
```

[65]:
```
[False, True, True, False, False, False, True, False]
```

Exercise - both ends

⊕ Given a list of strings having at least two characters each, produce a list of strings with the first and last characters of each

- INPUT: `['departing', 'for', 'the', 'battlefront']`
- OUTPUT: `['dg', 'fr', 'te', 'bt']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[66]:
```
# write here
[x[0] + x[-1] for x in ['departing', 'for', 'the', 'battlefront']]
```

[66]:
```
['dg', 'fr', 'te', 'bt']
```

</div>

[66]:
```
# write here
```

[66]:
```
['dg', 'fr', 'te', 'bt']
```

Exercise - dashes

⊛ Given a list of lists of characters, produce a list of strings with characters separated by dashes

- INPUT: `[['a','b'],['c','d','e'], ['f','g']]`
- OUTPUT: `['a-b', 'c-d-e', 'f-g']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[67]:
```
# write here
['-'.join(x) for x in [['a','b'],['c','d','e'], ['f','g']]]
```

[67]:
```
['a-b', 'c-d-e', 'f-g']
```

</div>

[67]:
```
# write here
```

[67]:
```
['a-b', 'c-d-e', 'f-g']
```

Exercise - lollosa

⊛ Given a string `s`, produce a list of tuples having for each character the number of occurrences of that character in the string

- INPUT: `s = 'lollosa'`
- OUTPUT: `[('l', 3), ('o', 2), ('l', 3), ('l', 3), ('o', 2), ('s', 1), ('a', 1)]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[68]:
```
s = 'lollosa'
# write here
[(car, s.count(car)) for car in s]
```

[68]:
```
[('l', 3), ('o', 2), ('l', 3), ('l', 3), ('o', 2), ('s', 1), ('a', 1)]
```

</div>

```
[68]: s = 'lollosa'
      # write here
```

```
[68]: [('l', 3), ('o', 2), ('l', 3), ('l', 3), ('o', 2), ('s', 1), ('a', 1)]
```

Exercise - dog cat

⊕ Given a list of strings of at least two characters each, produce a list with the strings without intial and final characters

- INPUT: `['donkey','eagle','ox', 'dog' ]`
- OUTPUT: `['onke', 'agl', '', 'o']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[69]: # write here
      [x[1:-1] for x in ['donkey','eagle','ox', 'dog' ]]
```

```
[69]: ['onke', 'agl', '', 'o']
```

</div>

```
[69]: # write here
```

```
[69]: ['onke', 'agl', '', 'o']
```

Exercise - smurfs

⊕ Given some names produce a list with the names sorted alphabetically and all in uppercase

- INPUT: `['Brainy', 'Hefty', 'Smurfette', 'Clumsy']`
- OUTPUT: `['BRAINY', 'CLUMSY', 'HEFTY', 'SMURFETTE']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[70]: # write here
      [x.upper() for x in sorted(['Brainy', 'Hefty', 'Smurfette', 'Clumsy'])]
```

```
[70]: ['BRAINY', 'CLUMSY', 'HEFTY', 'SMURFETTE']
```

</div>

```
[70]: # write here
```

```
[70]: ['BRAINY', 'CLUMSY', 'HEFTY', 'SMURFETTE']
```

Exercise - precious metals

⊕ Given two lists `values` and `metals` produce a list containing all the couples value-metal as tuples

INPUT:

```
values = [10,25,50]
metals = ['silver','gold','platinum']
```

OUTPUT: `[(10, 'silver'), (25, 'gold'), (50, 'platinum')]`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[71]:
```
values = [10,25,50]
metals = ['silver','gold','platinum']


# write here
list(zip(values, metals))
```

[71]:
```
[(10, 'silver'), (25, 'gold'), (50, 'platinum')]
```

</div>

[71]:
```
values = [10,25,50]
metals = ['silver','gold','platinum']


# write here
```

[71]:
```
[(10, 'silver'), (25, 'gold'), (50, 'platinum')]
```

5.17.12 Filtered list comprehensions

During the construction of a list comprehension we can filter the elements taken from the sequence by using an `if`. For example, the following expression takes from the sequence only numbers greater than `5`:

[72]:
```
[x for x in [7,4,8,2,9] if x > 5]
```

[72]:
```
[7, 8, 9]
```

After the `if` we can put any expression which reuses the variable on which we are iterating, for example if we are iterating a string we can keep only the uppercase characters:

[73]:
```
[x for x in 'The World Goes Round' if x.isupper()]
```

[73]:
```
['T', 'W', 'G', 'R']
```

WARNING: `else` **is not supported**

For example, writing this generates an error:

```
[x for x in [7,4,8,2,9] if x > 5 else x + 1]    # WRONG!

File "<ipython-input-74-9ba5c135c58c>", line 1
    [x for x in [7,4,8,2,9] if x > 5 else x + 1]
                                         ^
SyntaxError: invalid syntax
```

5.17.13 Questions - filtered list comprehensions

Look at the following code fragments, and for each try guessing the result it produces (or if it gives an error):

1. `[x for x in range(100) if False]`
2. `[x for x in range(3) if True]`
3. `[x for x in range(6) if x > 3 else 55]`
4. `[x for x in range(6) if x % 2 == 0]`
5. `[x for x in {'a','b','c'}]  # careful about ordering`
6. `[x for x in [[5], [2,3], [4,2,3], [4]] if len(x) > 2]`
7. `[(x,x) for x in 'xyxyxxy' if x != 'x' ]`
8. `[x for x in ['abCdEFg'] if x.upper() == x]`
9. ```
   la = [1,2,3,4,5]
   [x for x in la if x > la[len(la)//2]]
   ```

5.17.14 Exercises - filtered list comprehensions

Exercise - savannah

Given a list of strings, produce a list with only the strings of length greater than 6:

- INPUT: `['zebra', 'leopard', 'giraffe', 'gnu', 'rhinoceros', 'lion']`
- OUTPUT: `['leopard', 'giraffe', 'rhinoceros']`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[74]:
```
# write here
[x for x in ['zebra', 'leopard', 'giraffe', 'gnu', 'rhinoceros', 'lion'] if len(x) >
↪6]
```

[74]:
```
['leopard', 'giraffe', 'rhinoceros']
```

</div>
```

```
[74]: # write here
```

```
[74]: ['leopard', 'giraffe', 'rhinoceros']
```

Exercise - puZZled

Given a list of strings, produce a list with only the strings which contain at least a `'z'`. The selected strings must be transformed so to place the `Z` in uppercase.

- INPUT: `['puzzled', 'park','Aztec', 'run', 'mask', 'zodiac']`
- OUTPUT: `['puZZled', 'AZtec', 'Zodiac']`

```
[75]: [x.replace('z','Z') for x in ['puzzled', 'park','Aztec', 'run', 'mask', 'zodiac'] if
↪'z' in  x]
```

```
[75]: ['puZZled', 'AZtec', 'Zodiac']
```

5.17.15 References

- Think Python - Chapter 19.2[276]

```
[ ]:
```

5.18 Matrices: list of lists

5.18.1 Download exercises zip

Browse files online[277]

5.18.2 Introduction

Python natively does not provide easy and efficient ways to manipulate matrices. To do so, you would need an external library called numpy which will be seen later in the course. For now we will limit ourselves to using matrices as lists of lists because

There are a couple of ways in Python to represent matrices: as lists of lists, or with the external library Numpy[278]. The most used is surely Numpy but we see both representations anyway. Let's see the reason and main differences:

Lists of lists - as in this notebook:

1. native in Python
2. not efficient
3. lists are pervasive in Python, you will probably encounter matrices expressed as lists of lists anyway
4. you get an idea of how to construct a nested data structure

[276] http://greenteapress.com/thinkpython2/html/thinkpython2020.html#sec224
[277] https://github.com/DavidLeoni/softpython-en/tree/master/matrices-lists
[278] https://www.numpy.org/

5. we can discuss memory referencies and copies along the way

Numpy - see other tutorial Numpy matrices[279] 1. not natively available in Python 1. efficient 1. used by many scientific libraries (scipy, pandas) 1. the syntax to access elements is slightly different from lists of lists 1. in rare cases it might bring installation problems and/or conflicts (implementation is not pure Python)

5.18.3 What to do

- unzip exercises in a folder, you should get something like this:

```
matrix-lists
    matrix-list.ipynb
    matrix-list-sol.ipynb
    jupman.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

- open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `matrices-lists/matrices-lists.ipynb`
- Go on reading that notebook, and follow instuctions inside.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

Overview

So let's see these lists of lists.For example, we can consider the following a matrix with 3 rows and 2 columns, or in short 3x2 matrix:

[2]:
```
m = [
        ['a','b'],
        ['c','d'],
        ['a','e']
    ]
```

For convenience, we assume as input to our functions there won't be matrices with no rows, nor rows with no columns.

Going back to the example, in practice we have a big external list:

```
m = [


]
```

and each of its elements is another list which represents a row:

279 https://it.softpython.org/matrices-numpy/matrices-numpy-sol.html

```
m = [
        ['a','b'],
        ['c','d'],
        ['a','e']
    ]
```

So, to access the whole first row `['a','b']`, we would simply access the element at index 0 of the external list `m`:

```
[3]: m[0]
```

```
[3]: ['a', 'b']
```

To access the second whole second row `['c','d']`, we would access the element at index 1 of the external list `m`:

```
[4]: m[1]
```

```
[4]: ['c', 'd']
```

To access the second whole third row `['c','d']`, we would access the element at index 2 of the external list `m`:

```
[5]: m[2]
```

```
[5]: ['a', 'e']
```

To access the first element `'a'` of the first row `['a','b']` we would add another subscript operator with index 0:

```
[6]: m[0][0]
```

```
[6]: 'a'
```

To access the second elemnt `'b'` of the first row `['a','b']` we would use instead index 1 :

```
[7]: m[0][1]
```

```
[7]: 'b'
```

WARNING: When a matrix is a list of lists, you can only access values with notation `m[i][j]`, **NOT** with `m[i,j]` !!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[8]: # write here the wrong notation m[0,0] and see which error you get:
```

</div>

```
[8]: # write here the wrong notation m[0,0] and see which error you get:
```

5.18.4 Exercises

Now implement the following functions.

REMEMBER: if the cell is executed and nothing happens, it is because all the assert tests have worked! In such case you probably wrote correct code but careful, these kind of tests are never exhaustive so you could have still made some error.

III COMMANDMENT[280]: **You shall never reassign function parameters**

Do not perform any of these assignments, as you risk losing the parameter passed during function call:

[9]:
```python
def disgrace(my_list):
    my_list = [666]      # you lost the [1,2,3] passed from external call!
    print(my_list)       # prints 666

x = [1,2,3]
disgrace(x)
```

```
[666]
```

For the only case when you have composite parameters like lists or dictionaries, you can write like below IF AND ONLY IF the function description requires to MODIFY the internal elements of the parameter (like for example sorting a list in-place or changing the field of a dictionary.

[10]:
```python
# MODIFY my_list in some way
def allowed(my_list):
    my_list[2] = 9

outside = [8,5,7]
allowed(outside)
print(outside)
```

```
[8, 5, 9]
```

VI COMMANDMENT[281] **You shall use `return` command only if you see written RETURN in the function description!**

If there is no `return` in function description, the function is intended to return `None`. In this case you don't even need to write `return None`, as Python will do it implicitly for you.

If the function requires to RETURN a NEW object, you shall not fall into the temptation of modifying the input:

[11]:
```python

# RETURN a NEW sorted list
def pain(my_list):
    my_list.sort()     # BAD, you are modifying the input list instead of creating a␣
↪new one!
    return my_list
```

[280] https://en.softpython.org/commandments.html#III-COMMANDMENT
[281] https://en.softpython.org/commandments.html#VI-COMMANDMENT

[12]:
```python
# RETURN a NEW list
def crisis(my_list):
    my_list[0] = 5     # BAD, as above
    return my_list
```

Matrix dimensions

⊛ **EXERCISE**: For getting matrix dimensions, we can use normal list operations. Which ones? You can assume the matrix is well formed (all rows have equal length) and has at least one row and at least one column

[13]:
```python
m = [
        ['a','b'],
        ['c','d'],
        ['a','e']
    ]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[14]:
```python
# write here code for printing rows and columns

# the outer list is a list of rows, so to count htem we just use len(m)

print("rows")
print(len(m))

# if we assume the matrix is well formed and has at least one row and column,
# we can directly check the length of the first row

print("columns")
print(len(m[0]))
```

```
rows
3
columns
2
```

</div>

[14]:
```python
# write here code for printing rows and columns
```

```
rows
3
columns
2
```

5.18.5 Extracting rows and columns

How to extract a row

One of the first things you might want to do is to extract the i-th row. If you're implementing a function that does this, you have basically two choices. Either you

1. return a *pointer* to the *original* row
2. return a *copy* of the row.

Since a copy consumes memory, why should you ever want to return a copy? Sometimes you should because you don't know which use will be done of the data structure. For example, suppose you got a book of exercises which has empty spaces to write exercises in. It's such a great book everybody in the classroom wants to read it - but you are afraid if the book starts changing hands some careless guy might write on it. To avoid problems, you make a copy of the book and distribute it (let's leave copyright infringment matters aside :-)

Extracting row pointers

So first let's see what happens when you just return a *pointer* to the *original* row.

NOTE: For convenience, at the end of the cell we put a magic call to `jupman.pytut()` which shows the code execution like in Python tutor (for further info about `jupman.pytut()`, see here[282]). If you execute all the code in Python tutor, you will see that at the end you have two arrow pointers to the row `['a','b']`, one starting from `m` list and one from `row` variable.

[15]:
```
# WARNING: FOR PYTHON TUTOR TO WORK, REMEMBER TO EXECUTE THIS CELL with Shift+Enter
#          (it's sufficient to execute it only once)
import jupman
```

[16]:
```
def extrowp(mat, i):
    """ RETURN the ith row from mat
    """
    return mat[i]


m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

row = extrowp(m, 0)

jupman.pytut()
```

[16]:
```
<IPython.core.display.HTML object>
```

[282] https://en.softpython.org/tools/tools-sol.html#Visualizing-the-execution-with-Python-Tutor

extrowf

⊕ Now try to implement a version which returns a **copy** of the row.

QUESTION: You might be tempted to implement something like this - but it wouldn't work. Why?

[17]:
```python
# WARNING: WRONG CODE!!!!

def extrow_wrong(mat, i):
    """ RETURN the ith row from mat.
        NOTE: the row MUST be a new list ! """

    riga = []
    riga.append(mat[i])
    return riga

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

row = extrow_wrong(m,0)

jupman.pytut()
```

[17]: <IPython.core.display.HTML object>

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: The code above adds a LIST as element to another empty list. In other words, it is wrapping the row (which is already a list) into another list. If you checj the problem in Python Tutor, you will see an arrow going from row to a list of one element which will contain exactly one arrow to the original row.

</div>

You can build an actual copy in several ways, with a `for`, a slice or a list comprehension. Try to implement all versions, starting with the `for` here. Be sure to check your result with Python Tutor - to visualize python tutor inside the cell output, you might use the special command `jupman.pytut()` at the end of the cell as we did before. If you run the code with Python Tutor, you should only see *one* arrow going to the original `['a','b']` row in `m`, and there should be *another* `['a','b']` copy somewhere, with `row` variable pointing to it.

⊕ **EXERCISE**: Implement the function `esrowf` which RETURNS the `i`-th row from `mat`

- **NOTE**: the row MUST be a new list! To create a new list use a for cycle which iterates over the elements, *not* the indeces (so don't use range!)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[18]:
```python
def extrowf(mat, i):
    """ RETURN the ith row from mat.

    """

    riga = []
    for x in mat[i]:
        riga.append(x)
    return riga
```

(continues on next page)

(continued from previous page)

```
m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extrowf(m, 0) == ['a','b']
assert extrowf(m, 1) == ['c','d']
assert extrowf(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrowf(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'

# uncomment to visualize execution here
#jupman.pytut()
```

</div>

[18]:

```
def extrowf(mat, i):
    """ RETURN the ith row from mat.

    """
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extrowf(m, 0) == ['a','b']
assert extrowf(m, 1) == ['c','d']
assert extrowf(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrowf(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'

# uncomment to visualize execution here
#jupman.pytut()
```

Extract row with range

Let's first rapidly see `range(n)`. Maybe you think it should return a sequence of integers, from zero to `n - 1`. Is it really like this?

```
[19]: range(5)
```

```
[19]: range(0, 5)
```

Maybe you expected something like a list `[0,1,2,3,4]`, instead we discovered that Python is quite lazy here: as a matter of fact, `range(n)` returns an *iterable* object, which is not a real sequence materialized in memory.

To take a real integer list, we must explicitly ask this iterable object to give us the objects one by one.

When you write `for i in range(s)` the `for` loop is doing exactly this, at each round it asks the object `range` to generate a number from the sequence. If we want the whole sequence materialized in memory, we can generate it by converting the `range` into a list object:

```
[20]: list(range(5))
```

```
[20]: [0, 1, 2, 3, 4]
```

Be careful, though. According to the sequence dimension, this might be dangerous. A billion elements list might saturate your computer RAM (in 2020 notebooks often have 4 gigabytes of RAM, that is, 4 billion bytes).

⊕ **EXERCISE**: Now implement the `extrowr` iterating over a range of row indexes:

- **NOTE 1**: the row MUST be a new list! To create a new list use a `for` loop
- **NOTE 2**: remember to use a new name for the column index!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[21]: def extrowr(mat, i):
          """ RETURN the ith row from mat.
          """

          riga = []
          for j in range(len(mat[0])):
              riga.append(mat[i][j])
          return riga


      m = [
            ['a','b'],
            ['c','d'],
            ['a','e'],
      ]

      assert extrowr(m, 0) == ['a','b']
      assert extrowr(m, 1) == ['c','d']
      assert extrowr(m, 2) == ['a','e']

      # check it didn't change the original matrix !
      r = extrowr(m, 0)
      r[0] = 'z'
      assert m[0][0] == 'a'

      # uncomment to visualize execution here
      #jupman.pytut()
```

</div>

[21]:
```python
def extrowr(mat, i):
    """ RETURN the ith row from mat.
    """
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extrowr(m, 0) == ['a','b']
assert extrowr(m, 1) == ['c','d']
assert extrowr(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrowr(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'

# uncomment to visualize execution here
#jupman.pytut()
```

Extract row with a slice

⊛ Remember slices return a *copy* of a list? Now try to use them.

- **NOTE**: the row MUST be a new list! To create it, use slices.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```python
def extrows(mat, i):
    """ RETURN the ith row from mat.
    """

    return mat[i][:]  # if you omit start end end indexes, you get a copy of the␣
↪whole list


m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]


assert extrows(m, 0) == ['a','b']
assert extrows(m, 1) == ['c','d']
assert extrows(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrows(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'
```

(continues on next page)

(continued from previous page)

```
# uncomment to visualize execution here
#jupman.pytut()
```

</div>

[22]:
```
def extrows(mat, i):
    """ RETURN the ith row from mat.
    """
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]


assert extrows(m, 0) == ['a','b']
assert extrows(m, 1) == ['c','d']
assert extrows(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrows(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'

# uncomment to visualize execution here
#jupman.pytut()
```

Extract row with list comprehension

⊛ Implement `extrowc`, which RETURNs the `i`-th row from `mat`. To create a new list use a *list comprehension*.

- **NOTE**: the row MUST be a new list!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```
def extrowc(mat, i):

    return [x for x in mat[i]]


m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extrowc(m, 0) == ['a','b']
assert extrowc(m, 1) == ['c','d']
assert extrowc(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrowc(m, 0)
```

(continues on next page)

(continued from previous page)

```
r[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

</div>

[23]:
```
def extrowc(mat, i):
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extrowc(m, 0) == ['a','b']
assert extrowc(m, 1) == ['c','d']
assert extrowc(m, 2) == ['a','e']

# check it didn't change the original matrix !
r = extrowc(m, 0)
r[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

Extract column with a `for`

⊛⊛ Now try extracting a column at `j`th position. This time we will be forced to create a new list, so we don't have to wonder if we need to return a pointer or a copy.

Implement `extcolf`, which RETURN the `j`-th column from `mat`. To create it, use a `for` loop.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[24]:
```
def extcolf(mat, j):

    ret = []
    for row in mat:
        ret.append(row[j])
    return ret


m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extcolf(m, 0) == ['a','c','a']
assert extcolf(m, 1) == ['b','d','e']

# check returned column does not modify m
```

(continues on next page)

(continued from previous page)

```
c = extcolf(m,0)
c[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

</div>

[24]:
```
def extcolf(mat, j):
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extcolf(m, 0) == ['a','c','a']
assert extcolf(m, 1) == ['b','d','e']

# check returned column does not modify m
c = extcolf(m,0)
c[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

Extract column with a list comprehension

⊛⊛ Implement `extcolc`, which RETURNS the `j`-th column from `mat`: to create it, use a list comprehension.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[25]:
```
def extcolc(mat, j):

    return [row[j] for row in mat]


m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extcolc(m, 0) == ['a','c','a']
assert extcolc(m, 1) == ['b','d','e']

# check returned column does not modify m
c = extcolc(m,0)
c[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

</div>

[25]:
```python
def extcolc(mat, j):
    raise Exception('TODO IMPLEMENT ME !')

m = [
      ['a','b'],
      ['c','d'],
      ['a','e'],
]

assert extcolc(m, 0) == ['a','c','a']
assert extcolc(m, 1) == ['b','d','e']

# check returned column does not modify m
c = extcolc(m,0)
c[0] = 'z'
assert m[0][0] == 'a'

#jupman.pytut()
```

5.18.6 Creating new matrices

empty matrix

⊛⊛ There are several ways to create a new empty 3x5 matrix as lists of lists which contains zeros.

Implement `empty_matrix`, which RETURN a NEW matrix nxn as a list of lists filled with zeroes

- use two nested `for` cycles:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[26]:
```python
def empty_matrix(n, m):

    ret = []
    for i in range(n):
        row = []
        ret.append(row)
        for j in range(m):
            row.append(0)
    return ret


assert empty_matrix(1,1) == [[0]]

assert empty_matrix(1,2) == [ [0,0] ]

assert empty_matrix(2,1) == [ [0],
                              [0] ]

assert empty_matrix(2,2) == [ [0,0],
                              [0,0] ]

assert empty_matrix(3,3) == [ [0,0,0],
                              [0,0,0],
                              [0,0,0] ]
```

</div>

```
[26]: def empty_matrix(n, m):
          raise Exception('TODO IMPLEMENT ME !')

      assert empty_matrix(1,1) == [[0]]

      assert empty_matrix(1,2) == [ [0,0] ]

      assert empty_matrix(2,1) == [ [0],
                                    [0] ]

      assert empty_matrix(2,2) == [ [0,0],
                                    [0,0] ]

      assert empty_matrix(3,3) == [ [0,0,0],
                                    [0,0,0],
                                    [0,0,0] ]
```

`empty_matrix` the elegant way

To create a new list of 3 elements filled with zeros, you can write like this:

```
[27]: [0]*3
```

```
[27]: [0, 0, 0]
```

The `*` is kind of multiplying the elements in a list

Given the above, to create a 5x3 matrix filled with zeros, which is a list of seemingly equal lists, you might then be tempted to write like this:

```
[28]: # WRONG
      [[0]*3]*5
```

```
[28]: [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Why is that (possibly) wrong? Let's try to inspect it in Python Tutor:

```
[29]: bad = [[0]*3]*5
      jupman.pytut()
```

```
[29]: <IPython.core.display.HTML object>
```

If you look closely, you will see many arrows pointing to the same list of 3 zeros. This means that if we change one number, we will apparently change 5 of them in the whole column !

The right way to create a matrix as list of lists with zeroes is the following:

```
[30]: # CORRECT
      [[0]*3 for i in range(5)]
```

```
[30]: [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

EXERCISE: Try creating a matrix with 7 rows and 4 columns and fill it with 5.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[31]:
```python
# write here

[[5]*4 for i in range(7)]
```

[31]:
```
[[5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5]]
```

</div>

[31]:
```python
# write here
```

[31]:
```
[[5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5],
 [5, 5, 5, 5]]
```

deep_clone

⊛⊛ Let's try to produce a *complete* clone of the matrix, also called a *deep clone*, by creating a copy of the external list *and* also the internal lists representing the rows.

QUESTION: You might be tempted to write code like this, but it will not work. Why?

[32]:
```python

# WARNING: WRONG CODE
def deep_clone_wrong(mat):
    """ RETURN a NEW list of lists which is a COMPLETE DEEP clone
        of mat (which is a list of lists)
    """
    return mat[:]

m = [
        ['a','b'],
        ['b','d']
    ]

res = deep_clone_wrong(m)

# Notice you will have arrows in res list going to the _original_ mat. We don't want␣
↪this !
jupman.pytut()
```

[32]:
```
<IPython.core.display.HTML object>
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: `return mat[:]` is not sufficient, because it's a SHALLOW clone, and only copies the *external* list and not also the internal ones ! Note you will have rows in the `res` list which goes to the original matrix. We don't want this!

</div>

To fix the above code, you will need to iterate through the rows and *for each* row create a copy of that row.

⊛⊛ **EXERCISE**: Implement `deep_clone`, which RETURNS a NEW list as a complete DEEP CLONE of `mat` (which is a list of lists)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[33]:
```python
def deep_clone(mat):


    ret = []
    for row in mat:
        ret.append(row[:])
    return ret


m = [ ['a','b'],
      ['b','d'] ]

res = [ ['a','b'],
        ['b','d'] ]

# verify the copy
c = deep_clone(m)
assert c == res

# verify it is a DEEP copy (that is, it created also clones of the rows!)
c[0][0] = 'z'
assert m[0][0] == 'a'
```

</div>

[33]:
```python
def deep_clone(mat):
    raise Exception('TODO IMPLEMENT ME !')

m = [ ['a','b'],
      ['b','d'] ]

res = [ ['a','b'],
        ['b','d'] ]

# verify the copy
c = deep_clone(m)
assert c == res

# verify it is a DEEP copy (that is, it created also clones of the rows!)
c[0][0] = 'z'
assert m[0][0] == 'a'
```

5.18.7 Modifying matrices

fillc

⊛⊛ Implement the function `fillc` which takes as input `mat` (a list of lists with dimension `nrows` x `ncol`) and MODIFIES it by placing the character `c` inside all the matrix cells.

- to visit the matrix use for in range cycles

Ingredients:

- find matrix dimension
- two nested fors
- use range

NOTE: This function returns nothing!

If in the function text it is not mentioned to return values, DO NOT place the `return`. If by chance you put it anyway it is not the world's end, but to avoid confusion is much better having a behaviour consisting with the text.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[34]:
```python
def fillc(mat, c):

    nrows = len(mat)
    ncols = len(mat[0])

    for i in range(nrows):
        for j in range(ncols):
            mat[i][j] = c


m1 = [ ['a'] ]
m2 = [ ['z'] ]
fillc(m1,'z')
assert m1 == m2

m3 = [ ['a'] ]
m4 = [ ['y'] ]
fillc(m3,'y')
assert m3 == m4

m5 = [ ['a','b'] ]
m6 = [ ['z','z'] ]
fillc(m5,'z')
assert m5 == m6

m7 = [ ['a','b','c'],
       ['d','e','f'],
       ['g','h','i'] ]

m8 = [ ['y','y','y'],
       ['y','y','y'],
       ['y','y','y'] ]
fillc(m7,'y')
```

(continues on next page)

(continued from previous page)

```
assert m7 == m8

#      j  0   1
m9 = [ ['a','b'],     # 0
       ['c','d'],     # 1
       ['e','f'] ]    # 2

m10 = [ ['x','x'],     # 0
        ['x','x'],     # 1
        ['x','x'] ]    # 2
fillc(m9, 'x')
assert m9 == m10
```

</div>

[34]:
```
def fillc(mat, c):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a'] ]
m2 = [ ['z'] ]
fillc(m1,'z')
assert m1 == m2

m3 = [ ['a'] ]
m4 = [ ['y'] ]
fillc(m3,'y')
assert m3 == m4

m5 = [ ['a','b'] ]
m6 = [ ['z','z'] ]
fillc(m5,'z')
assert m5 == m6

m7 = [ ['a','b','c'],
       ['d','e','f'],
       ['g','h','i'] ]

m8 = [ ['y','y','y'],
       ['y','y','y'],
       ['y','y','y'] ]
fillc(m7,'y')
assert m7 == m8

#      j  0   1
m9 = [ ['a','b'],     # 0
       ['c','d'],     # 1
       ['e','f'] ]    # 2

m10 = [ ['x','x'],     # 0
        ['x','x'],     # 1
        ['x','x'] ]    # 2
fillc(m9, 'x')
assert m9 == m10
```

fillx

⊛⊛ Takes a matrix mat as list of lists and a column index `j`, and MODIFIES `mat` by placing the `'x'` character in all cells of the `j`-th column.

Example:

```
m = [
      ['a','b','c','d'],
      ['e','f','g','h'],
      ['i','l','m','n']
    ]
```

After the call to

```
fillx(m,2)
```

the matrix `m` will be changed like this:

```
>>> print(m)

[
    ['a','b','x','d'],
    ['e','f','x','h'],
    ['i','l','x','n']
]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[35]:
```
def fillx(mat, j):

    for row in mat:
        row[j] = 'x'




m1 = [ ['a'] ]
fillx(m1,0)
assert m1 == [ ['x'] ]

m2 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillx(m2,0)
assert m2 == [ ['x','b'],
               ['x','d'],
               ['x','f'] ]

m3 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillx(m3,1)
assert m3 == [ ['a','x'],
               ['c','x'],
               ['e','x'] ]

m4 = [ ['a','b','c','d'],
```

(continues on next page)

(continued from previous page)

```
       ['e','f','g','h'],
       ['i','l','m','n'] ]
fillx(m4,2)
assert m4 == [ ['a','b','x','d'],
               ['e','f','x','h'],
               ['i','l','x','n'] ]
```

</div>

[35]:
```
def fillx(mat, j):
    raise Exception('TODO IMPLEMENT ME !')


m1 = [ ['a'] ]
fillx(m1,0)
assert m1 == [ ['x'] ]

m2 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillx(m2,0)
assert m2 == [ ['x','b'],
               ['x','d'],
               ['x','f'] ]

m3 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillx(m3,1)
assert m3 == [ ['a','x'],
               ['c','x'],
               ['e','x'] ]

m4 = [ ['a','b','c','d'],
       ['e','f','g','h'],
       ['i','l','m','n'] ]
fillx(m4,2)
assert m4 == [ ['a','b','x','d'],
               ['e','f','x','h'],
               ['i','l','x','n'] ]
```

fillz

⊛⊛ Takes a matrix `mat` as list of lists and a row index `i`, and MODIFIES `mat` by placing the character `'z'` in all the cells of the `i`-th row.

Example:

```
m = [
    ['a','b'],
    ['c','d'],
    ['e','f'],
    ['g','h']
]
```

After the call to

```
fillz(m,2)
```

the matrix `m` will be changed like this:

```
>>> print(m)

[
    ['a','b'],
    ['c','d'],
    ['z','z'],
    ['g','h']
]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[36]:
```
def fillz(mat, i):

    ncol=len(mat[0])
    for j in range(ncol):
        mat[i][j] = 'z'



m1 = [ ['a'] ]
fillz(m1,0)
assert m1 == [ ['z'] ]

m2 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m2,0)
assert m2 == [ ['z','z'],
               ['c','d'],
               ['e','f'] ]

m3 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m3,1)
assert m3 == [ ['a','b'],
               ['z','z'],
               ['e','f'] ]

m4 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m4,2)
assert m4 == [ ['a','b'],
               ['c','d'],
               ['z','z'] ]
```

</div>

[36]:
```
def fillz(mat, i):
    raise Exception('TODO IMPLEMENT ME !')
```

(continues on next page)

(continued from previous page)

```
m1 = [ ['a'] ]
fillz(m1,0)
assert m1 == [ ['z'] ]

m2 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m2,0)
assert m2 == [ ['z','z'],
               ['c','d'],
               ['e','f'] ]

m3 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m3,1)
assert m3 == [ ['a','b'],
               ['z','z'],
               ['e','f'] ]

m4 = [ ['a','b'],
       ['c','d'],
       ['e','f'] ]
fillz(m4,2)
assert m4 == [ ['a','b'],
               ['c','d'],
               ['z','z'] ]
```

stitch_down

⊛⊛ Given matrices `mat1` and `mat2` as list of lists, with mat1 of size `u` x `n` and `mat2` of size `d` x `n`, RETURN a NEW matrix of size (`u`+`d`) x `n` as list of lists, by stitching second mat to the bottom of `mat1`

- NOTE: by NEW matrix we intend a matrix with no pointers to original rows (see previous deep clone exercise)
- for examples, see assert

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[37]:
```
def stitch_down(mat1, mat2):

    res = []
    for row in mat1:
        res.append(row[:])
    for row in mat2:
        res.append(row[:])
    return res


m1 = [ ['a'] ]
m2 = [ ['b'] ]
assert stitch_down(m1, m2) == [ ['a'],
                                ['b'] ]
```

(continues on next page)

(continued from previous page)

```
# check we are giving back a deep clone
s = stitch_down(m1, m2)
s[0][0] = 'z'
assert m1[0][0] == 'a'

m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b', 'h'],
       ['g','h', 'w'] ]

res = [ ['a','b','c'],
        ['d','b','a'],
        ['f','b','h'],
        ['g','h','w'] ]

assert stitch_down(m1, m2) == res
```

</div>

[37]:
```
def stitch_down(mat1, mat2):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a'] ]
m2 = [ ['b'] ]
assert stitch_down(m1, m2) == [ ['a'],
                                ['b'] ]

# check we are giving back a deep clone
s = stitch_down(m1, m2)
s[0][0] = 'z'
assert m1[0][0] == 'a'

m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b', 'h'],
       ['g','h', 'w'] ]

res = [ ['a','b','c'],
        ['d','b','a'],
        ['f','b','h'],
        ['g','h','w'] ]

assert stitch_down(m1, m2) == res
```

stitch_up

⊛⊛ Given matrices `mat1` and `mat2` as list of lists, with `mat1` of size `u` x `n` and `mat2` of size `d` x `n`, RETURN a NEW matrix of size (`u`+`d`) x `n` as list of lists, by stitching first mat to the bottom of mat2

- **NOTE**: by NEW matrix we intend a matrix with no pointers to original rows (see previous `deep_clone` exercise)
- To implement this function, use a call to the method stitch_down you implemented before.
- For examples, see assert

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[38]:
```python
def stitch_up(mat1, mat2):

    return stitch_down(mat2, mat1)



m1 = [ ['a'] ]
m2 = [ ['b'] ]
assert stitch_up(m1, m2) == [ ['b'],
                               ['a'] ]

# check we are giving back a deep clone
s = stitch_up(m1, m2)
s[0][0] = 'z'
assert m1[0][0] == 'a'

m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b', 'h'],
       ['g','h', 'w'] ]

res = [ ['f','b','h'],
        ['g','h','w'],
        ['a','b','c'],
        ['d','b','a'] ]

assert stitch_up(m1, m2) == res
```

</div>

[38]:
```python
def stitch_up(mat1, mat2):
    raise Exception('TODO IMPLEMENT ME !')


m1 = [ ['a'] ]
m2 = [ ['b'] ]
assert stitch_up(m1, m2) == [ ['b'],
                               ['a'] ]

# check we are giving back a deep clone
s = stitch_up(m1, m2)
s[0][0] = 'z'
assert m1[0][0] == 'a'

m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b', 'h'],
       ['g','h', 'w'] ]

res = [ ['f','b','h'],
        ['g','h','w'],
        ['a','b','c'],
        ['d','b','a'] ]

assert stitch_up(m1, m2) == res
```

stitch_right

⊛⊛⊛ Given matrices `mata` and `matb` as list of lists, with `mata` of size `n x l` and `matb` of size `n x r`, RETURN a NEW matrix of size `n x (l + r)` as list of lists, by stitching second `matb` to the right end of `mata`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[39]:

```
def stitch_right(mata,matb):

    ret = []
    for i in range(len(mata)):
        row_to_add =  mata[i][:]
        row_to_add.extend(matb[i])
        ret.append(row_to_add)
    return ret


ma1 = [ ['a','b','c'],
        ['d','b','a'] ]
mb1 = [ ['f','b'],
        ['g','h'] ]

r1 = [ ['a','b','c','f','b'],
       ['d','b','a','g','h'] ]

assert stitch_right(ma1, mb1) == r1
```

</div>

[39]:

```
def stitch_right(mata,matb):
    raise Exception('TODO IMPLEMENT ME !')

ma1 = [ ['a','b','c'],
        ['d','b','a'] ]
mb1 = [ ['f','b'],
        ['g','h'] ]

r1 = [ ['a','b','c','f','b'],
       ['d','b','a','g','h'] ]

assert stitch_right(ma1, mb1) == r1
```

threshold

⊛⊛ Takes a matrix as a list of lists (every list has the same dimension) and RETURN a NEW matrix as list of lists where there is `True` if the corresponding input element is greater than `t`, otherwise return `False`

Ingredients:

- a variable for the matrix to return
- for each original row, we need to create a new list

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[40]:
```
def threshold(mat, t):

    ret = []
    for row in mat:
        new_row = []
        ret.append(new_row)
        for el in row:
            new_row.append(el > t)

    return ret


morig = [ [1,4,2],
          [7,9,3] ]

m1 = [ [1,4,2],
       [7,9,3] ]

r1 = [ [False,False,False],
       [True,True,False] ]
assert threshold(m1,4) == r1
assert m1 == morig   # verify original didn't change

m2 = [ [5,2],
       [3,7] ]

r2 = [ [True,False],
       [False,True] ]
assert threshold(m2,4) == r2
```

</div>

[40]:
```
def threshold(mat, t):
    raise Exception('TODO IMPLEMENT ME !')

morig = [ [1,4,2],
          [7,9,3] ]

m1 = [ [1,4,2],
       [7,9,3] ]

r1 = [ [False,False,False],
       [True,True,False] ]
assert threshold(m1,4) == r1
assert m1 == morig   # verify original didn't change

m2 = [ [5,2],
       [3,7] ]

r2 = [ [True,False],
       [False,True] ]
assert threshold(m2,4) == r2
```

swap_rows

⊛⊛ We will try swapping a couple of rows of a matrix

There are several ways to proceed. Before continuing, make sure to know how to exchange two values by solving this simple exercise - check your result in Python Tutor

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[41]:
```python
x = 3
y = 7

# write here the code to swap x and y (do not directly use the constants 3 and 7!)

k = x
x = y
y = k

#jupman.pytut()
```

</div>

[41]:
```python
x = 3
y = 7

# write here the code to swap x and y (do not directly use the constants 3 and 7!)
```

⊛⊛ Takes a matrix `mat` as list of lists, and RETURN a NEW matrix where rows at indexes `i1` and `i2` are swapped

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[42]:
```python
def swap_rows(mat, i1, i2):


    # deep clones
    ret = []
    for row in mat:
        ret.append(row[:])
    #swaps
    s = ret[i1]
    ret[i1] = ret[i2]
    ret[i2] = s
    return ret


m1 = [ ['a','d'],
       ['b','e'],
       ['c','f'] ]

r1 = swap_rows(m1, 0, 2)

assert r1 == [ ['c','f'],
               ['b','e'],
```

(continues on next page)

(continued from previous page)

```
                ['a','d'] ]

r1[0][0] = 'z'
assert m1[0][0] == 'a'

m2 = [ ['a','d'],
       ['b','e'],
       ['c','f'] ]


# swap with itself should in fact generate a deep clone
r2 = swap_rows(m2, 0, 0)

assert r2 == [ ['a','d'],
               ['b','e'],
               ['c','f'] ]

r2[0][0] = 'z'
assert m2[0][0] == 'a'
```

</div>

[42]:
```
def swap_rows(mat, i1, i2):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a','d'],
       ['b','e'],
       ['c','f'] ]

r1 = swap_rows(m1, 0, 2)

assert r1 == [ ['c','f'],
               ['b','e'],
               ['a','d'] ]

r1[0][0] = 'z'
assert m1[0][0] == 'a'

m2 = [ ['a','d'],
       ['b','e'],
       ['c','f'] ]


# swap with itself should in fact generate a deep clone
r2 = swap_rows(m2, 0, 0)

assert r2 == [ ['a','d'],
               ['b','e'],
               ['c','f'] ]

r2[0][0] = 'z'
assert m2[0][0] == 'a'
```

swap_cols

⊛⊛ Takes a matrix `mat` and two column indeces `j1` and `j2` and RETURN a NEW matrix where the columns `j1` and `j2` are swapped

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[43]:
```python
def swap_cols(mat, j1, j2):

    ret = []
    for row in mat:
        new_row = row[:]
        new_row[j1] = row[j2]
        new_row[j2] = row[j1]
        ret.append(new_row)
    return ret


m1 = [ ['a','b','c'],
       ['d','e','f'] ]

r1 = swap_cols(m1, 0,2)

assert r1 == [ ['c','b','a'],
               ['f','e','d'] ]

r1[0][0] = 'z'
assert m1[0][0] == 'a'
```

</div>

[43]:
```python
def swap_cols(mat, j1, j2):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a','b','c'],
       ['d','e','f'] ]

r1 = swap_cols(m1, 0,2)

assert r1 == [ ['c','b','a'],
               ['f','e','d'] ]

r1[0][0] = 'z'
assert m1[0][0] == 'a'
```

5.18.8 Other exercises

diag

`diag` extracts the diagonal of a matrix. To do so, `diag` requires an nxn matrix as input. To make sure we actually get an nxn matrix, this time you will have to validate the input, that is check if the number of rows is equal to the number of columns (as always we assume the matrix has at least one row and at least one column). If the matrix is not nxn, the function should stop raising an exception. In particular, it shoud raise a ValueError[283], which is the standard Python exception to raise when the expected input is not correct and you can't find any other more specific error.

[283] https://docs.python.org/3/library/exceptions.html#ValueError

Just for illustrative puroposes, we show here the index numbers `i` and `j` and avoid putting apices around strings:

```
\ j  0,1,2,3
i
   [
0   [a,b,c,d],
1   [e,f,g,h],
2   [p,q,r,s],
3   [t,u,v,z]
   ]
```

Let's see a step by step execution:

```
                              \ j  0,1,2,3
                              i
                                 [
extract from row at i=0  -->  0   [a,b,c,d],   'a' is extracted from  mat[0][0]
                              1   [e,f,g,h],
                              2   [p,q,r,s],
                              3   [t,u,v,z]
                                 ]
```

```
                              \ j  0,1,2,3
                              i
                                 [
                              0   [a,b,c,d],
extract from row at i=1  -->  1   [e,f,g,h],   'f' is extracted from mat[1][1]
                              2   [p,q,r,s],
                              3   [t,u,v,z]
                                 ]
```

```
                              \ j  0,1,2,3
                              i
                                 [
                              0   [a,b,c,d],
                              1   [e,f,g,h],
extract from row at i=2  -->  2   [p,q,r,s],   'r' is extracted from mat[2][2]
                              3   [t,u,v,z]
                                 ]
```

```
                               \ j  0,1,2,3
                               i
                                  [
                               0   [a,b,c,d],
                               1   [e,f,g,h],
                               2   [p,q,r,s],
 extract from row at i=3  -->  3   [t,u,v,z]    'z' is extracted from mat[3][3]
                                  ]

From the above, we notice we need elements from these indeces:

 i, j
 1, 1
 2, 2
 3, 3
```

There are two ways to solve this exercise, one is to use a double `for` (a nested for to be precise) while the other method uses only one `for`. Try to solve it in both ways. How many steps do you need with double for? and with only one?

⊛⊛ **EXERCISE**: Implement the `diag` function, which given an `nxn` matrix `mat` as a list of lists, RETURN a list which contains the elemets in the diagonal (top left to bottom right corner).

- if `mat` is not `nxn` raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[44]:

```
def diag(mat):

    if len(mat) != len(mat[0]):
        raise ValueError("Matrix should be nxn, found instead %s x %s" % (len(mat),␣
↪len(mat[0])))
    ret = []
    for i in range(len(mat)):
        ret.append(mat[i][i])
    return ret


# TEST START - DO NOT TOUCH!
# if you wrote the whole code correct, and execute the cell, Python shouldn't raise␣
↪`AssertionError`
m = [ ['a','b','c'],
      ['d','e','f'],
      ['g','h','i'] ]

assert diag(m) == ['a','e','i']

try:
    diag([ ['a','b'] ])     # 1x2 dimension, not square

    raise Exception("SHOULD HAVE FAILED !")  # if diag raises an exception which is␣
↪ValueError as we
                                             # expect it to do, the code should never␣
↪arrive here

except ValueError: # this only catches ValueError. Other types of errors are not␣
↪catched
    pass           # In an except clause you always need to put some code.
                   # Here we put a placeholder just to fill in
# TEST END
```

</div>

[44]:

```
def diag(mat):
    raise Exception('TODO IMPLEMENT ME !')

# TEST START - DO NOT TOUCH!
# if you wrote the whole code correct, and execute the cell, Python shouldn't raise␣
↪`AssertionError`
m = [ ['a','b','c'],
      ['d','e','f'],
      ['g','h','i'] ]

assert diag(m) == ['a','e','i']

try:
```

(continues on next page)

(continued from previous page)

```
    diag([ ['a','b'] ])     # 1x2 dimension, not square

    raise Exception("SHOULD HAVE FAILED !")  # if diag raises an exception which is␣
↪ValueError as we
                                              # expect it to do, the code should never␣
↪arrive here

except ValueError: # this only catches ValueError. Other types of errors are not␣
↪catched
    pass           # In an except clause you always need to put some code.
                   # Here we put a placeholder just to fill in
# TEST END
```

anti_diag

⊛⊛ Given an nxn matrix mat as a list of lists, RETURN a list which contains the elemets in the antidiagonal (top right to bottom left corner).

- If mat is not `nxn` raise `ValueError`

Before implementing it, be sure to write down understand the required indeces as we did in the example for the *diag* function.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[45]:
```
def anti_diag(mat):

    n = len(mat)
    ret = []
    for i in range(n):
        ret.append(mat[i][n-i-1])
    return ret


m = [ ['a','b','c'],
      ['d','e','f'],
      ['g','h','i'] ]

assert anti_diag(m) == ['c','e','g']

# If you have doubts about the indexes remember to try it in python tutor !
# jupman.pytut()
```

</div>

[45]:
```
def anti_diag(mat):
    raise Exception('TODO IMPLEMENT ME !')

m = [ ['a','b','c'],
      ['d','e','f'],
      ['g','h','i'] ]

assert anti_diag(m) == ['c','e','g']
```

(continues on next page)

(continued from previous page)

```
# If you have doubts about the indexes remember to try it in python tutor !
# jupman.pytut()
```

is_utriang

⊛⊛⊛ You will now try to iterate only the lower triangular half of a matrix. Let's look at an example:

[46]:
```
m = [
        [3,2,5,8],
        [0,6,2,3],
        [0,0,4,9],
        [0,0,0,5]
    ]
```

Just for illustrative puroposes, we show here the index numbers `i` and `j`:

```
\ j  0,1,2,3
i
   [
0   [3,2,5,8],
1   [0,6,2,3],
2   [0,0,4,9],
3   [0,7,0,5]
   ]
```

Let's see a step by step execution an a non-upper triangular matrix:

```
                                \ j  0,1,2,3
                                i
                                   [
                                0   [3,2,5,8],
start from row at index i=1 ->  1   [0,6,2,3],  Check until column limit j=0 included
                                2   [0,0,4,9],
                                3   [0,7,0,5]
                                   ]
```

One zero is found, time to check next row.

```
                                \ j  0,1,2,3
                                i
                                   [
                                0   [3,2,5,8],
                                1   [0,6,2,3],
check row at index i=2    --->  2   [0,0,4,9],  Check until column limit j=1 included
                                3   [0,7,0,5]
                                   ]
```

Two zeros are found. Time to check next row.

```
                                \ j  0,1,2,3
                                i
                                   [
                                0   [3,2,5,8],
                                1   [0,6,2,3],
                                2   [0,0,4,9],
```

(continues on next page)

(continued from previous page)

```
check row at index i=3    --->  3   [0,7,0,5]    Check until column limit j=2 included
                                    ]             BUT can stop sooner at j=1 because
                                                  number at j=1 is different from zero.
                                                  As soon as 7 is found, can return␣
↪False
                                                  In this case the matrix is not upper
                                                  triangular
```

VII COMMANDMENT[284] **You shall also write on paper!**

When you develop these algorithms, it is fundamental to write down a step by step example like the above to get a clear picture of what is happening. Also, if you write down the indeces correctly, you will easily be able to derive a generalization. To find it, try to further write the found indeces in a table.

For example, from above for each row index `i` we can easily find out which limit index `j` we need to reach for our hunt for zeros:

i	limit j (included)	Notes
1	0	we start from row at index i=1
2	1	
3	2	

From the table, we can see the limit for j can be calculated in terms of the current row index `i` with the simple formula `i - 1`

The fact you need to span through rows and columns suggest you need two `for`s, one for rows and one for columns - that is, a *nested for*.

- please use ranges of indexes to carry out the task (no `for row in mat` ..)
- please use letter `i` as index for rows, `j` as index of columns and in case you need it `n` letter as matrix dimension

HINT 1: remember you can set range to start from a specific index, like `range(3,7)` will start from 3 and end to 6 *included* (last 7 is *excluded*!)

HINT 2: To implement this, it is best looking for numbers *different* from zero. As soon as you find one, you can stop the function and return False. Only after *all* the number checking is done you can return True.

Finally, be reminded of the following:

II COMMANDMENT[285] **Whenever you introduce a variable with a `for` cycle, such variable must be new**

If you defined a variable before, you shall not reintroduce it in a `for`, since it's confusing and error prone.

So avoid these sins:

[47]:
```python
i = 7
for i in range(3):   # sin, you lose i variable
    print(i)
```

```
0
1
2
```

[284] https://en.softpython.org/commandments.html#VII-COMMANDMENT
[285] https://en.softpython.org/commandments.html#VII-COMMANDMENT

[48]:
```
def f(i):
    for i in range(3): # sin again, you lose i parameter
        print(i)
```

[49]:
```
for i in range(2):
    for i in  range(3):  # debugging hell, you lose i from outer for
        print(i)
```

```
0
1
2
0
1
2
```

⊛⊛⊛ **EXERCISE**: If you read *all* the above, start implementing the function `is_utriang`, which RETURN `True` if the provided `nxn` matrix is upper triangular, that is, has all the entries below the diagonal set to zero. Return `False` otherwise.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[50]:
```
def is_utriang(mat):

    n = len(mat)
    m = len(mat[0])

    for i in range(1,n):
        for j in range(i): # notice it arrives until i *excluded*, that is, arrives␣
↪to i - 1 *included*
            if mat[i][j] != 0:
                return False
    return True


assert is_utriang([ [1] ]) == True
assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [0,0,4] ]) == True

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [1,0,4] ]) == False

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [1,1,4] ]) == False

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [0,1,4] ]) == False


assert is_utriang([ [3,2,5],
                    [1,6,2],
                    [1,0,4] ]) == False
```

</div>

[50]:
```python
def is_utriang(mat):
    raise Exception('TODO IMPLEMENT ME !')

assert is_utriang([ [1] ]) == True
assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [0,0,4] ]) == True

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [1,0,4] ]) == False

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [1,1,4] ]) == False

assert is_utriang([ [3,2,5],
                    [0,6,2],
                    [0,1,4] ]) == False


assert is_utriang([ [3,2,5],
                    [1,6,2],
                    [1,0,4] ]) == False
```

stitch_left_mod

This time let's try to *modify* `mat1` *in place*, by stitching `mat2` *to the left* of `mat1`.

So this time **don't** put a `return` instruction.

You will need to perform list insertion, which can be tricky. There are many ways to do it in Python, one could be using the weird splice assignment insertion:

```python
mylist[0:0] = list_to_insert
```

see here for more info: https://stackoverflow.com/a/10623383

⊛⊛⊛ **EXERCISE**: Implement `stitch_left_mod`, which given the matrices `mat1` and `mat2` as list of lists, with `mat1` of size `n` x `l` and `mat2` of size `n` x `r`, MODIFIES `mat1` so that it becomes of size `n` x (`l` + `r`), by stitching second `mat2` to the left of `mat1`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[51]:
```python
def stitch_left_mod(mat1,mat2):

    for i in range(len(mat1)):
        mat1[i][0:0] = mat2[i]


m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b'],
       ['g','h'] ]

res = [ ['f','b','a','b','c'],
```

(continues on next page)

(continued from previous page)

```
        ['g','h','d','b','a'] ]

stitch_left_mod(m1, m2)
assert m1 == res
```

</div>

[51]:
```
def stitch_left_mod(mat1,mat2):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a','b','c'],
       ['d','b','a'] ]
m2 = [ ['f','b'],
       ['g','h'] ]

res = [ ['f','b','a','b','c'],
        ['g','h','d','b','a'] ]

stitch_left_mod(m1, m2)
assert m1 == res
```

transpose_1

Let's see how to transpose a matrix *in-place*. The transpose M^T of a matrix M is defined as

$M^T[i][j] = M[j][i]$

The definition is simple yet implementation might be tricky. If you're not careful, you could easily end up swapping the values twice and get the same original matrix. To prevent this, iterate only the upper triangular part of the matrix and remember `range` funciton can also have a start index:

[52]:
```
list(range(3,7))
```

[52]:
```
[3, 4, 5, 6]
```

Also, make sure you know how to swap just two values by solving first this very simple exercise - also check the result in Python Tutor

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[53]:
```
x = 3
y = 7

# write here code for swapping x and y (don't directly use the constants 3 and 7!)

k = x
x = y
y = k

#jupman.pytut()
```

</div>

[53]:
```
x = 3
y = 7
```

(continues on next page)

(continued from previous page)

```
# write here code for swapping x and y (don't directly use the constants 3 and 7!)
```

Going back to the transpose, for now we will consider only an nxn matrix. To make sure we actually get an nxn matrix, we will validate the input as before.

IV COMMANDMENT[286] (adapted for matrices): You shall never ever reassign function parameters

```
def myfun(M):

    # M is a parameter, so you shall *not* do any of such evil:

    M = [
            [6661,6662],
            [6663,6664 ]
        ]


    # For the sole case of composite parameters like lists (or lists of lists ..)
    # you can write stuff like this IF AND ONLY IF the function specification
    # requires you to modify the parameter internal elements (i.e. transposing _in-
↪place_):

    M[0][1] =  6663
```

⊛⊛⊛ **EXERCISE** If you read *all* the above, you can now proceed implementing the `transpose_1` function, which MODIFIES the given `nxn` matrix `mat` by transposing it *in-place*.

- If the matrix is not `nxn`, raises a `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[54]:
```
def transpose_1(mat):

    if len(mat) != len(mat[0]):
        raise ValueError("Matrix should be nxn, found instead %s x %s" % (len(mat),␣
↪len(mat[0])))
    for i in range(len(mat)):
        for j in range(i+1,len(mat[i])):
            el = mat[i][j]
            mat[i][j] = mat[j][i]
            mat[j][i] = el




# let's try wrong matrix dimensions:
try:
    transpose_1([ [3,5] ])
    raise Exception("SHOULD HAVE FAILED !")
```

(continues on next page)

[286] https://en.softpython.org/commandments.html#IV-COMMANDMENT

(continued from previous page)

```
except ValueError:
    pass

m1 = [ ['a'] ]

transpose_1(m1)
assert m1 == [ ['a'] ]

m2 = [ ['a','b'],
       ['c','d'] ]

transpose_1(m2)
assert m2 == [ ['a','c'],
               ['b','d'] ]
```

</div>

[54]:
```
def transpose_1(mat):
    raise Exception('TODO IMPLEMENT ME !')



# let's try wrong matrix dimensions:
try:
    transpose_1([ [3,5] ])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    pass

m1 = [ ['a'] ]

transpose_1(m1)
assert m1 == [ ['a'] ]

m2 = [ ['a','b'],
       ['c','d'] ]

transpose_1(m2)
assert m2 == [ ['a','c'],
               ['b','d'] ]
```

transpose_2

⊛⊛ Now let's try to transpose a generic nxm matrix. This time for simplicity we will return a whole new matrix.

RETURN a NEW mxn matrix which is the transpose of the given `nxm` matrix `mat` as list of lists.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[55]:
```
def transpose_2(mat):

    n = len(mat)
    m = len(mat[0])
    ret = [[0]*n for i in range(m)]
    for i in range(n):
```

(continues on next page)

(continued from previous page)

```
        for j in range(m):
            ret[j][i] = mat[i][j]
    return ret


m1 = [ ['a'] ]

r1 = transpose_2(m1)

assert  r1 == [ ['a'] ]
r1[0][0] = 'z'
assert m1[0][0] == 'a'

m2 = [ ['a','b','c'],
       ['d','e','f'] ]

assert transpose_2(m2) == [ ['a','d'],
                            ['b','e'],
                            ['c','f'] ]
```

</div>

[55]:

```
def transpose_2(mat):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ ['a'] ]

r1 = transpose_2(m1)

assert  r1 == [ ['a'] ]
r1[0][0] = 'z'
assert m1[0][0] == 'a'

m2 = [ ['a','b','c'],
       ['d','e','f'] ]

assert transpose_2(m2) == [ ['a','d'],
                            ['b','e'],
                            ['c','f'] ]
```

lab

⊛⊛⊛ If you're a teacher that often see new students, you have this problem: if two students who are friends sit side by side they can start chatting way too much. To keep them quiet, you want to somehow randomize student displacement by following this algorithm:

1. first sort the students alphabetically
2. then sorted students progressively sit at the available chairs one by one, first filling the first row, then the second, till the end.

Now implement the algorithm.

INPUT:

- students: a list of strings of length <= n*m
- chairs: an nxm matrix as list of lists filled with None values (empty chairs)

OUTPUT: MODIFIES BOTH students and chairs inputs, without returning anything

If students are more than available chairs, raises ValueError

Example:

```
ss =  ['b', 'd', 'e', 'g', 'c', 'a', 'h', 'f' ]

mat = [
            [None, None, None],
            [None, None, None],
            [None, None, None],
            [None, None, None]
         ]

lab(ss,  mat)

# after execution, mat should result changed to this:

assert mat == [
                ['a',  'b', 'c'],
                ['d',  'e', 'f'],
                ['g',  'h',  None],
                [None, None, None],
             ]
# after execution, input ss should now be ordered:

assert ss == ['a','b','c','d','e','f','g','f']
```

For more examples, see tests

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[56]:
```
def lab(students, chairs):


    n = len(chairs)
    m = len(chairs[0])

    if len(students) > n*m:
        raise ValueError("There are more students than chairs ! Students = %s, chairs␣
↪= %sx%s" % (len(students), n, m))

    i = 0
    j = 0
    students.sort()
    for s in students:
        chairs[i][j] = s

        if j == m - 1:
            j = 0
            i += 1
        else:
            j += 1



try:
```

(continues on next page)

(continued from previous page)

```
    lab(['a','b'], [[None]])
    raise Exception("TEST FAILED: Should have failed before with a ValueError!")
except ValueError:
    "Test passed"

try:
    lab(['a','b','c'], [[None,None]])
    raise Exception("TEST FAILED: Should have failed before with a ValueError!")
except ValueError:
    "Test passed"


m0 = [ [None] ]

r0 = lab([],m0)
assert m0 == [ [None] ]
assert r0 == None  # function is not meant to return anything (so returns None by␣
↪default)


m1 = [ [None] ]
r1 = lab(['a'], m1)

assert m1 == [ ['a'] ]
assert r1 == None  # function is not meant to return anything (so returns None by␣
↪default)

m2 = [ [None, None] ]
lab(['a'], m2)  # 1 student 2 chairs in one row

assert m2 == [ ['a', None] ]


m3 = [ [None],
       [None] ]
lab(['a'], m3) # 1 student 2 chairs in one column
assert m3 == [ ['a'],
               [None] ]

ss4 = ['b', 'a']
m4 = [ [None, None] ]
lab(ss4, m4)  # 2 students 2 chairs in one row

assert m4 == [ ['a','b'] ]

assert ss4 == ['a', 'b']  # also modified input list as required by function text

m5 = [ [None, None],
       [None, None] ]
lab(['b', 'c', 'a'], m5)  # 3 students 2x2 chairs

assert m5 == [ ['a','b'],
               ['c', None] ]

m6 = [ [None, None],
       [None, None] ]
lab(['b', 'd', 'c', 'a'], m6)  # 4 students 2x2 chairs
```

(continues on next page)

(continued from previous page)

```
assert m6 == [ ['a','b'],
               ['c','d'] ]

m7 = [ [None, None, None],
       [None, None, None] ]
lab(['b', 'd', 'e', 'c', 'a'], m7)  # 5 students 3x2 chairs

assert m7 == [ ['a','b','c'],
               ['d','e',None] ]

ss8 = ['b', 'd', 'e', 'g', 'c', 'a', 'h', 'f' ]
m8 = [ [None, None, None],
       [None, None, None],
       [None, None, None],
       [None, None, None] ]
lab(ss8, m8)  # 8 students 3x4 chairs

assert m8 == [ ['a',  'b',  'c'],
               ['d',  'e',  'f'],
               ['g',  'h',  None],
               [None, None, None] ]

assert ss8 == ['a','b','c','d','e','f','g','h']
```

</div>

[56]:
```
def lab(students, chairs):
    raise Exception('TODO IMPLEMENT ME !')


try:
    lab(['a','b'], [[None]])
    raise Exception("TEST FAILED: Should have failed before with a ValueError!")
except ValueError:
    "Test passed"

try:
    lab(['a','b','c'], [[None,None]])
    raise Exception("TEST FAILED: Should have failed before with a ValueError!")
except ValueError:
    "Test passed"


m0 = [ [None] ]

r0 = lab([],m0)
assert m0 == [ [None] ]
assert r0 == None  # function is not meant to return anything (so returns None by␣
↪default)


m1 = [ [None] ]
r1 = lab(['a'], m1)

assert m1 == [ ['a'] ]
assert r1 == None  # function is not meant to return anything (so returns None by␣
↪default)
```

(continues on next page)

(continued from previous page)

```
m2 = [ [None, None] ]
lab(['a'], m2)  # 1 student 2 chairs in one row

assert m2 == [ ['a', None] ]


m3 = [ [None],
       [None] ]
lab(['a'], m3) # 1 student 2 chairs in one column
assert m3 == [ ['a'],
               [None] ]

ss4 = ['b', 'a']
m4 = [ [None, None] ]
lab(ss4, m4)  # 2 students 2 chairs in one row

assert m4 == [ ['a','b'] ]

assert ss4 == ['a', 'b']  # also modified input list as required by function text

m5 = [ [None, None],
       [None, None] ]
lab(['b', 'c', 'a'], m5)  # 3 students 2x2 chairs

assert m5 == [ ['a','b'],
               ['c', None] ]

m6 = [ [None, None],
       [None, None] ]
lab(['b', 'd', 'c', 'a'], m6)  # 4 students 2x2 chairs

assert m6 == [ ['a','b'],
               ['c','d'] ]

m7 = [ [None, None, None],
       [None, None, None] ]
lab(['b', 'd', 'e', 'c', 'a'], m7)  # 5 students 3x2 chairs

assert m7 == [ ['a','b','c'],
               ['d','e',None] ]

ss8 = ['b', 'd', 'e', 'g', 'c', 'a', 'h', 'f' ]
m8 = [ [None, None, None],
       [None, None, None],
       [None, None, None],
       [None, None, None] ]
lab(ss8, m8)  # 8 students 3x4 chairs

assert m8 == [ ['a',  'b',  'c'],
               ['d',  'e',  'f'],
               ['g',  'h',  None],
               [None, None, None] ]

assert ss8 == ['a','b','c','d','e','f','g','h']
```

dump

The multinational ToxiCorp wants to hire you for devising an automated truck driver which will deposit highly contaminated waste in the illegal dumps they own worldwide. You find it ethically questionable, but they pay well, so you accept.

A dump is modelled as a rectangular region of dimensions `nrow` and `ncol`, implemented as a list of lists matrix. Every cell `i, j` contains the tons of waste present, and can contain *at most* `7` tons of waste.

The dumpster truck will transport `q` tons of waste, and try to fill the dump by depositing waste in the first row, filling each cell up to 7 tons. When the first row is filled, it will proceed to the second one *from the left* , then to the third one again *from the left* until there is no waste to dispose of.

Function `dump(m, q)` takes as input the dump `mat` and the number of tons `q` to dispose of, and RETURN a NEW list representing a plan with the sequence of tons to dispose. If waste to dispose exceeds dump capacity, raises `ValueError`.

NOTE: the function does **not** modify the matrix

Example:

```
m = [
        [5,4,6],
        [4,7,1],
        [3,2,6],
        [3,6,2],
]

dump(m, 22)

[2, 3, 1, 3, 0, 6, 4, 3]
```

For first row we dispose of 2,3,1 tons in three cells, for second row we dispose of 3,0,6 tons in three cells, for third row we only dispose 4, 3 tons in two cells as limit q=22 is reached.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[57]:
```python
def dump(mat, q):

    rem = q
    ret = []

    for riga in mat:
        for j in range(len(riga)):
            cellfill = 7 - riga[j]
            unload = min(cellfill, rem)
            rem -= unload

            if rem > 0:
                ret.append(unload)
            else:
                if unload > 0:
                    ret.append(unload)
                return ret

    if rem > 0:
        raise ValueError("Couldn't fill the dump, %s tons remain!")
```

(continues on next page)

(continued from previous page)

```
m1 = [ [5] ]

assert dump(m1,0) == []  # nothing to dump

m2 = [ [4] ]

assert dump(m2,2) == [2]

m3 = [ [5,4] ]

assert dump(m3,3) == [2, 1]


m3 = [ [5,7,3] ]

assert dump(m3,3) == [2, 0, 1]


m5 = [ [2,5],    # 5 2
       [4,3] ]   # 3 1

assert dump(m5,11) == [5,2,3,1]

                  # tons to dump in each cell
m6 = [ [5,4,6],   # 2 3 1
       [4,7,1],   # 3 0 6
       [3,2,6],   # 4 3 0
       [3,6,2] ]  # 0 0 0

assert dump(m6, 22) == [2,3,1,3,0,6,4,3]

try:
    dump ([[5]], 10)
    raise Exception("Should have failed !")
except ValueError:
    pass
```

</div>

[57]:
```
def dump(mat, q):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ [5] ]

assert dump(m1,0) == []  # nothing to dump

m2 = [ [4] ]

assert dump(m2,2) == [2]

m3 = [ [5,4] ]

assert dump(m3,3) == [2, 1]


m3 = [ [5,7,3] ]
```

(continues on next page)

(continued from previous page)

```
assert dump(m3,3) == [2, 0, 1]


m5 = [ [2,5],    # 5 2
       [4,3] ]   # 3 1

assert dump(m5,11) == [5,2,3,1]

                   # tons to dump in each cell
m6 = [ [5,4,6],    # 2 3 1
       [4,7,1],    # 3 0 6
       [3,2,6],    # 4 3 0
       [3,6,2] ]   # 0 0 0

assert dump(m6, 22) == [2,3,1,3,0,6,4,3]

try:
    dump ([[5]], 10)
    raise Exception("Should have failed !")
except ValueError:
    pass
```

matrix multiplication

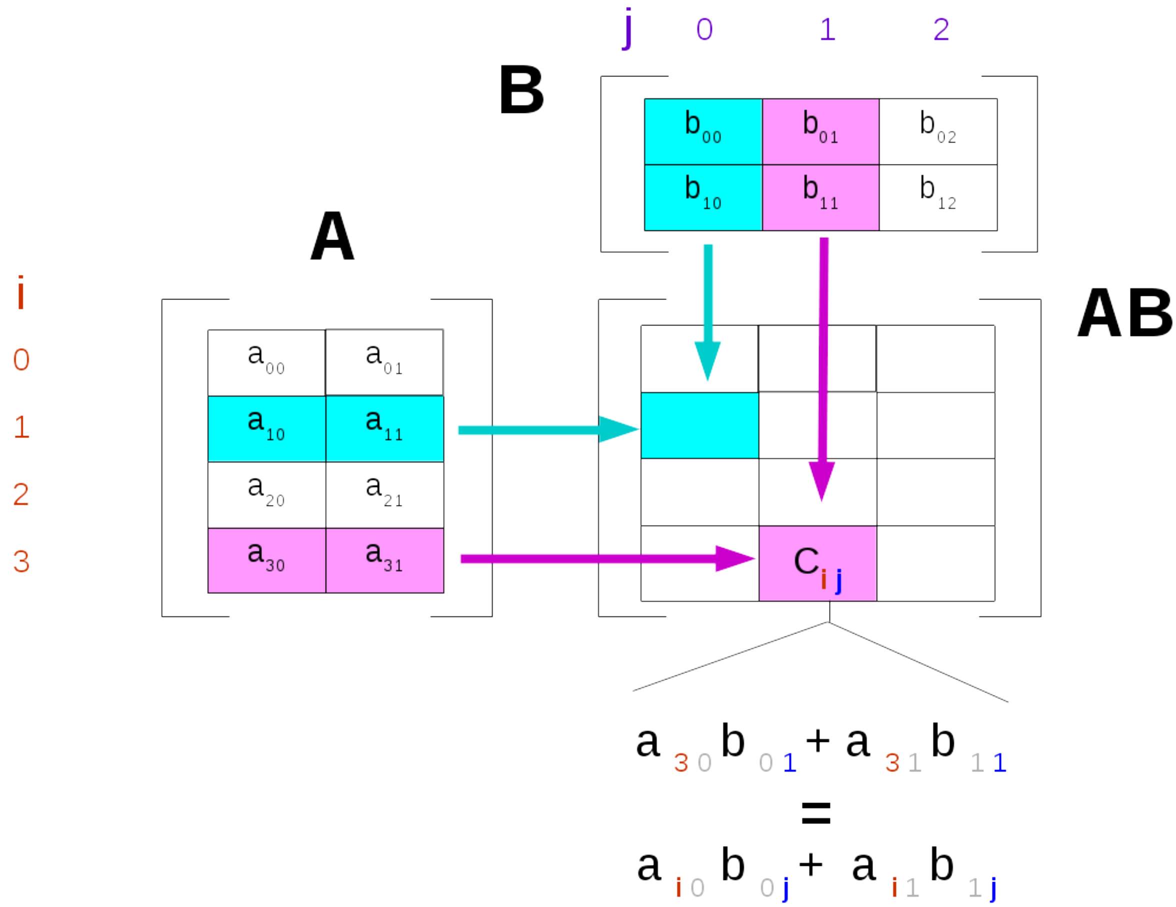
Have a look at matrix multiplication definition[287] on Wikipedia and try to implement it in the following function.

Basically, given nxm matrix A and mxp matrix B you need to output an nxp matrix C calculating the entries c_{ij} with the formula

$c_{ij} = a_{i1}b_{1j} + \cdots + a_{im}b_{mj} = \sum_{k=1}^{m} a_{ik}b_{kj}$

You need to fill all the nxp cells of C, so sure enough to fill a rectangle you need two `for`s. Do you also need another `for` ? Help yourself with the following visualization.

[287] https://en.wikipedia.org/w/index.php?title=Matrix_multiplication&section=2#Definition



⊛⊛⊛ **EXERCISE**: Given matrices `n x m mata` and `m x p matb`, RETURN a NEW `n x p` matrix which is the result of the multiplication of `mata` by `matb`.

- If `mata` has column number different from `matb` row number, raises a `ValueError`.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[58]:
```python
def mul(mata, matb):

    n = len(mata)
    m = len(mata[0])
    p = len(matb[0])
    if m != len(matb):
        raise ValueError("mat1 column number %s must be equal to mat2 row number %s !
↪" % (m, len(matb)))
    ret = [[0]*p for i in range(n)]
    for i in range(n):
        for j in range(p):
            ret[i][j] = 0
            for k in range(m):
                ret[i][j] += mata[i][k] * matb[k][j]
    return ret


# TEST START - DO NOT TOUCH!
```

(continues on next page)

(continued from previous page)

```
# if you wrote the whole code correct, and execute the cell, Python shouldn't raise␣
↪`AssertionError`

# let's try wrong matrix dimensions:
try:
    mul([[3,5]], [[7]])
    raise Exception("SHOULD HAVE FAILED!")
except ValueError:
    "passed test"

ma1 = [ [3] ]
mb1 = [ [5] ]
r1 = mul(ma1,mb1)
assert r1 == [ [15] ]

ma2 = [ [3],
        [5] ]

mb2 = [ [2,6] ]

r2 = mul(ma2,mb2)

assert r2 == [ [3*2, 3*6],
               [5*2, 5*6] ]

ma3 = [ [3,5]  ]

mb3 = [  [2],
         [6] ]

r3 = mul(ma3,mb3)

assert r3 == [ [3*2 + 5*6] ]

ma4 = [ [3,5],
        [7,1],
        [9,4] ]

mb4 = [ [4,1,5,7],
        [8,5,2,7] ]
r4 = mul(ma4,mb4)

assert r4 == [ [52, 28, 25, 56],
               [36, 12, 37, 56],
               [68, 29, 53, 91] ]
```

</div>

[58]:

```
def mul(mata, matb):
    raise Exception('TODO IMPLEMENT ME !')

# TEST START - DO NOT TOUCH!
# if you wrote the whole code correct, and execute the cell, Python shouldn't raise␣
↪`AssertionError`

# let's try wrong matrix dimensions:
try:
```

(continues on next page)

(continued from previous page)

```
    mul([[3,5]], [[7]])
    raise Exception("SHOULD HAVE FAILED!")
except ValueError:
    "passed test"

ma1 = [ [3] ]
mb1 = [ [5] ]
r1 = mul(ma1,mb1)
assert r1 == [ [15] ]

ma2 = [ [3],
        [5] ]

mb2 = [ [2,6] ]

r2 = mul(ma2,mb2)

assert r2 == [ [3*2, 3*6],
               [5*2, 5*6] ]

ma3 = [ [3,5]  ]

mb3 = [  [2],
         [6] ]

r3 = mul(ma3,mb3)

assert r3 == [ [3*2 + 5*6] ]

ma4 = [ [3,5],
        [7,1],
        [9,4] ]

mb4 = [ [4,1,5,7],
        [8,5,2,7] ]
r4 = mul(ma4,mb4)

assert r4 == [ [52, 28, 25, 56],
               [36, 12, 37, 56],
               [68, 29, 53, 91] ]
```

check_nqueen

⊛⊛⊛⊛ This is a hard problem but don't worry, exam exercises will be simpler!

You have an nxn matrix of booleans representing a chessboard where True means there is a queen in a cell,and False there is nothing.

For the sake of visualization, we can represent a configurations using `o` to mean `False` and letters like 'A' and 'B' are queens. Contrary to what we've done so far, for later convenience we show the matrix with the `j` going from bottom to top.

Let's see an example. In this case A and B can not attack each other, so the algorithm would return `True`:

```
    7  ......B.
    6  ........
```

(continues on next page)

(continued from previous page)

```
    5  ........
    4  ........
    3  ....A...
    2  ........
    1  ........
    0  ........
    i
     j 01234567


Let's see why by evidencing A attack lines ..

    7  \...|.B.
    6  .\..|../
    5  ..\.|./.
    4  ...\|/..
    3  ----A---
    2  .../|\..
    1  ../.|.\.
    0  ./..|..\
    i
     j 01234567


... and B attack lines:

    7  ------B-
    6  ...../|\
    5  ..../.|.
    4  .../..|.
    3  ../.A.|.
    2  ./....|.
    1  /.....|.
    0  ......|.
    i
     j 01234567
```

In this other case the algorithm would return False as `A` and `B` can attack each other:

```
7  \./.|...
6  -B--|--/
5  /|\.|./.
4  .|.\|/..
3  ----A---
2  .|./|\..
1  .|/.|.\.
0  ./..|..\
i
 j 01234567
```

In your algorithm, first you need to scan for queens. When you find one (and for each one of them !), you need to check if it can hit some other queen. Let's see how:

In this 7x7 table we have only one queen A, with at position `i=1` and `j=4`

```
6  ....|..
5  \...|..
```

(continues on next page)

(continued from previous page)

```
4   .\..|..
3   ..\.|./
2   ...\|/.
1   ----A--
0   .../|\.
i
 j 0123456
```

To completely understand the range of the queen and how to calculate the diagonals, it is convenient to visually extend the table like so to have the diagonals hit the vertical axis. Notice we also added letters `y` and `x`

NOTE: in the algorithm you **do not** need to extend the matrix !

```
 y
 6   ....|....
 5   \...|.../
 4   .\..|../.
 3   ..\.|./..
 2   ...\|/...
 1   ----A----
 0   .../|\...
-1   ../.|.\..
-2   ./..|..\.
-3   /...|...\
 i
  j 01234567 x
```

We see that the top-left to bottom-right diagonal hits the vertical axis at `y = 5` and the bottom-left to top-right diagonal hits the axis at `y = -3`. You should use this info to calculate the line equations.

Now you should have all the necessary hints to proceed with the implementation.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[59]:

```python
def check_nqueen(mat):
    """ Takes an nxn matrix of booleans representing a chessboard where True means␣
↪there is a queen in a cell,
        and False there is nothing. RETURN True if no queen can attack any other one,␣
↪False otherwise

    """


    # bottom-left to top-right line equation
    # y = x - 3
    # -3 = -j + i
    # y = x -j + i

    # top-left to bottom-right line equation
    # y = x + 5
    # 5 = j + i
    # y = x + j + i
```

(continues on next page)

(continued from previous page)

```
    n = len(mat)
    for i in range(n):
        for j in range(n):
            if mat[i][j]:  # queen is found at i,j
                for y in range(n):                # vertical scan
                    if y != i and mat[y][j]:
                        return False
                for x in range(n):                # horizontal scan
                    if x != j and mat[i][x]:
                        return False
                for x in range(n):
                    y = x + j + i        # top-left to bottom-right
                    if y >= 0 and y < n and y != i and x != j and mat[y][x]:
                        return False
                    y = x - j + i        # bottom-left to top-right
                    if y >= 0 and y < n and y != i and x != j and mat[y][x]:
                        return False

    return True


assert check_nqueen([ [True] ])
assert check_nqueen([ [True, True],
                      [False, False] ]) == False

assert check_nqueen([ [True, False],
                      [False, True] ]) == False

assert check_nqueen([ [True, False],
                      [True, False] ]) == False

assert check_nqueen([ [True,  False, False],
                      [False, False, True],
                      [False, False, False] ]) == True

assert check_nqueen([ [True,  False, False],
                      [False, False, False],
                      [False, False, True] ]) == False


assert check_nqueen([ [False, True,  False],
                      [False, False, False],
                      [False, False, True] ]) == True

assert check_nqueen([ [False, True,  False],
                      [False, True, False],
                      [False, False, True] ]) == False
```

</div>

[59]:

```
def check_nqueen(mat):
    """ Takes an nxn matrix of booleans representing a chessboard where True means␣
↪there is a queen in a cell,
        and False there is nothing. RETURN True if no queen can attack any other one,␣
↪False otherwise
```

(continues on next page)

(continued from previous page)

```
    """
    raise Exception('TODO IMPLEMENT ME !')

assert check_nqueen([ [True] ])
assert check_nqueen([ [True, True],
                      [False, False] ]) == False

assert check_nqueen([ [True, False],
                      [False, True] ]) == False

assert check_nqueen([ [True, False],
                      [True, False] ]) == False

assert check_nqueen([ [True,  False, False],
                      [False, False, True],
                      [False, False, False] ]) == True

assert check_nqueen([ [True,  False, False],
                      [False, False, False],
                      [False, False, True] ]) == False


assert check_nqueen([ [False, True,  False],
                      [False, False, False],
                      [False, False, True] ]) == True

assert check_nqueen([ [False, True,  False],
                      [False, True, False],
                      [False, False, True] ]) == False
```

[ ]:

```

```

5.19 Error handling and testing solutions

5.19.1 Download exercises zip

Browse files online[288]

5.19.2 Introduction

In this notebook we will try to understand what our program should do when it encounters unforeseen situations, and how to test the code we write.

For some strange reason, many people believe that computer programs do not need much error handling nor testing. Just to make a simple comparison, would you ever drive a car that did not undergo scrupolous checks? We wouldn't.

[288] https://github.com/DavidLeoni/softpython-en/tree/master/errors-and-testing

What to do

- unzip exercises in a folder, you should get something like this:

```
errors-and-testing
    errors-and-testing.ipynb
    errors-and-testing-sol.ipynb
    jupman.py
```

WARNING 1: to correctly visualize the notebook, it MUST be in an unzipped folder !

- open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `strings/strings.ipynb`

WARNING 2: DO NOT use the *Upload* button in Jupyter, instead navigate to the unzipped folder while in Jupyter browser!

- Go on reading that notebook, and follow instuctions inside.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

5.19.3 Unforeseen situations

It is evening, there is to party for a birthday and they asked you to make a pie. You need the following steps:

1. take milk
2. take sugar
3. take flour
4. mix
5. heat in the oven

You take the milk, the sugar, but then you discover there is no flour. It is evening, and there aren't open shops. Obviously, it makes no sense to proceed to point 4 with the mixture, and you have to give up on the pie, telling the guest of honor the problem. You can only hope she/he decides for some alternative.

Translating everything in Python terms, we can ask ourselves if during the function execution, when we find an unforeseen situation, is it possible to:

1. **interrupt** the execution flow of the program
2. **signal** to whoever called the function that a problem has occurred
3. **allow to manage** the problem to whoever called the function

The answer is yes, you can do it with the mechanism of **exceptions** (`Exception`)

make_problematic_pie

Let's see how we can represent the above problem in Python. A basic version might be the following:

```
[2]: def make_problematic_pie(milk, sugar, flour):
    """ Suppose you need 1.3 kg for the milk, 0.2kg for the sugar and 1.0kg for the␣
↪flour

        - takes as parameters the quantities we have in the sideboard
    """

    if milk > 1.3:
        print("take milk")
    else:
        print("Don't have enough milk !")

    if sugar > 0.2:
        print("take sugar")
    else:
        print("Don't have enough sugar!")

    if flour > 1.0:
        print("take flour")
    else:
        print("Don't have enough flour !")

    print("Mix")
    print("Heat")
    print("I made the pie!")


make_problematic_pie(5,1,0.3)  # not enough flour ...

print("Party")
```

```
take milk
take sugar
Don't have enough flour !
Mix
Heat
I made the pie!
Party
```

QUESTION: this above version has a serious problem. Can you spot it ??

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: the program above is partying even when we do not have enough ingredients !

</div>

5.19.4 Check with the return

EXERCISE: We could correct the problems of the above pie by adding `return` commands. Implement the following function.

> **WARNING: DO NOT move the `print("Party")` inside the function**
>
> The exercise goal is keeping it outside, so to use the value returned by `make_pie` for deciding whether to party or not.

If you have any doubts on functions with return values, check Chapter 6 of Think Python[289]

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[3]:
```python
def make_pie(milk, sugar, flour):
    """  - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

         - takes as parameters the quantities we have in the sideboard
         IMPROVE WITH return COMMAND: RETURN True if the pie is doable,
                                             False otherwise

         *OUTSIDE* USE THE VALUE RETURNED TO PARTY OR NOT

    """
    # implement here the function

    if milk > 1.3:
        print("take milk")
        # return True  # NO, it would finish right here
    else:
        print("Don't have enough milk !")
        return False

    if sugar > 0.2:
        print("take sugar")
    else:
        print("Don't have enouch sugar !")
        return False

    if flour > 1.0:
        print("take flour")
    else:
        print("Don't have enough flour !")
        return False

    print("Mix")
    print("Heat")
    print("I made the pie !")
    return True



# now write here the function call, make_pie(5,1,0.3)
# using the result to declare whether it is possible or not to party :-(
```

(continues on next page)

[289] http://greenteapress.com/thinkpython2/html/thinkpython2007.html

(continued from previous page)

```
made_pie = make_pie(5,1,0.3)

if made_pie == True:
    print("Party")
else:
    print("No party !")
```

```
take milk
take sugar
Don't have enough flour !
No party !
```

</div>

[3]:
```
def make_pie(milk, sugar, flour):
    """  - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

         - takes as parameters the quantities we have in the sideboard
         IMPROVE WITH return COMMAND: RETURN True if the pie is doable,
                                          False otherwise

         *OUTSIDE* USE THE VALUE RETURNED TO PARTY OR NOT

    """
    # implement here the function



# now write here the function call, make_pie(5,1,0.3)
# using the result to declare whether it is possible or not to party :-(
```

```
take milk
take sugar
Don't have enough flour !
No party !
```

5.19.5 Exceptions

Real Python - Python Exceptions: an Introduction[290]

Using `return` we improved the previous function, but remains a problem: the responsability to understand whether or not the pie is properly made is given to the caller of the function, who has to take the returned value and decide upon that whether to party or not. A careless programmer might forget to do the check and party even with an ill-formed pie.

So we ask ourselves: is it possible to stop the execution not just of the function, but of the whole program when we find an unforeseen situation?

To improve on our previous attempt, we can use the *exceptions*. To tell Python to **interrupt** the program execution in a given point, we can insert the instruction `raise` like this:

[290] https://realpython.com/python-exceptions/

```python
raise Exception()
```

If we want, we can also write a message to help programmers (who could be ourselves …) to understand the problem origin. In our case it could be a message like this:

```python
raise Exception("Don't have enough flour !")
```

Note: in professional programs, the exception messages are intended for programmers, verbose, and tipically end up hidden in system logs. To final users you should only show short messages which are understanble by a non-technical public. At most, you can add an error code which the user might give to the technician for diagnosing the problem.

EXERCISE: Try to rewrite the function above by substituting the rows containing `return` with `raise Exception()`:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[4]:
```python
def make_exceptional_pie(milk, sugar, flour):
    """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

        - takes as parameters the quantities we have in the sideboard

        - if there are missing ingredients, raises Exception

    """
    # implement function


    if milk > 1.3:
        print("take milk")
    else:
        raise Exception("Don't have enough milk !")
    if sugar > 0.2:
        print("take sugar")
    else:
        raise Exception("Don't have enough sugar!")
    if flour > 1.0:
        print("take flour")
    else:
        raise Exception("Don't have enough flour!")
    print("Mix")
    print("Heat")
    print("I made the pie !")
```

</div>

[4]:
```python
def make_exceptional_pie(milk, sugar, flour):
    """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

        - takes as parameters the quantities we have in the sideboard

        - if there are missing ingredients, raises Exception

    """
    # implement function
```

Once implemented, by writing

```
make_exceptional_pie(5,1,0.3)
print("Party")
```

you should see the following (note how "Party" is *not* printed):

```
take milk
take sugar

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-10-02c123f44f31> in <module>()
----> 1 make_exceptional_pie(5,1,0.3)
      2
      3 print("Party")

<ipython-input-9-030239f08ca5> in make_exceptional_pie(milk, sugar, flour)
     18         print("take flour")
     19     else:
---> 20         raise Exception("Don't have enough flour !")
     21     print("Mix")
     22     print("Heat")

Exception: Don't have enough flour !
```

We see the program got interrupted before arriving to mix step (inside the function), and it didn't even arrived to party (which is outside the function). Let's try now to call the function with enough ingredients in the sideboard:

[5]:
```
make_exceptional_pie(5,1,20)
print("Party")
```

```
take milk
take sugar
take flour
Mix
Heat
I made the pie !
Party
```

Manage exceptions

Instead of brutally interrupting the program when problems are spotted, we might want to try some alternative (like go buying some ice cream). We could use some `try except` blocks like this:

[6]:
```
try:
    make_exceptional_pie(5,1,0.3)
    print("Party")
except:
    print("Can't make the pie, what about going out for an ice cream?")
```

```
take milk
take sugar
Can't make the pie, what about going out for an ice cream?
```

If you note, the execution jumped the `print("Party"` but no exception has been printed, and the execution passed to the row right after the `except`

Particular exceptions

Until know we used a generic `Exception`, but, if you will, you can use more specific exceptions to better signal the nature of the error. For example, when you implement a function, since checking the input values for correctness is very frequent, Python gives you an exception called `ValueError`. If you use it instead of `Exception`, you allow the function caller to intercept only that particular error type.

If the function raises an error which is not intercepted in the catch, the program will halt.

[7]:

```python
def make_exceptional_pie_2(milk, sugar, flour):
    """ - suppose we need 1.3 kg for milk, 0.2kg for sugar and 1.0kg for flour

        - takes as parameters the quantities we have in the sideboard

        - if there are missing ingredients, raises Exception
    """

    if milk > 1.3:
        print("take milk")
    else:
        raise ValueError("Don't have enough milk !")
    if sugar > 0.2:
        print("take sugar")
    else:
        raise ValueError("Don't have enough sugar!")
    if flour > 1.0:
        print("take flour")
    else:
        raise ValueError("Don't have enough flour!")
    print("Mix")
    print("Heat")
    print("I made the pie !")

try:
    make_exceptional_pie_2(5,1,0.3)
    print("Party")
except ValueError:
    print()
    print("There must be a problem with the ingredients!")
    print("Let's try asking neighbors !")
    print("We're lucky, they gave us some flour, let's try again!")
    print("")
    make_exceptional_pie_2(5,1,4)
    print("Party")
except:  # manages all exceptions
    print("Guys, something bad happened, don't know what to do. Better to go out and␣
↪take an ice-cream !")
```

```
take milk
take sugar

There must be a problem with the ingredients!
Let's try asking neighbors !
We're lucky, they gave us some flour, let's try again!

take milk
```

(continues on next page)

(continued from previous page)

```
take sugar
take flour
Mix
Heat
I made the pie !
Party
```

For more explanations about `try catch`, you can see Real Python - Python Exceptions: an Introduction[291]

5.19.6 assert

They asked you to develop a program to control a nuclear reactor. The reactor produces a lot of energy, but requires at least 20 meters of water to cool down, and your program needs to regulate the water level. Without enough water, you risk a meltdown. You do not feel exactly up to the job, and start sweating.

Nervously, you write the code. You check and recheck the code - everything looks fine.

On inauguration day, the reactor is turned on. Unexpectedly, the water level goes down to 5 meters, and an uncontrolled chain reaction occurs. Plutoniom fireworks follow.

Could we have avoided all of this? We often believe everything is good but then for some reason we find variables with unexpected values. The wrong program described above might have been written like so:

[8]:
```
# we need water to cool our reactor

water_level = 40 #  seems ok

print("water level: ", water_level)


# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5  # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)

# a lot of code

# a lot of code

# a lot of code

# a lot of code

# after a lot of code we might not know if there are the proper conditions so that␣
↪everything works allright

print("turn on nuclear reactor")
```

[291] https://realpython.com/python-exceptions/

```
water level:  40
WARNING: water level low!  5
turn on nuclear reactor
```

How could we improve it? Let's look at the `assert` command, which must be written by following it with a boolean condition.

`assert True` does absolutely nothing:

[9]:
```python
print("before")
assert True
print("after")
```

```
before
after
```

Instead, `assert False` completely blocks program execution, by launching an exception of type `AssertionError` (Note how `"after"` is not printed):

```python
print("before")
assert False
print("after")
```

```
before
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-7-a871fdc9ebee> in <module>()
----> 1 assert False

AssertionError:
```

To improve the previous program, we might use `assert` like this:

```python
# we need water to cool our reactor

water_level = 40   # seems ok

print("water level: ", water_level)


# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5  # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)

# a lot of code

# a lot of code

# a lot of code
```

(continues on next page)

(continued from previous page)

```
# a lot of code

# after a lot of code we might not know if there are the proper conditions so that
# everything works allright so before doing critical things, it is always a good idea
# to perform a check ! if asserts fail (that is, the boolean expression is False),
# the execution suddenly stops

assert water_level >= 20

print("turn on nuclear reactor")
```

```
water level:  40
WARNING: water level low!  5

---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-3-d553a90d4f64> in <module>
     31 # the execution suddenly stops
     32
---> 33 assert water_level >= 20
     34
     35 print("turn on nuclear reactor")

AssertionError:
```

When to use assert?

The case above is willingly exagerated, but shows how a check more sometimes prevents disasters.

Asserts are a quick way to do checks, so much so that Python even allows to ignore them during execution to improve the performance (calling `python` with the `-O` parameter like in `python -O my_file.py`).

But if performance are not a problem (like in the reactor above), it's more convenient to rewrite the program using an `if` and explicitly raising an `Exception`:

```
# we need water to cool our reactor

water_level = 40   # seems ok

print("water level: ", water_level)


# a lot of code

# a lot of code

# a lot of code

# a lot of code

water_level = 5  # forgot somewhere this bad row !

print("WARNING: water level low! ", water_level)
```

(continues on next page)

(continued from previous page)

```
# a lot of code

# a lot of code

# a lot of code

# a lot of code

# after a lot of code we might not know if there are the proper conditions so
# that everything works all right. So before doing critical things, it is always
# a good idea to perform a check !

if water_level < 20:
    raise Exception("Water level too low !")  # execution stops here

print("turn on nuclear reactor")
```

```
water level:  40
WARNING: water level low!  5

---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-30-4840536c3388> in <module>
     30
     31 if water_level < 20:
---> 32     raise Exception("Water level too low !")  # execution stops here
     33
     34 print("turn on nuclear reactor")

Exception: Water level too low !
```

Note how the reactor was *not* turned on.

5.19.7 Testing

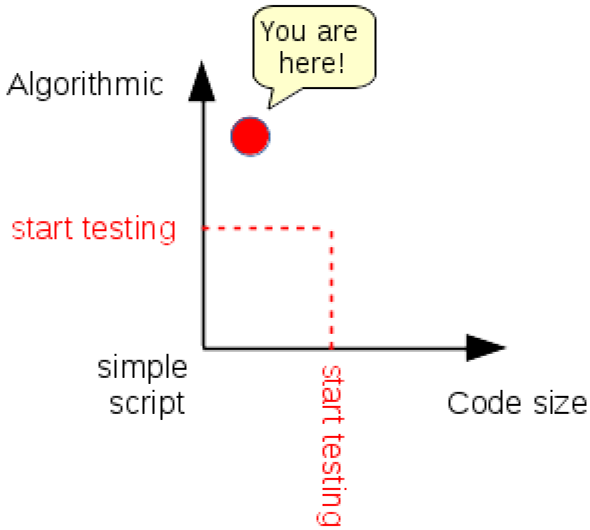
- If it seems to work, then it actually works? *Probably not.*
- The devil is in the details, especially for complex algorithms.
- We will do a crash course on testing in Python

WARNING: Bad software can cause losses of million $/€ or even kill people. Suggested reading: Software Horror Stories[292]

[292] https://www.cs.tau.ac.il/~nachumd/horror.html

Where Is Your Software?

As a data scientist, you might likely end up with code which is algorithmically complex, but maybe not too big in size. Either way, when red line is crossed you should start testing properly:



In a typical scenario, you are a junior programmer and your senior colleague ask you to write a function to perform some task, giving only an informal description:

```
[10]: def my_sum(x,y):
          """ RETURN the sum of x and y
          """
          raise Exception("TODO IMPLEMENT ME!")
```

Even better, your colleague might provide you with some automated tests you might run to check your function meets his/her expectations. If you are smart, you will even write tests for your own functions to make sure every little piece you add to your software is a solid block you can build upon.

5.19.8 Testing with asserts

NOTE: in this book we test with `assert`, but there are much better frameworks for testing!

If you get serious about software development, please consider using something like PyTest[293] (recent and clean) or Unittest[294] (Python default testing suite, has more traditional approach)

We can use *assert* to quickly test functions, and verify they behave like they should.

For example, from this function:

```
[11]: def my_sum(x, y):
          s = x + y
          return s
```

We expect that `my_sum(2,3)` gives `5`. We can write in Python this expectation by using an `assert`:

[293] https://docs.pytest.org/en/stable/
[294] https://docs.python.org/3/library/unittest.html

```
[12]: assert my_sum(2,3) == 5
```

Se `my_sum` is correctly implemented:

1. `my_sum(2,3)` will give `5`
2. the boolean expression `my_sum(2,3) == 5` will give `True`
3. `assert True` will be exectued without producing any result, and the program execution will continue.

Otherwise, if `my_sum` is NOT correctly implemented like in this case:

```
def my_sum(x,y):
    return 666
```

1. `my_sum(2,3)` will produce the number `666`
2. the boolean expression `my_sum(2,3) == 5` will give`False`
3. `assert False` will interrupt the program execution, raising an exception of type `AssertionError`

Exercise structure

Exercises are often structured in the following format:

```
def my_sum(x,y):
    """ RETURN the sum of numbers x and y
    """
    raise Exception("TODO IMPLEMENT ME!")


assert my_sum(2,3) == 5
assert my_sum(3,1) == 4
assert my_sum(-2,5) == 3
```

If you attempt to execute the cell, you will see this error:

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-16-5f5c8512d42a> in <module>()
      6
      7
----> 8 assert my_sum(2,3) == 5
      9 assert my_sum(3,1) == 4
     10 assert my_sum(-2,5) == 3

<ipython-input-16-5f5c8512d42a> in somma(x, y)
      3     """ RETURN the sum of numbers x and y
      4     """
----> 5     raise Exception("TODO IMPLEMENT ME!")
      6
      7

Exception: TODO IMPLEMENT ME!
```

To fix them, you will need to:

1. substitute the row `raise Exception("TODO IMPLEMENT ME!")` with the body of the function
2. execute the cell

If cell execution doesn't result in raised exceptions, perfect ! It means your function does what it is expected to do (the `assert` which succeed do not produce any output)

Otherwise, if you see some `AssertionError`, probably you did something wrong.

NOTE: The `raise Exception("TODO IMPLEMENT ME")` is put there to remind you that the function has a big problem, that is, it doesn't have any code !!! In long programs, it might happen you know you need a function, but in that moment you don't know what code put in th efunction body. So, instead of putting in the body commands that do nothing like `print()` or `pass` or `return None`, it is WAY BETTER to raise exceptions so that if by chance the program reaches the function, the execution is suddenly stopped and the user is signalled with the nature and position of the problem. Many editors for programmers, when automatically generating code, put inside function skeletons to implement some Exception like this.

Let's try to willingly write a wrong function body, which always return `5`, independently from `x` and `y` given in input:

```
def my_sum(x,y):
    """ RETURN the sum of numbers x and y
    """
    return 5

assert my_sum(2,3) == 5
assert my_sum(3,1) == 4
assert my_sum(-2,5) == 3
```

In this case the first assertion succeeds and so the execution simply passes to the next row, which contains another `assert`. We expect that `my_sum(3,1)` gives 4, but our ill-written function returns `5` so this `assert` fails. Note how the execution is interrupted at the *second* `assert`:

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-19-e5091c194d3c> in <module>()
      6
      7 assert my_sum(2,3) == 5
----> 8 assert my_sum(3,1) == 4
      9 assert my_sum(-2,5) == 3

AssertionError:
```

If we implement well the function and execute the cell we will see no output: this means the function successfully passed the tests and we can conclude that it is *correct with reference to the tests*:

ATTENTION: always remember that these kind of tests are *never* exhaustive ! If tests pass it is only an indication the function *might* be correct, but it is never a certainty !

[13]:

```
def my_sum(x,y):
    """ RETURN the sum of numbers x and y
    """
    return x + y

assert my_sum(2,3) == 5
assert my_sum(3,1) == 4
assert my_sum(-2,5) == 3
```

EXERCISE: Try to write the body of the function `multiply`:

- substitute `raise Exception("TODO IMPLEMENT ME")` with `return x * y` and execute the cell. If you have written correctly, nothing should happen. In this case, congratulatins! The code you have written is correct *with reference to the tests* !
- Try to substitute instead with `return 10` and see what happens.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[14]:
```python
def my_mul(x,y):
    """ RETURN the multiplication of numbers x and y
    """

    return x * y



assert my_mul(2,5) == 10
assert my_mul(0,2) == 0
assert my_mul(3,2) == 6
```

</div>

[14]:
```python
def my_mul(x,y):
    """ RETURN the multiplication of numbers x and y
    """
    raise Exception('TODO IMPLEMENT ME !')


assert my_mul(2,5) == 10
assert my_mul(0,2) == 0
assert my_mul(3,2) == 6
```

even_numbers example

Let’s see a slightly more complex function:

[15]:
```python
def even_numbers(n):
    """
    Return a list of the first n even numbers

    Zero is considered to be the first even number.

    >>> even_numbers(5)
    [0,2,4,6,8]
    """
    raise Exception("TODO IMPLEMENT ME!")
```

In this case, if you run the function as it is, you are reminded to implement it:

```python
>>> even_numbers(5)
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-2-d2cbc915c576> in <module>()
----> 1 even_numbers(5)

<ipython-input-1-a20a4ea4b42a> in even_numbers(n)
      8     [0,2,4,6,8]
      9     """
---> 10     raise Exception("TODO IMPLEMENT ME!")

Exception: TODO IMPLEMENT ME!
```

Why? The instruction

```
raise Exception("TODO IMPLEMENT ME!")
```

tells Python to immediatly stop execution, and signal an error to the caller of the function `even_number`. If there were commands right after `raise Exception("TODO IMPLEMENT ME")`, they would not be executed. Here, we are directly calling the function from the prompt, and we didn't tell Python how to handle the `Exception`, so Python just stopped and showed the error message given as parameter to the `Exception`

Spend time reading the function text!

Always carefully read the function text and ask yourself questions! What is the supposed input? What should be the output? Is there any output to return at all, or should you instead modify *in-place* a passed parameter (i.e. for example, when you sort a list)? Are there any edge cases, es what happens for `n=0`)? What about `n < 0` ?

Let's code a possible solution. As it often happens, first version may be buggy, in this case for example purposes we intentionally introduce a bug:

[16]:
```
def even_numbers(n):
    """
    Return a list of the first n even numbers

    Zero is considered to be the first even number.

    >>> even_numbers(5)
    [0,2,4,6,8]
    """
    r = [2 * x for x in range(n)]
    r[n // 2] = 3    # <-- evil bug, puts number '3' in the middle, and 3 is not even .
↪.
    return r
```

Typically the first test we do is printing the output and do some 'visual inspection' of the result, in this case we find many numbers are correct but we might miss errors such as the wrong `3` in the middle:

[17]:
```
print(even_numbers(5))
```

```
[0, 2, 3, 6, 8]
```

Furthermore, if we enter commands a the prompt, each time we fix something in the code, we need to enter commands again to check everything is ok. This is inefficient, boring, and prone to errors.

Let's add assertions

To go beyond the dumb "visual inspection" testing, it's better to write some extra code to allow Python checking for us if the function actually returns what we expect, and throws an error otherwise. We can do so with `assert` command, which verifies if its argument is True. If it is not, it raises an `AssertionError` immediately stopping execution.

Here we check the result of `even_numbers(5)` is actually the list of even numbers `[0,2,4,6,8]` we expect:

```
assert even_numbers(5) == [0,2,4,6,8]
```

Since our code is faulty, `even_numbers` returns the wrong list `[0,2,3,6,8]` which is different from `[0,2,4,6,8]` so assertion fails showing `AssertionError`:

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-21-d4198f229404> in <module>()
----> 1 assert even_numbers(5) != [0,2,4,6,8]

AssertionError:
```

We got some output, but we would like to have it more informative. To do so, we may add a message, separated by a comma:

```
assert even_numbers(5) == [0,2,4,6,8], "even_numbers is not working !!"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-18-8544fcd1b7c8> in <module>()
----> 1 assert even_numbers(5) == [0,2,4,6,8], "even_numbers is not working !!"

AssertionError: even_numbers is not working !!
```

So if we modify code to fix bugs we can just launch the assert commands and have a quick feedback about possible errors.

Error kinds

As a fact of life, errors happen. Sometimes, your program may have inconsistent data, like wrong parameter type passed to a function (i.e. string instead of integer). A good principle to follow in these cases is to try have the program detect weird situations, and stop as early as such a situation is found (i.e. in the Therac 25 case, if you detect excessive radiation, showing a warning sign is not enough, it's better to stop). Note stopping might not always be the desirable solution (if one pidgeon enters one airplane engine, you don't want to stop all the other engines). If you want to check function parameters are correct, you do the so called *precondition checking*.

There are roughly two cases for errors, external user misusing you program, and just plain wrong code. Let's analyze both:

Error kind a) An external user misuses you program.

You can assume whover uses your software, final users or other programmers , they will try their very best to wreck your precious code by passing all sort of non-sense to functions. Everything can come in, strings instead of numbers, empty arrays, `None` objects … In this case you should signal the user he made some mistake. The most crude signal you can have is raising an `Exception` with `raise Exception("Some error occurred")`, which will stop the program and print the stacktrace in the console. Maybe final users won't understand a stacktrace, but at least programmers hopefully will get a clue about what is happening.

In these case you can raise an appropriate Exception, like TypeError[295] for wrong types and ValueError[296] for more generic errors. Other basic exceptions can be found in Python documentation[297]. Notice you can also define your own, if needed (we won't consider custom exceptions in this course).

NOTE: Many times, you can consider yourself the 'careless external user' to guard against.

Let's enrich the function with some appropriate type checking:

Note that for checking input types, you can use the function `type()` :

[18]:
```
type(3)
```

[18]:
```
int
```

[19]:
```
type("ciao")
```

[19]:
```
str
```

Let's add the code for checking the *even_numbers example*:

[20]:
```
def even_numbers(n):
    """
    Return a list of the first n even numbers

    Zero is considered to be the first even number.

    >>> even_numbers(5)
    [0,2,4,6,8]
    """
    if type(n) is not int:
        raise TypeError("Passed a non integer number: " + str(n))

    if n < 0:
        raise ValueError("Passed a negative number: " + str(n))

    r = [2 * x for x in range(n)]
    return r
```

Let's pass a wrong type and see what happens:

```
>>> even_numbers("ciao")

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-a908b20f00c4> in <module>()
```

(continues on next page)

295 https://docs.python.org/3/library/exceptions.html#TypeError
296 https://docs.python.org/3/library/exceptions.html#ValueError
297 https://docs.python.org/3/library/exceptions.html#built-in-exceptions

(continued from previous page)

```
----> 1 even_numbers("ciao")

<ipython-input-13-b0b3a85f2b2a> in even_numbers(n)
      9     """
     10     if type(n) is not int:
---> 11         raise TypeError("Passed a non integer number: " + str(n))
     12
     13     if n < 0:

TypeError: Passed a non integer number: ciao
```

Now let's try to pass a negative number - it should suddenly stop with a meaningful message:

```
>>> even_numbers(-5)

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-15-3f648fdf6de7> in <module>()
----> 1 even_numbers(-5)

<ipython-input-13-b0b3a85f2b2a> in even_numbers(n)
     12
     13     if n < 0:
---> 14         raise ValueError("Passed a negative number: " + str(n))
     15
     16     r = [2 * x for x in range(n)]

ValueError: Passed a negative number: -5
```

Now, even if you ship your code to careless users, and as soon as they commit a mistrake, they will get properly notified.

Error kind b): Your code is just plain wrong

In this case, it's 100% your fault, and these sort of bugs should never pop up in production. For example your code passes internally wrong stuff, like strings instead of integers, or wrong ranges (typically integer outside array bounds). So if you have an internal function nobody else should directly call, and you suspect it is being passed wrong parameters or at some point it has inconsistent data, to quickly spot the error you could add an assertion:

[21]:
```python
def even_numbers(n):
    """
    Return a list of the first n even numbers

    Zero is considered to be the first even number.

    >>> even_numbers(5)
    [0,2,4,6,8]
    """
    assert type(n) is int, "type of n is not correct: " + str(type(n))
    assert n >= 0, "Found negative n: " + str(n)

    r = [2 * x for x in range(n)]

    return r
```

As before, the function will stop as soon we call it we wrong parameters. The big difference is, this time we are assuming `even_numbers` is just for personal use and nobody else except us should directly call it.

Since assertion consume CPU time, IF we care about performances AND once we are confident our program behaves correctly, we can even remove them from compiled code by using the `-O` compiler flag. For more info, see Python wiki[298]

EXERCISE: try to call latest definition of `even_numbers` with wrong parameters, and see what happens.

NOTE: here we are using the correct definition of `even_numbers`, not the buggy one with the `3` in the middle of returned list !

[ ]:

```

```

5.20 Commandments

The Supreme Committee for the Doctrine of Coding has ruled important Commandments you shall follow.

If you accept their wise words, you shall become a true Python Jedi.

WARNING: if you don't follow the Commandments, you will end up in *Debugging Hell* !

5.20.1 I COMMANDMENT

You shall write Python code

Who does not writes Python code, does not learn Python

5.20.2 II COMMANDMENT

Whenever you insert a variable in a `for` cycle, such variables must be new

If you defined the variable before, you shall not reintroduce it in a `for`, because doing so might bring confusion in the minds of the readers.

So avoid such sins:

[1]:

```python
i = 7
for i in range(3):  # sin, you lose variable i
    print(i)

print(i)   # prints 2 and not 7  !!
```

```
0
1
2
2
```

298 https://wiki.python.org/moin/UsingAssertionsEffectively

[2]:
```
for i in range(2):

    for i in  range(5):  # debugging hell, you lose the i of external cycle
        print(i)

    print(i)  # prints 4 !!
```

```
0
1
2
3
4
4
0
1
2
3
4
4
```

[3]:
```
def f(i):
    for i in range(3): # sin, you lose parameter i
        print(i)

    print(i)  # prints 2, not the 7 we passed!

f(7)
```

```
0
1
2
2
```

5.20.3 III COMMANDMENT

You shall never reassign function parameters

You shall never ever perform any of these assignments, as you risk losing the parameter passed during function call:

[4]:
```
def sin(my_int):
    my_int = 666             # you lost the 5 passed from external call!
    print(my_int)            # prints 666

x = 5
sin(x)
```

```
666
```

Same reasoning can be applied to all other types:

[5]:
```
def evil(my_string):
    my_string = "666"
```

[6]:
```
def disgrace(my_list):
    my_list = [666]
```

```
[7]: def delirium(my_dict):
         my_dict = {"evil":666}
```

For the sole case when you have composite parameters like lists or dictionaries, you can write like below IF AND ONLY IF the function description requires to MODIFY the internal elements of the parameter (like for example sorting a list in-place or changing the field of a dictionary.

```
[8]: # MODIFY my_list in some way
     def allowed(my_list):
         my_list[2] = 9

     outside = [8,5,7]
     allowed(outside)
     print(outside)
```

```
[8, 5, 9]
```

On the other hand, if the function requires to RETURN a NEW object, you shall not fall into the temptation of modifying the input:

```
[9]:
     # RETURN a NEW sorted list
     def pain(my_list):
         my_list.sort()     # BAD, you are modifying the input list instead of creating a␣
     ↪new one!
         return my_list
```

```
[10]: # RETURN a NEW list
      def crisis(my_list):
          my_list[0] = 5     # BAD, as above
          return my_list
```

```
[11]: # RETURN a NEW dictionary
      def torment(my_dict):
          my_dict['a'] = 6   # BAD, you are modifying the input dictionary instead of␣
      ↪creating a new one!
          return my_dict
```

```
[12]: # RETURN a NEW class instance
      def desperation(my_instance):
          my_instance.my_field = 6   # BAD, you are modifying the input object
                                     #      instead of creating a new one!
          return istanza_di_classe
```

5.20.4 IV COMMANDMENT

You shall never ever reassign values to function calls or mmethods

WRONG:

```
my_function() = 666
my_function() = 'evil'
my_function() = [666]
```

CORRECT:

```
x = 5
y = my_fun()
z = []
z[0] = 7
d = dict()
d["a"] = 6
```

Function calls like `my_function()` return calculations results and store them in a box in memory which is only created for the purposes of the call, and Python will not allow us to reuse it like it were a variabile.

Whenever you see `name()` in the left part, it *cannot* be followed by the equality sign = (but it can be followed by two equals sign == if you are doing a comparison).

5.20.5 V COMMANDMENT

You shall never ever redefine system functions

Python has several system defined functions. For example `list` is a Python type: as such, you can use it for example as a function to convert some type to a list:

```
[13]: list("ciao")
```

```
[13]: ['c', 'i', 'a', 'o']
```

When you allow the forces of evil to take the best of you, you might be tempted to use reserved words like `list` as a variable for you own miserable purposes:

```
[14]: list = ['my', 'pitiful', 'list']
```

Python allows you to do so, but we do **not**, for the consequences are disastrous.

For example, if you now attempt to use `list` for its intended purpose like casting to list, it won't work anymore:

```
list("ciao")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-4-c63add832213> in <module>()
----> 1 list("ciao")

TypeError: 'list' object is not callable
```

In particular, we recommend to **not redefine** these precious functions:

- `bool`, `int`,`float`,`tuple`,`str`,`list`,`set`,`dict`
- `max`, `min`, `sum`
- `next`, `iter`
- `id`, `dir`, `vars`,`help`

5.20.6 VI COMMANDMENT

You shall use `return` command only if you see written RETURN in function description!

If there is no `return` in function description, the function is intended to return `None`. In this case you don't even need to write `return None`, as Python will do it implicitly for you.

5.20.7 VII COMMANDMENT

You shall also write on paper!

If staring at the monitor doesn't work, help yourself and draw a representation of the state sof the program. Tables, nodes, arrows, all can help figuring out a solution for the problem.

5.20.8 VIII COMMANDMENT

You shall never ever reassing `self` !

You shall never write horror such as this:

[15]:
```
class MyClass:
    def my_method(self):
        self = {'my_field':666}    # SIN
```

Since `self` is a kind of a dictionary, you might be tempted to write like above, but to external world it will bring no effect.

For example, let's suppose somebody from outside makes a call like this:

[16]:
```
mc = MyClass()
mc.my_method()
```

After the call `mc` will not point to `{'my_field':666}`

[17]:
```
mc
```

[17]:
```
<__main__.MyClass at 0x7f6b20246160>
```

and will not have `my_field`:

```
mc.my_field
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-26-5c4e6630908d> in <module>()
----> 1 mc.my_field

AttributeError: 'MyClass' object has no attribute 'my_field'
```

Following the same reasoning, you shall never reassign `self` to lists or others things:

[18]:
```python
class MyClass:
    def my_method(self):
        self = ['evil']      # YET ANOTHER SIN
        self = 666           # NO NO NO
```

5.20.9 IX COMMANDMENT

You shall test!

Untested code by definition *does not work*. For ideas on how to test it, have a look at Errors and testing[299]

5.20.10 X COMMANDMENT

You shall never ever add nor remove elements from a sequence you are iterating with a `for` !

Falling into such temptations **would produce totally unpredictable behaviours** (do you know the expression *pulling the rug out from under your feet ?*)

Do not add, because you risk to walk on a tapis roulant that never turns off:

```python
my_list = ['a','b','c','d','e']
for el in my_list:
    my_list.append(el)   # YOU ARE CLOGGING COMPUTER MEMORY
```

Do not remove, because you risk to corrupt the natural order of things:

[19]:
```python
my_list = ['a','b','c','d','e']

for el in my_list:
    my_list.remove(el)    # VERY BAD IDEA
```

Look at the code. You think we removed eveything, uh?

[20]:
```python
my_list
```

[20]:
```
['b', 'd']
```

`O_o'` Do not even try to make sense of such sorcery - nobody can, because it is related to Python internal implementation.

Our version of Python gives this absurd result, yours may give another. Same applies for iteration on sets and dictionaries. **You are warned**.

If you really need to remove stuff from the sequence you are iterating on, use a while cycle[300] or first make a copy of the original sequence.

[ ]:

[299] https://en.softpython.org/errors-and-testing/errors-and-testing-sol.ipynb
[300] https://en.softpython.org/control-flow/flow3-while-sol.html

CHAPTER

SIX

B - DATA ANALYSIS

6.1 Data formats solutions

6.1.1 Download exercises zip

Browse files online[301]

6.1.2 Introduction

In this tutorial we will see how to load and write tabular data such as CSV, and we will mention tree-like data such as JSON files. We will also spend a couple of words about opendata catalogs and licenses (creative commons).

Graph formats will be discussed in a separate notebook[302]

What to do

- unzip exercises in a folder, you should get something like this:

```
-jupman.py
-formats
     formats.ipynb
     formats-sol.ipynb
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

- open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `formats/formats.ipynb`
- Go on reading that notebook, and follow instuctions inside.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

[301] https://github.com/DavidLeoni/sciprog-ds/tree/master/formats
[302] https://en.softpython.org/graph-formats/graph-formats-sol.html

6.1.3 1. line files

Line files are typically text files which contain information grouped by lines. An example using historical characters might be like the following:

```
Leonardo
da Vinci
Sandro
Botticelli
Niccolò
Macchiavelli
```

We can immediately see a regularity: first two lines contain data of Leonardo da Vinci, second one the name and then the surname. Successive lines instead have data of Sandro Botticelli, with again first the name and then the surname and so on.

We might want to do a program that reads the lines and prints on the terminal names and surnames like the following:

```
Leonardo da Vinci
Sandro Botticelli
Niccolò Macchiavelli
```

To start having an approximation of the final result, we can open the file, read only the first line and print it:

[1]:
```python
with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    print(line)
```

```
Leonardo
```

What happened? Let's examing first rows:

open command

The command

```python
open('people-simple.txt', encoding='utf-8')
```

allows us to open the text file by telling PYthon the file path `'people-simple.txt'` and the encoding in which it was written (`encoding='utf-8'`).

The encoding

The encoding dependes on the operating system and on the editor used to write the file. When we open a file, Python is not capable to divine the encoding, and if we do not specify anything Python might open the file assuming an encoding different from the original - in other words, if we omit the encoding (or we put a wrong one) we might end up seeing weird characters (like little squares instead of accented letters).

In general, when you open a file, try first to specify the encoding `utf-8` which is the most common one. If it doesn't work try others, for example for files written in south Europe with Windows you might check `encoding='latin-1'`. If you open a file written elsewhere, you might need other encodings. For more in-depth information, you can read

Dive into Python - Chapter 4 - Strings[303], and Dive into Python - Chapter 11 - File[304], **both of which are extremely recommended readings**.

with block

The `with` defines a block with instructions inside:

```
with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    print(line)
```

We used the `with` to tell PYthon that in any case, even if errors occur, we want that after having used the file, that is after having executed the instructions inside the internal block (the `line=f.readline()` and `print(line)`) Python must automatically close the file. Properly closing a file avoids to waste memory resources and creating hard to find paranormal errors. If you want to avoid hunting for never closed zombie files, always remember to open all files in `with` blocks! Furthermore, at the end of the row in the part `as f:` we assigned the file to a variable hereby called `f`, but we could have used any other name we liked.

WARNING: To indent the code, ALWAYS use sequences of four white spaces. Sequences of 2 spaces. Sequences of only 2 spaces even if allowed are not recommended.

WARNING: Depending on the editor you use, by pressing TAB you might get a sequence o f white spaces like it happens in Jupyter (4 spaces which is the recommended length), or a special tabulation character (to avoid)! As much as this annoying this distinction might appear, remember it because it might generate very hard to find errors.

WARNING: In the commands to create blocks such as `with`, always remember to put the character of colon **:** at the end of the line !

The command

```
line=f.readline()
```

puts in the variable `line` the entire line, like a string. Warning: the string will contain at the end the special character of line return !

You might wonder where that `readline` comes from. Like everything in Python, our variable `f` which represents the file we just opened is an object, and like any object, depending on its type, it has particular methods we can use on it. In this case the method is `readline`.

The following command prints the string content:

```
print(line)
```

⊕ **1.1 EXERCISE**: Try to rewrite here the block we've just seen, and execute the cell by pressing Control-Enter. Rewrite the code with the fingers, not with copy-paste ! Pay attention to correct indentation with spaces in the block.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[303] https://diveintopython3.problemsolving.io/strings.html
[304] https://diveintopython3.problemsolving.io/files.html

[2]:
```
# write here

with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    print(line)
```

```
Leonardo
```

</div>

[2]:
```
# write here
```

```
Leonardo
```

⊛ **1.2 EXERCISE**: you might wondering what exactly is that `f`, and what exatly the method `readlines` should be doing. When you find yourself in these situations, you might help yourself with functions `type` and `help`. This time, directly copy paste the same code here, but insert inside `with` block the commands:

- `print(type(f))`
- `print(help(f))`
- `print(help(f.readline))` # Attention: remember the f. before the readline !!

Every time you add something, try to execute with Control+Enter and see what happens

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[3]:
```
# write here the code (copy and paste)
with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    print(type(f))
    print(help(f.readline))
    print(help(f))
    print(line)
```

```
<class '_io.TextIOWrapper'>
Help on built-in function readline:

readline(size=-1, /) method of _io.TextIOWrapper instance
    Read until newline or EOF.

    Returns an empty string if EOF is hit immediately.

None
Help on TextIOWrapper object:

class TextIOWrapper(_TextIOBase)
 |  Character and line based layer over a BufferedIOBase object, buffer.
 |
 |  encoding gives the name of the encoding that the stream will be
 |  decoded or encoded with. It defaults to locale.getpreferredencoding(False).
 |
```

(continues on next page)

(continued from previous page)

```
|  errors determines the strictness of encoding and decoding (see
|  help(codecs.Codec) or the documentation for codecs.register) and
|  defaults to "strict".
|
|  newline controls how line endings are handled. It can be None, '',
|  '\n', '\r', and '\r\n'.  It works as follows:
|
|  * On input, if newline is None, universal newlines mode is
|    enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
|    these are translated into '\n' before being returned to the
|    caller. If it is '', universal newline mode is enabled, but line
|    endings are returned to the caller untranslated. If it has any of
|    the other legal values, input lines are only terminated by the given
|    string, and the line ending is returned to the caller untranslated.
|
|  * On output, if newline is None, any '\n' characters written are
|    translated to the system default line separator, os.linesep. If
|    newline is '' or '\n', no translation takes place. If newline is any
|    of the other legal values, any '\n' characters written are translated
|    to the given string.
|
|  If line_buffering is True, a call to flush is implied when a call to
|  write contains a newline character.
|
|  Method resolution order:
|      TextIOWrapper
|      _TextIOBase
|      _IOBase
|      builtins.object
|
|  Methods defined here:
|
|  __getstate__(…)
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __new__(*args, **kwargs) from builtins.type
|      Create and return a new object.  See help(type) for accurate signature.
|
|  __next__(self, /)
|      Implement next(self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  close(self, /)
|      Flush and close the IO object.
|
|      This method has no effect if the file is already closed.
|
|  detach(self, /)
|      Separate the underlying buffer from the TextIOBase and return it.
|
|      After the underlying buffer has been detached, the TextIO is in an
|      unusable state.
|
```

(continues on next page)

(continued from previous page)

```
|  fileno(self, /)
|      Returns underlying file descriptor if one exists.
|
|      OSError is raised if the IO object does not use a file descriptor.
|
|  flush(self, /)
|      Flush write buffers, if applicable.
|
|      This is not implemented for read-only and non-blocking streams.
|
|  isatty(self, /)
|      Return whether this is an 'interactive' stream.
|
|      Return False if it can't be determined.
|
|  read(self, size=-1, /)
|      Read at most n characters from stream.
|
|      Read from underlying buffer until we have n characters or we hit EOF.
|      If n is negative or omitted, read until EOF.
|
|  readable(self, /)
|      Return whether object was opened for reading.
|
|      If False, read() will raise OSError.
|
|  readline(self, size=-1, /)
|      Read until newline or EOF.
|
|      Returns an empty string if EOF is hit immediately.
|
|  seek(self, cookie, whence=0, /)
|      Change stream position.
|
|      Change the stream position to the given byte offset. The offset is
|      interpreted relative to the position indicated by whence.  Values
|      for whence are:
|
|      * 0 -- start of stream (the default); offset should be zero or positive
|      * 1 -- current stream position; offset may be negative
|      * 2 -- end of stream; offset is usually negative
|
|      Return the new absolute position.
|
|  seekable(self, /)
|      Return whether object supports random access.
|
|      If False, seek(), tell() and truncate() will raise OSError.
|      This method may need to do a test seek().
|
|  tell(self, /)
|      Return current stream position.
|
|  truncate(self, pos=None, /)
|      Truncate file to size bytes.
|
|      File pointer is left unchanged.  Size defaults to the current IO
```

(continues on next page)

(continued from previous page)

```
 |      position as reported by tell().  Returns the new size.
 |
 |  writable(self, /)
 |      Return whether object was opened for writing.
 |
 |      If False, write() will raise OSError.
 |
 |  write(self, text, /)
 |      Write string to stream.
 |      Returns the number of characters written (which is always equal to
 |      the length of the string).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  buffer
 |
 |  closed
 |
 |  encoding
 |      Encoding of the text stream.
 |
 |      Subclasses should override.
 |
 |  errors
 |      The error setting of the decoder or encoder.
 |
 |      Subclasses should override.
 |
 |  line_buffering
 |
 |  name
 |
 |  newlines
 |      Line endings translated so far.
 |
 |      Only line endings translated during reading are considered.
 |
 |      Subclasses should override.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _IOBase:
 |
 |  __del__(...)
 |
 |  __enter__(...)
 |
 |  __exit__(...)
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  readlines(self, hint=-1, /)
 |      Return a list of lines from the stream.
 |
 |      hint can be specified to control the number of lines read: no more
 |      lines will be read if the total size (in bytes/characters) of all
```

(continues on next page)

(continued from previous page)

```
 |      lines so far exceeds hint.
 |
 |  writelines(self, lines, /)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _IOBase:
 |
 |  __dict__

None
Leonardo
```

</div>

```
[3]: # write here the code (copy and paste)
```

```
<class '_io.TextIOWrapper'>
Help on built-in function readline:

readline(size=-1, /) method of _io.TextIOWrapper instance
    Read until newline or EOF.

    Returns an empty string if EOF is hit immediately.

None
Help on TextIOWrapper object:

class TextIOWrapper(_TextIOBase)
 |  Character and line based layer over a BufferedIOBase object, buffer.
 |
 |  encoding gives the name of the encoding that the stream will be
 |  decoded or encoded with. It defaults to locale.getpreferredencoding(False).
 |
 |  errors determines the strictness of encoding and decoding (see
 |  help(codecs.Codec) or the documentation for codecs.register) and
 |  defaults to "strict".
 |
 |  newline controls how line endings are handled. It can be None, '',
 |  '\n', '\r', and '\r\n'.  It works as follows:
 |
 |  * On input, if newline is None, universal newlines mode is
 |    enabled. Lines in the input can end in '\n', '\r', or '\r\n', and
 |    these are translated into '\n' before being returned to the
 |    caller. If it is '', universal newline mode is enabled, but line
 |    endings are returned to the caller untranslated. If it has any of
 |    the other legal values, input lines are only terminated by the given
 |    string, and the line ending is returned to the caller untranslated.
 |
 |  * On output, if newline is None, any '\n' characters written are
 |    translated to the system default line separator, os.linesep. If
 |    newline is '' or '\n', no translation takes place. If newline is any
 |    of the other legal values, any '\n' characters written are translated
 |    to the given string.
 |
```

(continues on next page)

(continued from previous page)

```
 |  If line_buffering is True, a call to flush is implied when a call to
 |  write contains a newline character.
 |
 |  Method resolution order:
 |      TextIOWrapper
 |      _TextIOBase
 |      _IOBase
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __getstate__(…)
 |
 |  __init__(self, /, *args, **kwargs)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  __next__(self, /)
 |      Implement next(self).
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  close(self, /)
 |      Flush and close the IO object.
 |
 |      This method has no effect if the file is already closed.
 |
 |  detach(self, /)
 |      Separate the underlying buffer from the TextIOBase and return it.
 |
 |      After the underlying buffer has been detached, the TextIO is in an
 |      unusable state.
 |
 |  fileno(self, /)
 |      Returns underlying file descriptor if one exists.
 |
 |      OSError is raised if the IO object does not use a file descriptor.
 |
 |  flush(self, /)
 |      Flush write buffers, if applicable.
 |
 |      This is not implemented for read-only and non-blocking streams.
 |
 |  isatty(self, /)
 |      Return whether this is an 'interactive' stream.
 |
 |      Return False if it can't be determined.
 |
 |  read(self, size=-1, /)
 |      Read at most n characters from stream.
 |
 |      Read from underlying buffer until we have n characters or we hit EOF.
 |      If n is negative or omitted, read until EOF.
 |
```

(continues on next page)

(continued from previous page)

```
 |  readable(self, /)
 |      Return whether object was opened for reading.
 |
 |      If False, read() will raise OSError.
 |
 |  readline(self, size=-1, /)
 |      Read until newline or EOF.
 |
 |      Returns an empty string if EOF is hit immediately.
 |
 |  seek(self, cookie, whence=0, /)
 |      Change stream position.
 |
 |      Change the stream position to the given byte offset. The offset is
 |      interpreted relative to the position indicated by whence.  Values
 |      for whence are:
 |
 |      * 0 -- start of stream (the default); offset should be zero or positive
 |      * 1 -- current stream position; offset may be negative
 |      * 2 -- end of stream; offset is usually negative
 |
 |      Return the new absolute position.
 |
 |  seekable(self, /)
 |      Return whether object supports random access.
 |
 |      If False, seek(), tell() and truncate() will raise OSError.
 |      This method may need to do a test seek().
 |
 |  tell(self, /)
 |      Return current stream position.
 |
 |  truncate(self, pos=None, /)
 |      Truncate file to size bytes.
 |
 |      File pointer is left unchanged.  Size defaults to the current IO
 |      position as reported by tell().  Returns the new size.
 |
 |  writable(self, /)
 |      Return whether object was opened for writing.
 |
 |      If False, write() will raise OSError.
 |
 |  write(self, text, /)
 |      Write string to stream.
 |      Returns the number of characters written (which is always equal to
 |      the length of the string).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  buffer
 |
 |  closed
 |
 |  encoding
 |      Encoding of the text stream.
```

(continues on next page)

(continued from previous page)

```
 |
 |      Subclasses should override.
 |
 |  errors
 |      The error setting of the decoder or encoder.
 |
 |      Subclasses should override.
 |
 |  line_buffering
 |
 |  name
 |
 |  newlines
 |      Line endings translated so far.
 |
 |      Only line endings translated during reading are considered.
 |
 |      Subclasses should override.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from _IOBase:
 |
 |  __del__(…)
 |
 |  __enter__(…)
 |
 |  __exit__(…)
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  readlines(self, hint=-1, /)
 |      Return a list of lines from the stream.
 |
 |      hint can be specified to control the number of lines read: no more
 |      lines will be read if the total size (in bytes/characters) of all
 |      lines so far exceeds hint.
 |
 |  writelines(self, lines, /)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from _IOBase:
 |
 |  __dict__

None
Leonardo
```

First we put the content of the first line into the variable `line`, now we might put it in a variable witha more meaningful name, like `name`. Also, we can directly read the next row into the variable `surname` and then print the concatenation of both:

[4]:
```python
with open('people-simple.txt', encoding='utf-8') as f:
    name=f.readline()
    surname=f.readline()
```

(continues on next page)

(continued from previous page)

```
    print(name + ' ' + surname)
```

```
Leonardo
 da Vinci
```

PROBLEM ! The printing puts a weird carriage return. Why is that? If you remember, first we said that `readline` reads the line content in a string adding to the end also the special newline character. To eliminate it, you can use the command `rstrip()`:

[5]:
```
with open('people-simple.txt', encoding='utf-8') as f:
    name=f.readline().rstrip()
    surname=f.readline().rstrip()
    print(name + ' ' + surname)
```

```
Leonardo da Vinci
```

⊛ **1.3 EXERCISE**: Again, rewrite the block above in the cell below, ed execute the cell with Control+Enter. Question: what happens if you use `strip()` instead of `rstrip()`? What about `lstrip()`? Can you deduce the meaning of `r` and `l`? If you can't manage it, try to use python command `help` by calling `help(string.rstrip)`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[6]:
```
# write here

with open('people-simple.txt', encoding='utf-8') as f:
    name=f.readline().rstrip()
    surname=f.readline().rstrip()
    print(name + ' ' + surname)
```

```
Leonardo da Vinci
```

</div>

[6]:
```
# write here
```

```
Leonardo da Vinci
```

Very good, we have the first line ! Now we can read all the lines in sequence. To this end, we can use a `while` cycle:

[7]:
```
with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    while line != "":
        name = line.rstrip()
        surname=f.readline().rstrip()
        print(name + ' ' + surname)
        line=f.readline()
```

```
Leonardo da Vinci
Sandro Botticelli
Niccolò Macchiavelli
```

NOTE: In Python there are shorter ways[305] to read a text file line by line, we used this approach to make explicit all passages.

What did we do? First, we added a `while` cycle in a new block

WARNING: In new block, since it is already within the external `with`, the instructions are indented of 8 spaces and not 4! If you use the wrong spaces, bad things happen !

We first read a line, and two cases are possible:

a. we are the end of the file (or file is empty) : in this case `readline()` call returns an empty string

b. we are not at the end of the file: the first line is put as a string inside the variable `line`. Since Python internally uses a pointer to keep track at which position we are when reading inside the file, after the read such pointer is moved at the beginning of the next line. This way the next call to `readline()` will read a line from the new position.

In `while` block we tell Python to continue the cycle as long as `line` is *not* empty. If this is the case, inside the `while` block we parse the name from the line and put it in variable `name` (removing extra newline character with `rstrip()` as we did before), then we proceed reading the next line and parse the result inside the `surname` variable. Finally, we read again a line into the `line` variable so it will be ready for the next round of name extraction. If line is empty the cycle will terminate:

```
while line != "":                    # enter cycle if line contains characters
    name = line.rstrip()             # parses the name
    surname=f.readline().rstrip()    # reads next line and parses surname
    print(name + ' ' + surname)
    line=f.readline()                # read next line
```

⊛ **1.4 EXERCISE**: As before, rewrite in the cell below the code with the `while`, paying attention to the indentation (for the external `with` line use copy-and-paste):

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[8]:
```
# write here the code of internal while

with open('people-simple.txt', encoding='utf-8') as f:
    line=f.readline()
    while line != "":
        name = line.rstrip()
        surname=f.readline().rstrip()
        print(name + ' ' + surname)
        line=f.readline()
```

```
Leonardo da Vinci
Sandro Botticelli
Niccolò Macchiavelli
```

</div>

[8]:
```
# write here the code of internal while
```

305 https://thispointer.com/5-different-ways-to-read-a-file-line-by-line-in-python/

```
Leonardo da Vinci
Sandro Botticelli
Niccolò Macchiavelli
```

people-complex line file:

Look at the file `people-complex.txt`:

```
name: Leonardo
surname: da Vinci
birthdate: 1452-04-15
name: Sandro
surname: Botticelli
birthdate: 1445-03-01
name: Niccolò
surname: Macchiavelli
birthdate: 1469-05-03
```

Supposing to read the file to print this output, how would you do it?

```
Leonardo da Vinci, 1452-04-15
Sandro Botticelli, 1445-03-01
Niccolò Macchiavelli, 1469-05-03
```

Hint 1: to obtain the string `'abcde'`, the substring `'cde'`, which starts at index 2, you can ue the operator square brackets, using the index followed by colon **:**

[9]:
```python
x = 'abcde'
x[2:]
```

[9]:
```
'cde'
```

[10]:
```python
x[3:]
```

[10]:
```
'de'
```

Hint 2: To know the length of a string, use the function `len`:

[11]:
```python
len('abcde')
```

[11]:
```
5
```

⊕ **1.5 EXERCISE**: Write here the solution of the exercise 'People complex':

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```python
# write here

with open('people-complex.txt', encoding='utf-8') as f:
    line=f.readline()
    while line != "":
        name = line.rstrip()[len("name: "):]
        surname= f.readline().rstrip()[len("surname: "):]
        born = f.readline().rstrip()[len("birthdate: "):]
        print(name + ' ' + surname + ', ' + born)
        line=f.readline()
```

```
Leonardo da Vinci, 1452-04-15
Sandro Botticelli, 1445-03-01
Niccolò Macchiavelli, 1469-05-03
```

</div>

[12]:
```
# write here
```

```
Leonardo da Vinci, 1452-04-15
Sandro Botticelli, 1445-03-01
Niccolò Macchiavelli, 1469-05-03
```

Exercise - line file immersione-in-python-toc

⊛⊛⊛ This exercise is more challenging, if you are a beginner you might skip it and go on to CSVs

The book Dive into Python is nice and for the italian version there is a PDF, which has a problem though: if you try to print it, you will discover that the index is missing. Without despairing, we found a program to extract titles in a file as follows, but you will discover it is not exactly nice to see. Since we are Python ninjas, we decided to transform raw titles in a real table of contents[306]. Sure enough there are smarter ways to do this, like loading the pdf in Python with an appropriate module for pdfs, still this makes for an interesting exercise.

You are given the file `immersione-in-python-toc.txt`:

```
BookmarkBegin
BookmarkTitle: Il vostro primo programma Python
BookmarkLevel: 1
BookmarkPageNumber: 38
BookmarkBegin
BookmarkTitle: Immersione!
BookmarkLevel: 2
BookmarkPageNumber: 38
BookmarkBegin
BookmarkTitle: Dichiarare funzioni
BookmarkLevel: 2
BookmarkPageNumber: 41
BookmarkBeginint
BookmarkTitle: Argomenti opzionali e con nome
BookmarkLevel: 3
BookmarkPageNumber: 42
BookmarkBegin
BookmarkTitle: Scrivere codice leggibile
BookmarkLevel: 2
BookmarkPageNumber: 44
BookmarkBegin
BookmarkTitle: Stringhe di documentazione
BookmarkLevel: 3
BookmarkPageNumber: 44
BookmarkBegin
BookmarkTitle: Il percorso di ricerca di import
BookmarkLevel: 2
BookmarkPageNumber: 46
BookmarkBegin
```

(continues on next page)

306 http://softpython.readthedocs.io/it/latest/_static/toc-immersione-in-python-3.txt

(continued from previous page)

```
BookmarkTitle: Ogni cosa &#232; un oggetto
BookmarkLevel: 2
BookmarkPageNumber: 47
```

Write a python program to print the following output:

```
Il vostro primo programma Python  38
   Immersione!  38
   Dichiarare funzioni  41
      Argomenti opzionali e con nome  42
   Scrivere codice leggibile  44
      Stringhe di documentazione  44
   Il percorso di ricerca di import  46
   Ogni cosa è un oggetto  47
```

For this exercise, you will need to insert in the output artificial spaces, in a qunatity determined by the rows `BookmarkLevel`

QUESTION: what's that weird value `è` at the end of the original file? Should we report it in the output?

HINT 1: To convert a string into an integer number, use the function `int`:

[13]:
```python
x = '5'
```

[14]:
```python
x
```

[14]:
```
'5'
```

[15]:
```python
int(x)
```

[15]:
```
5
```

Warning: `int(x)` returns a value, and never modifies the argument `x`!

HINT 2: To substitute a substring in a string, you can use the method `.replace`:

[16]:
```python
x = 'abcde'
x.replace('cd', 'HELLO' )
```

[16]:
```
'abHELLOe'
```

HINT 3: while there is only one sequence to substitute, `replace` is fine, but if we had a milion of horrible sequences like `>`, `>`, `&x3e;`, what should we do? As good data cleaners, we recognize these are HTML escape sequences[307], so we could use methods specific to sequences like html.escape[308]. TRy it instead of `replace` and check if it works!

NOTE: Before using `html.unescape`, import the module `html` with the command:

```python
import html
```

HINT 4: To write *n* copies of a character, use `*` like this:

[17]:
```python
"b" * 3
```

[307] https://corsidia.com/materia/web-design/caratterispecialihtml
[308] https://docs.python.org/3/library/html.html#html.unescape

[17]:
```
'bbb'
```

[18]:
```
"b" * 7
```

[18]:
```
'bbbbbbb'
```

IMPLEMENTATION: Write here the solution for the line file `immersione-in-python-toc.txt`, and try execute it by pressing Control + Enter:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[19]:
```
# write here

import html

with open("immersione-in-python-toc.txt", encoding='utf-8') as f:

    line=f.readline()
    while line != "":
        line = f.readline().strip()
        title = html.unescape(line[len("BookmarkTitle: "):])
        line=f.readline().strip()
        level = int(line[len("BookmarkLevel: "):])
        line=f.readline().strip()
        page = line[len("BookmarkPageNumber: "):]
        print(("    " * level) + title + "   " + page)
        line=f.readline()
```

```
   Il vostro primo programma Python   38
      Immersione!   38
      Dichiarare funzioni   41
         Argomenti opzionali e con nome   42
      Scrivere codice leggibile   44
         Stringhe di documentazione   44
      Il percorso di ricerca di import   46
      Ogni cosa è un oggetto   47
```

</div>

[19]:
```
# write here
```

```
   Il vostro primo programma Python   38
      Immersione!   38
      Dichiarare funzioni   41
         Argomenti opzionali e con nome   42
      Scrivere codice leggibile   44
         Stringhe di documentazione   44
      Il percorso di ricerca di import   46
      Ogni cosa è un oggetto   47
```

6.1.4 2. CSV files

There can be various formats for tabular data, among which you surely know Excel (`.xls` or `.xslx`). Unfortunately, if you want to programmatically process data, you should better avoid them and prefer if possible the CSV format, literally 'Comma Separated Value'. Why? Excel format is very complex and may hide several things which have nothing to do with the raw data:

- formatting (bold fonts, colors …)
- merged cells
- formulas
- multiple tabs
- macros

Correctly parsing complex files may become a nightmare. Instead, CSVs are far simpler, so much so you can even open them witha simple text editor.

We will try to open some CSV, taking into consideration the possible problems we might get. CSVs are not necessarily the perfect solution for everything, but they offer more control over reading and typically if there are conversion problems is because we made a mistake, and not because the reader module decided on its own to exchange days with months in dates.

Why parsing a CSV ?

To load and process CSVs there exist many powerful and intuitive modules such as Pandas in Python or R dataframes. Yet, in this notebook we will load CSVs using the most simple method possible, that is reading row by row, mimicking the method already seen in the previous part of the tutorial. Don't think this method is primitive or stupid, according to the situation it may save the day. How? Some files may potentially occupy huge amounts of memory, and in moder laptops as of 2019 we only have 4 gigabytes of RAM, the memory where Python stores variables. Given this, Python base functions to read files try their best to avoid loading everything in RAM. Tyipcally a file is read sequentially one piece at a time, putting in RAM only one row at a time.

QUESTION 2.1: if we want to know if a given file of 1000 terabytes contains only 3 million rows in which the word 'ciao' is present, are we obliged to put in RAM *all* of the rows ?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: no, it is sufficient to keep in memory one row at a time, and hold the count in another variable

</div>

QUESTION 2.2: What if we wanted to take a 100 terabyte file and create another one by appending to each row of the first one the word 'ciao'? Should we put in RAM at the same time all the rows of the first file ? What about the rows of second one?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: No, it is enough to keep in RAM one row at a time, which is first read from the first file and then written right away in the second file.

</div>

Reading a CSV

We will start with artifical example CSV. Let's look at example-1.csv which you can find in the same folder as this Jupyter notebook. It contains animals with their expected lifespan:

```
animal, lifespan
dog, 12
cat, 14
pelican, 30
squirrel, 6
eagle, 25
```

We notice right away that the CSV is more structured than files we've seen in the previous section

- in the first line there are column names, separated with commas: `animal, lifespan`
- fields in successive rows are also separated by commas `,`: `dog, 12`

Let's try now to import this file in Python:

[20]:
```python
import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:

    # we create an object 'my_reader' which will take rows from the file
    my_reader = csv.reader(f, delimiter=',')

    # 'my_reader' is an object considered 'iterable', that is,
    # if used in a 'for' will produce a sequnce of rows from csv
    # NOTE: here every file row is converted into a list of Python strings !

    for row in my_reader:
        print('We just read a row !')
        print(row)  # prints variable 'row', which is a list of strings
        print('')   # prints an empty string, to separate in vertical
```

```
We just read a row !
['animal', ' lifespan']

We just read a row !
['dog', '12']

We just read a row !
['cat', '14']

We just read a row !
['pelican', '30']

We just read a row !
['squirrel', '6']

We just read a row !
['eagle', '25']
```

We immediatly notice from output that example file is being printed, but there are square parrenthesis (`[]`). What do they mean? Those we printed are *lists of strings*

Let's analyze what we did:

```python
import csv
```

Python natively has a module to deal with csv files, which has the intuitive `csv` name. With this instruction, we just loaded the module.

What happens next? As already did for files with lines before, we open the file in a `with` block:

```python
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    for row in my_reader:
        print(row)
```

For now ignore the `newline=''` and notice how first we specificed the encoding

Once the file is open, in the row

```python
my_reader = csv.reader(f, delimiter=',')
```

we ask to `csv` module to create a reader object called `my_reader` for our file, telling Python that comma is the delimiter for fields.

NOTE: `my_reader` is the name of the variable we are creating, it could be any name.

This reader object can be exploited as a sort of generator of rows by using a `for` cycle:

```
for row in my_reader:
    print(row)
```

In `for` cycle we employ `lettore` to iterate in the reading of the file, producing at each iteration a row we call `row` (but it could be any name we like). At each iteration, the variable `row` gets printed.

If you look closely the prints of first lists, you will see that each time to each row is assigned only one Python list. The list contains as many elements as the number of fields in the CSV.

⊕ **EXERCISE 2.3**: Rewrite in the cell below the instructions to read and print the CSV, paying attention to indentation:

[21]:
```python
import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:

    # we create an object 'my_reader' which will take rows from the file
    my_reader = csv.reader(f, delimiter=',')

    # 'my_reader' is an object considered 'iterable', that is,
    # if used in a 'for' will produce a sequnce of rows from csv
    # NOTE: here every file row is converted into a list of Python strings !

    for row in my_reader:
        print("We just read a row !")
        print(row)   # prints variable 'row', which is a list of strings
        print('')    # prints an empty string, to separate in vertical
```

```
We just read a row !
['animal', ' lifespan']

We just read a row !
['dog', '12']

We just read a row !
```

(continues on next page)

(continued from previous page)

```
['cat', '14']

We just read a row !
['pelican', '30']

We just read a row !
['squirrel', '6']

We just read a row !
['eagle', '25']
```

⊛⊛ **EXERCISE 2.4**: try to put into `big_list` a list containing all the rows extracted from the file, which will be a list of lists like so:

```
[['eagle', 'lifespan'],
 ['dog', '12'],
 ['cat', '14'],
 ['pelican', '30'],
 ['squirrel', '6'],
 ['eagle', '25']]
```

HINT: Try creating an empty list and then adding elements with `.append` method

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```python
# write here


import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:

    # we create an object 'my_reader' which will take rows from the file
    my_reader = csv.reader(f, delimiter=',')

    # 'my_reader' is an object considered 'iterable', that is,
    # if used in a 'for' will produce a sequnce of rows from csv
    # NOTE: here every file row is converted into a list of Python strings !

    big_list = []
    for row in my_reader:
        big_list.append(row)
    print(big_list)
```

```
[['animal', ' lifespan'], ['dog', '12'], ['cat', '14'], ['pelican', '30'], ['squirrel
↪', '6'], ['eagle', '25']]
```

</div>

[22]:
```python
# write here
```

```
[['animal', ' lifespan'], ['dog', '12'], ['cat', '14'], ['pelican', '30'], ['squirrel
↪', '6'], ['eagle', '25']]
```

⊛⊛ **EXERCISE 2.5**: You may have noticed that numbers in lists are represented as strings like `'12'` (note apeces), instead that like Python integer numbers (represented without apeces), `12`:

```
We just read a row!
['dog', '12']
```

So, by reading the file and using normal for cycles, try to create a new variable `big_list` like this, which

- has only data, the row with the header is not present
- numbers are represented as proper integers

```
[['dog', 12],
 ['cat', 14],
 ['pelican', 30],
 ['squirrel', 6],
 ['eagle', 25]]
```

HINT 1: to jump a row you can use the instruction `next(my_reader)`

HINT 2: to convert a string into an integer, you can use for example. `int('25')`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```python
# write here

import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    big_list = []
    next(my_reader)
    for row in my_reader:
        big_list.append([row[0], int(row[1])])
    print(big_list)
```

```
[['dog', 12], ['cat', 14], ['pelican', 30], ['squirrel', 6], ['eagle', 25]]
```

</div>

[23]:
```python
# write here
```

```
[['dog', 12], ['cat', 14], ['pelican', 30], ['squirrel', 6], ['eagle', 25]]
```

What's a reader ?

We said that `my_reader` generates a sequence of rows, and it is *iterable*. In `for` cycle, at every cycle we ask to read a new line, which is put into variable `row`. We might then ask ourselves, what happens if we directly print `my_reader`, without any `for`? Will we see a nice list or something else? Let's try:

[24]:
```python
import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    print(my_reader)
```

```
<_csv.reader object at 0x7fa700853978>
```

This result is quite disappointing

⊕ **EXERCISE 2.6**: you probably found yourself in the same situation when trying to print a sequence generated by a call to `range(5)`: instead of the actual sequence you get a `range` object. If you want to convert the generator to a list, what should you do?

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[25]:
```python
# write here

import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    print(list(my_reader))
```

```
[['animal', ' lifespan'], ['dog', '12'], ['cat', '14'], ['pelican', '30'], ['squirrel
↪', '6'], ['eagle', '25']]
```

</div>

[25]:
```python
# write here
```

```
[['animal', ' lifespan'], ['dog', '12'], ['cat', '14'], ['pelican', '30'], ['squirrel
↪', '6'], ['eagle', '25']]
```

Consuming a file

Not all sequences are the same. From what you've seen so far, going through a file in Python looks a lot like iterating a list. Which is very handy, but you need to pay attention to some things. Given that files potentially might occupy terabytes, basic Python functions to load them avoid loading everything into memory and typically a file is read one piece at a time. But if the whole file is loaded into Python environment in one shot, what happens if we try to go through it twice inside the same `with` ? What happens if we try using it outside `with`? To find out look at next exercises.

⊕ **EXERCISE 2.7**: taking the solution to previous exercise, try to call `print(list(my_reader))` twice, in sequence. Do you get the same output in both occasions?

[ ]:
```python

```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[26]:
```python
# write here the code

#import csv
#with open('example-1.csv', encoding='utf-8', newline='') as f:
#    my_reader = csv.reader(f, delimiter=',')
#    print(list(my_reader))
#    print(list(my_reader))
```

</div>

[26]:
```python
# write here the code
```

⊛ **EXERCISE 2.8**: Taking the solution from previous exercise (using only one print), try down here to move the print to the left (removing any spaces). Does it still work ?

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[27]:
```
# write here

import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
#print(list(my_reader))     # COMMENTED, AS IT WOULD RAISE ON ERROR OF CLOSED FILE
                            # We can't use commands which read the file outside the␣
↪with !
```

</div>

[27]:
```
# write here
```

⊛⊛ **EXERCISE 2.9**: Now that we understood which kind of beast `my_reader` is, try to produce this result as done before, but using a *list comprehension* instead of the `for`:

```
[['dog', 12],
 ['cat', 14],
 ['pelican', 30],
 ['squirrel', 6],
 ['eagle', 25]]
```

- If you can, try also to write the whole transformation to create `big_list` in one row, usinf the function itertools.islice[309] to jump the header (for example `itertools.islice(['A', 'B', 'C', 'D', 'E'], 2, None)` first two elements and produces the sequence C D E F G - in our case the elements produced by `my_reader` would be rows)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[28]:
```
import csv
import itertools
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    # write here
    big_list = [[row[0], int(row[1])] for row in itertools.islice(my_reader, 1, None)]
    print(big_list)
```

```
[['dog', 12], ['cat', 14], ['pelican', 30], ['squirrel', 6], ['eagle', 25]]
```

</div>

[28]:
```
import csv
import itertools
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.reader(f, delimiter=',')
    # write here
```

[309] https://docs.python.org/3/library/itertools.html#itertools.islice

```
[['dog', 12], ['cat', 14], ['pelican', 30], ['squirrel', 6], ['eagle', 25]]
```

⊕ **EXERCISE 2.10**: Create a file `my-example.csv` in the same folder where this Jupyter notebook is, and copy inside the content of the file `example-1.csv`. Then add a column `description`, remembering to separate the column name from the preceding one with a comma. As column values, put into successive rows strings like `dogs walk`, `pelicans fly`, etc according to the animal, remembering to separate them from lifespan using a comma, like this:

```
dog,12,dogs walk
```

After this, copy and paste down here the Python code to load the file, putting the file name `my-example.csv`, and try to load everything, just to check everything is working:

```
[29]: # write here
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER:

```
animal,lifespan,description
dog,12,dogs walk
cat,14,cats walk
pelican,30,pelicans fly
squirrel,6,squirrels fly
eagle,25,eagles fly
```

</div>

⊕ **Exercise 2.11**: Not every CSV is structured in the same way, sometimes when we write csvs or import them some tweak is necessary. Let's see which problems may arise:

- In the file, try to put one or two spaces before numbers, for example write down here and look what happens

```
dog, 12,dogs fly
```

QUESTION 2.11.1: Does the space get imported?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: yes

</div>

QUESTION 2.11.2: if we convert to integer, is the space a problem?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: no

</div>

QUESTION 2.11.3 Modify only dogs description from `dogs walk` to `dogs walk, but don't fly` and try to riexecute the cell which opens the file. What happens?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: Python reads one element more in the list

</div>

QUESTION 2.11.4: To overcome previous problem, a solution you can adopt in CSVs is to round strings containing commas with double quotes, like this: `"dogs walk, but don't fly"`. Does it work ?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: yes

</div>

Reading as dictionaries

To read a CSV, instead of getting lists, you may more conveniently get dictionaries in the form of `OrderedDicts`

See Python documentation[310]

NOTE: different Python versions give different dictionaries:

- $<$ 3.6: `dict`
- 3.6, 3.7: `OrderedDict`
- $\geq$ 3.8: `dict`

Python 3.8 returned to old `dict` because in the implementation of its dictionariesthe key order is guaranteed, so it will be consistent with the one of CSV headers

[30]:
```
import csv
with open('example-1.csv', encoding='utf-8', newline='') as f:
    my_reader = csv.DictReader(f, delimiter=',')   # Notice we now used DictReader
    for d in my_reader:
        print(d)
```

```
{' lifespan': '12', 'animal': 'dog'}
{' lifespan': '14', 'animal': 'cat'}
{' lifespan': '30', 'animal': 'pelican'}
{' lifespan': '6', 'animal': 'squirrel'}
{' lifespan': '25', 'animal': 'eagle'}
```

Writing a CSV

You can easily create a CSV by instantiating a `writer` object:

ATTENTION: BE SURE TO WRITE IN THE CORRECT FILE!

If you don't pay attention to file names, **you risk deleting data !**

[31]:
```
import csv

# To write, REMEMBER to specify the `w` option.
# WARNING: 'w' *completely* replaces existing files !!
```

(continues on next page)

310 https://docs.python.org/3/library/csv.html#csv.DictReader

(continued from previous page)

```
with open('written-file.csv', 'w', newline='') as csvfile_out:

    my_writer = csv.writer(csvfile_out, delimiter=',')

    my_writer.writerow(['This', 'is', 'a header'])
    my_writer.writerow(['some', 'example', 'data'])
    my_writer.writerow(['some', 'other', 'example data'])
```

Reading and writing a CSV

To create a copy of an existing CSV, you may nest a `with` for writing inside another for reading:

ATTENTION: CAREFUL NOT TO SWAP FILE NAMES!

When we read and write it's easy to make mistakes and accidentally overwrite our precious data.

To avoid issues:

- use explicit names both for output files (es: `example-1-enriched.csv` and handles (i.e. `csvfile_out`)
- backup data to read
- always check before carelessly executing code you just wrote !

[32]:
```
import csv

# To write, REMEMBER to specify the `w` option.
# WARNING: 'w' *completely* replaces existing files !!
# WARNING: handle here  is called *csvfile_out*
with open('example-1-enriched.csv', 'w', encoding='utf-8', newline='') as csvfile_out:
    my_writer = csv.writer(csvfile_out, delimiter=',')

    # Notice how this 'with' is *inside* the outer one:
    # WARNING: handle here is called *csvfile_in*
    with open('example-1.csv', encoding='utf-8', newline='') as csvfile_in:
        my_reader = csv.reader(csvfile_in, delimiter=',')

        for row in my_reader:
            row.append('something else')
            my_writer.writerow(row)
            my_writer.writerow(row)
            my_writer.writerow(row)
```

Let's see the new file was actually created by reading it:

[33]:
```
with open('example-1-enriched.csv', encoding='utf-8', newline='') as csvfile_in:
    my_reader = csv.reader(csvfile_in, delimiter=',')

    for row in my_reader:
        print(row)
```

```
['animal', ' lifespan', 'something else']
['animal', ' lifespan', 'something else']
['animal', ' lifespan', 'something else']
['dog', '12', 'something else']
```

(continues on next page)

(continued from previous page)

```
['dog', '12', 'something else']
['dog', '12', 'something else']
['cat', '14', 'something else']
['cat', '14', 'something else']
['cat', '14', 'something else']
['pelican', '30', 'something else']
['pelican', '30', 'something else']
['pelican', '30', 'something else']
['squirrel', '6', 'something else']
['squirrel', '6', 'something else']
['squirrel', '6', 'something else']
['eagle', '25', 'something else']
['eagle', '25', 'something else']
['eagle', '25', 'something else']
```

CSV Botteghe storiche

Usually in open data catalogs like the popular CKAN platform (for example dati.trentino.it[311], data.gov.uk[312], European data portal[313] run instances of CKAN) files are organized in *datasets*, which are collections of *resources*: each resource directly contains a file inside the catalog (typically CSV, JSON or XML) or a link to the real file located in a server belonging to the organizazion which created the data.

The first dataset we wil look at will be 'Botteghe storiche del Trentino':

https://dati.trentino.it/dataset/botteghe-storiche-del-trentino

Here you will find some generic information about the dataset, of importance note the data provider: Provincia Autonoma di Trento and the license Creative Commons Attribution v4.0[314], which basically allows any reuse provided you cite the author.

Inside the dataset page, there is a resource called 'Botteghe storiche'

https://dati.trentino.it/dataset/botteghe-storiche-del-trentino/resource/43fc327e-99b4-4fb8-833c-1807b5ef1d90

At the resource page, we find a link to the CSV file (you can also find it by clicking on the blue button 'Go to the resource'):

http://www.commercio.provincia.tn.it/binary/pat_commercio/valorizzazione_luoghi_storici/Albo_botteghe_storiche_in_ordine_iscrizione_9_5_2019.1557403385.csv

Accordingly to the browser and operating system you have, by clicking on the link above you might get different results. In our case, on browser Firefox and operating system Linux we get (here we only show first 10 rows):

```
Numero,Insegna,Indirizzo,Civico,Comune,Cap,Frazione/LocalitÃ ,Note
1,BAZZANELLA RENATA,Via del Lagorai,30,Sover,38068,Piscine di Sover,"generi misti,␣
↪bar - ristorante"
2,CONFEZIONI MONTIBELLER S.R.L.,Corso Ausugum,48,Borgo Valsugana,38051,,esercizio␣
↪commerciale
3,FOTOGRAFICA TRINTINAGLIA UMBERTO S.N.C.,Largo Dordi,8,Borgo Valsugana,38051,,
↪"esercizio commerciale, attivitÃ  artigianale"
4,BAR SERAFINI DI MINATI RENZO,,24,Grigno,38055,Serafini,esercizio commerciale
6,SEMBENINI GINO & FIGLI S.R.L.,Via S. Francesco,35,Riva del Garda,38066,,
7,HOTEL RISTORANTE PIZZERIA â€œALLA NAVEâ€,Via Nazionale,29,Lavis,38015,Nave San␣
↪Felice,
```

(continues on next page)

[311] http://dati.trentino.it/
[312] https://data.gov.uk/
[313] https://www.europeandataportal.eu/
[314] https://creativecommons.org/licenses/by/4.0/deed.en

(continued from previous page)

```
8,OBRELLI GIOIELLERIA DAL 1929 S.R.L.,Via Roma,33,Lavis,38015,,
9,MACELLERIE TROIER S.A.S. DI TROIER DARIO E C.,Via Roma,13,Lavis,38015,,
10,NARDELLI TIZIANO,Piazza Manci,5,Lavis,38015,,esercizio commerciale
```

As expected, values are separated with commas.

Problem: wrong characters ??

You can suddenly discover a problem in the first row of headers, in the column `Frazione/LocalitÃ`. It seems last character is wrong, in italian it should show accented like `à`. Is it truly a problem of the file ? Not really. Probably, the server is not telling Firefox which encoding is the correct one for the file. Firefox is not magical, and tries its best to show the CSV on the base of the info it has, which may be limited and / or even wrong. World is never like we would like it to be …

⊕ **2.12 EXERCISE**: download the CSV, and try opening it in Excel and / or LibreOffice Calc. Do you see a correct accented character? If not, try to set the encoding to 'Unicode (UTF-8)' (in Calc is called 'Character set').

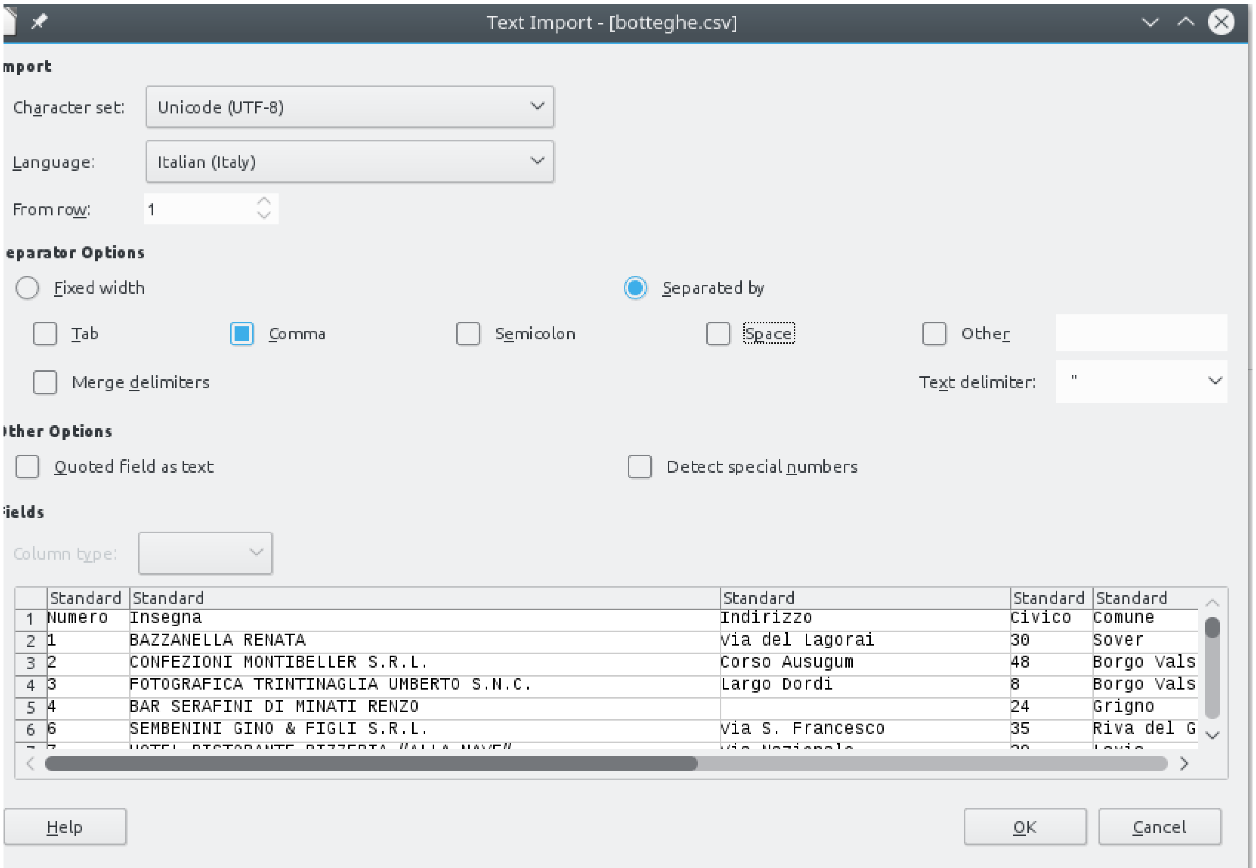
WARNING: CAREFUL IF YOU USE Excel!

By clicking directly on `File->Open in Excel`, probably Excel will try to guess on its own how to put the CSV in a table, and will make the mistake to place everything in a column. To avoid the problem, we have to tell Excel to show a panel to ask us how we want to open the CSV, by doing like so:

- In old Excels, find `File-> Import`
- In recent Excels, click on tab `Data` and then select `From text`. For further information, see copytrans guide[315]

- **NOTE**: If the file is not available, in the folder where this notebook is you will find the same file renamed to `botteghe-storiche.csv`

[315] https://www.copytrans.net/support/how-to-open-a-csv-file-in-excel/



We should get a table like this. Notice how the `Frazione/Località` header displays with the right accent because we selected Character set: Unicode (UTF-8) which is the appropriate one for this dataset:

	A	B	C	D	E	F	G	H
1	Numero	Insegna	Indirizzo	Civico	Comune	Cap	Frazione/Località	Note
2	1	BAZZANELLA RENATA	Via del Lagorai	30	Sover	38068	Piscine di Sover	generi misti, bar - ristorante
3	2	CONFEZIONI MONTIBELLER S.R.L.	Corso Ausugum	48	Borgo Valsugana	38051		esercizio commerciale
4	3	FOTOGRAFICA TRINTINAGLIA UMBERTO S.N.C.	Largo Dordi	8	Borgo Valsugana	38051		esercizio commerciale, attività artigianale
5	4	BAR SERAFINI DI MINATI RENZO		24	Grigno	38055	Serafini	esercizio commerciale
6	6	SEMBENINI GINO & FIGLI S.R.L.	Via S. Francesco	35	Riva del Garda	38066		
7	7	HOTEL RISTORANTE PIZZERIA “ALLA NAVE”	Via Nazionale	29	Lavis	38015	Nave San Felice	
8	8	OBRELLI GIOIELLERIA DAL 1929 S.R.L.	Via Roma	33	Lavis	38015		
9	9	MACELLERIE TROIER S.A.S. DI TROIER DARIO E C.	Via Roma	13	Lavis	38015		
10	10	NARDELLI TIZIANO	Piazza Manci	5	Lavis	38015		esercizio commerciale

Botteghe storiche in Python

Now that we understood a couple of things about encoding, let’s try to import the file in Python.

If we load in Python the first 5 entries with a csv DictReader and print them we should see something like this:

```
OrderedDict([('Numero', '1'),
              ('Insegna', 'BAZZANELLA RENATA'),
              ('Indirizzo', 'Via del Lagorai'),
              ('Civico', '30'),
              ('Comune', 'Sover'),
              ('Cap', '38068'),
              ('Frazione/Località', 'Piscine di Sover'),
              ('Note', 'generi misti, bar - ristorante')]),
OrderedDict([('Numero', '2'),
             ('Insegna', 'CONFEZIONI MONTIBELLER S.R.L.'),
             ('Indirizzo', 'Corso Ausugum'),
```

(continues on next page)

(continued from previous page)

```
            ('Civico', '48'),
            ('Comune', 'Borgo Valsugana'),
            ('Cap', '38051'),
            ('Frazione/Località', ''),
            ('Note', 'esercizio commerciale')]),
OrderedDict([('Numero', '3'),
            ('Insegna', 'FOTOGRAFICA TRINTINAGLIA UMBERTO S.N.C.'),
            ('Indirizzo', 'Largo Dordi'),
            ('Civico', '8'),
            ('Comune', 'Borgo Valsugana'),
            ('Cap', '38051'),
            ('Frazione/Località', ''),
            ('Note', 'esercizio commerciale, attività artigianale')]),
OrderedDict([('Numero', '4'),
            ('Insegna', 'BAR SERAFINI DI MINATI RENZO'),
            ('Indirizzo', ''),
            ('Civico', '24'),
            ('Comune', 'Grigno'),
            ('Cap', '38055'),
            ('Frazione/Località', 'Serafini'),
            ('Note', 'esercizio commerciale')]),
OrderedDict([('Numero', '6'),
            ('Insegna', 'SEMBENINI GINO & FIGLI S.R.L.'),
            ('Indirizzo', 'Via S. Francesco'),
            ('Civico', '35'),
            ('Comune', 'Riva del Garda'),
            ('Cap', '38066'),
            ('Frazione/Località', ''),
            ('Note', '')])
```

We would like to know which different categories of *bottega* there are, and count them. Unfortunately, there is no specific field for *Categoria*, so we will need to extract this information from other fields such as `Insegna` and `Note`. For example, this `Insegna` contains the category `BAR`, while the `Note` (*commercial enterprise*) is a bit too generic to be useful:

```
'Insegna': 'BAR SERAFINI DI MINATI RENZO',
'Note': 'esercizio commerciale',
```

while this other `Insegna` contains just the owner name and `Note` holds both the categories `bar` and `ristorante`:

```
'Insegna': 'BAZZANELLA RENATA',
'Note': 'generi misti, bar - ristorante',
```

As you see, data is non uniform:

- sometimes the category is in the `Insegna`
- sometimes is in the `Note`
- sometimes is in both
- sometimes is lowercase
- sometimes is uppercase
- sometimes is single
- sometimes is multiple (`bar - ristorante`)

First we want to extract all categories we can find, and rank them according their frequency, from most frequent to least frequent.

To do so, you need to

- count all words you can find in both `Insegna` and `Note` fields, and sort them. Note you need to normalize the uppercase.
- consider a category relevant if it is present at least 11 times in the dataset.
- filter non relevant words: some words like prepositions, type of company (`'S.N.C'`, `S.R.L.`, ..), etc will appear a lot, and will need to be ignored. To detect them, you are given a list called `stopwords`.

NOTE: the rules above do not actually extract all the categories, for the sake of the exercise we only keep the most frequent ones.

To know how to proceed, read the following.

Botteghe storiche - rank_categories

Load the file with `csv.DictReader` and while you are loading it, calculate the words as described above. Afterwards, return a list of words with their frequencies.

Do **not** load the whole file into memory, just process one dictionary at a time and update statistics accordingly.

Expected output:

```
[('BAR', 191),
 ('RISTORANTE', 150),
 ('HOTEL', 67),
 ('ALBERGO', 64),
 ('MACELLERIA', 27),
 ('PANIFICIO', 22),
 ('CALZATURE', 21),
 ('FARMACIA', 21),
 ('ALIMENTARI', 20),
 ('PIZZERIA', 16),
 ('SPORT', 16),
 ('TABACCHI', 12),
 ('FERRAMENTA', 12),
 ('BAZAR', 11)]
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[34]:
```python
def rank_categories(stopwords):

    ret = {}
    import csv
    with open('botteghe.csv', newline='',  encoding='utf-8',) as csvfile:
        reader = csv.DictReader(csvfile,  delimiter=',')
        for d in reader:
            words = d['Insegna'].split(" ") + d['Note'].upper().split(" ")
            for word in words:
                if word in ret and not word in stopwords:
                    ret[word] += 1
                else:
                    ret[word] = 1
    return sorted([(key, val) for key,val in ret.items() if val > 10], key=lambda c:␣
↪c[1], reverse=True)
```

(continues on next page)

(continued from previous page)

```
stopwords = ['',
             'S.N.C.', 'SNC','S.A.S.', 'S.R.L.', 'S.C.A.R.L.', 'SCARL','S.A.S',
↪'COMMERCIALE','FAMIGLIA','COOPERATIVA',
             '-', '&', 'C.', 'ESERCIZIO',
             'IL', 'DE', 'DI','A', 'DA', 'E', 'LA', 'AL',  'DEL', 'ALLA', ]
categories = rank_categories(stopwords)

categories
```

[34]:
```
[('BAR', 191),
 ('RISTORANTE', 150),
 ('HOTEL', 67),
 ('ALBERGO', 64),
 ('MACELLERIA', 27),
 ('PANIFICIO', 22),
 ('FARMACIA', 21),
 ('CALZATURE', 21),
 ('ALIMENTARI', 20),
 ('SPORT', 16),
 ('PIZZERIA', 16),
 ('FERRAMENTA', 12),
 ('TABACCHI', 12),
 ('BAZAR', 11)]
```

</div>

[34]:
```
def rank_categories(stopwords):
    raise Exception('TODO IMPLEMENT ME !')

stopwords = ['',
             'S.N.C.', 'SNC','S.A.S.', 'S.R.L.', 'S.C.A.R.L.', 'SCARL','S.A.S',
↪'COMMERCIALE','FAMIGLIA','COOPERATIVA',
             '-', '&', 'C.', 'ESERCIZIO',
             'IL', 'DE', 'DI','A', 'DA', 'E', 'LA', 'AL',  'DEL', 'ALLA', ]
categories = rank_categories(stopwords)

categories
```

[34]:
```
[('BAR', 191),
 ('RISTORANTE', 150),
 ('HOTEL', 67),
 ('ALBERGO', 64),
 ('MACELLERIA', 27),
 ('PANIFICIO', 22),
 ('FARMACIA', 21),
 ('CALZATURE', 21),
 ('ALIMENTARI', 20),
 ('SPORT', 16),
 ('PIZZERIA', 16),
 ('FERRAMENTA', 12),
 ('TABACCHI', 12),
 ('BAZAR', 11)]
```

Botteghe storiche - enrich

Once you found the categories, implement function `enrich`, which takes the db and previously computed categories, and WRITES a NEW file `botteghe-enriched.csv` where the rows are enriched with a new field `Categorie`, which holds a list of the categories a particular *bottega* belongs to.

- Write the new file with a `DictWriter`, see documentation[316]

The new file should contain rows like this (showing only first 5):

```
OrderedDict([   ('Numero', '1'),
                ('Insegna', 'BAZZANELLA RENATA'),
                ('Indirizzo', 'Via del Lagorai'),
                ('Civico', '30'),
                ('Comune', 'Sover'),
                ('Cap', '38068'),
                ('Frazione/Località', 'Piscine di Sover'),
                ('Note', 'generi misti, bar - ristorante'),
                ('Categorie', "['BAR', 'RISTORANTE']")])
OrderedDict([   ('Numero', '2'),
                ('Insegna', 'CONFEZIONI MONTIBELLER S.R.L.'),
                ('Indirizzo', 'Corso Ausugum'),
                ('Civico', '48'),
                ('Comune', 'Borgo Valsugana'),
                ('Cap', '38051'),
                ('Frazione/Località', ''),
                ('Note', 'esercizio commerciale'),
                ('Categorie', '[]')])
OrderedDict([   ('Numero', '3'),
                ('Insegna', 'FOTOGRAFICA TRINTINAGLIA UMBERTO S.N.C.'),
                ('Indirizzo', 'Largo Dordi'),
                ('Civico', '8'),
                ('Comune', 'Borgo Valsugana'),
                ('Cap', '38051'),
                ('Frazione/Località', ''),
                ('Note', 'esercizio commerciale, attività artigianale'),
                ('Categorie', '[]')])
OrderedDict([   ('Numero', '4'),
                ('Insegna', 'BAR SERAFINI DI MINATI RENZO'),
                ('Indirizzo', ''),
                ('Civico', '24'),
                ('Comune', 'Grigno'),
                ('Cap', '38055'),
                ('Frazione/Località', 'Serafini'),
                ('Note', 'esercizio commerciale'),
                ('Categorie', "['BAR']")])
OrderedDict([   ('Numero', '6'),
                ('Insegna', 'SEMBENINI GINO & FIGLI S.R.L.'),
                ('Indirizzo', 'Via S. Francesco'),
                ('Civico', '35'),
                ('Comune', 'Riva del Garda'),
                ('Cap', '38066'),
                ('Frazione/Località', ''),
                ('Note', ''),
                ('Categorie', '[]')])
```

<a class="jupman-sol jupman-sol-toggler" onclick="jupman.toggleSolution(this);" data-jupman-show="Show solution"

[316] https://docs.python.org/3/library/csv.html#csv.DictWriter

data-jupman-hide="Hide">Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[35]:
```python
def enrich(categories):

    ret = []


    fieldnames = []
    # read headers
    with open('botteghe.csv', newline='',  encoding='utf-8') as csvfile_in:
        reader = csv.DictReader(csvfile_in,  delimiter=',')
        d1 = next(reader)
        fieldnames = list(d1.keys())  # otherwise we cannot append

    fieldnames.append('Categorie')

    with open('botteghe-enriched-solution.csv', 'w', newline='', encoding='utf-8') as␣
↪csvfile_out:

        writer = csv.DictWriter(csvfile_out, fieldnames=fieldnames)
        writer.writeheader()

        with open('botteghe.csv', newline='',  encoding='utf-8',) as csvfile_in:
            reader = csv.DictReader(csvfile_in,  delimiter=',')
            for d in reader:

                new_d = {key:val for key,val in d.items()}
                new_d['Categorie'] = []
                for cat in categories:
                    if cat[0] in d['Insegna'].upper() or cat[0] in d['Note'].upper():
                        new_d['Categorie'].append(cat[0])
                writer.writerow(new_d)



enrich(rank_categories(stopwords))
```

</div>

[35]:
```python
def enrich(categories):
    raise Exception('TODO IMPLEMENT ME !')

enrich(rank_categories(stopwords))
```

[36]:
```python
# let's see if we created the file we wanted
# (using botteghe-enriched-solution.csv to avoid polluting your file)

with open('botteghe-enriched-solution.csv', newline='',  encoding='utf-8',) as␣
↪csvfile_in:
    reader = csv.DictReader(csvfile_in,  delimiter=',')
    # better to pretty print the OrderedDicts, otherwise we get unreadable output
    # for documentation see https://docs.python.org/3/library/pprint.html
    import pprint
    pp = pprint.PrettyPrinter(indent=4)
```

(continues on next page)

(continued from previous page)

```python
    for i in range(5):
        d = next(reader)
        pp.pprint(d)
```

```
{   'Cap': '38068',
    'Categorie': "['BAR', 'RISTORANTE']",
    'Civico': '30',
    'Comune': 'Sover',
    'Frazione/Località': 'Piscine di Sover',
    'Indirizzo': 'Via del Lagorai',
    'Insegna': 'BAZZANELLA RENATA',
    'Note': 'generi misti, bar - ristorante',
    'Numero': '1'}
{   'Cap': '38051',
    'Categorie': '[]',
    'Civico': '48',
    'Comune': 'Borgo Valsugana',
    'Frazione/Località': '',
    'Indirizzo': 'Corso Ausugum',
    'Insegna': 'CONFEZIONI MONTIBELLER S.R.L.',
    'Note': 'esercizio commerciale',
    'Numero': '2'}
{   'Cap': '38051',
    'Categorie': '[]',
    'Civico': '8',
    'Comune': 'Borgo Valsugana',
    'Frazione/Località': '',
    'Indirizzo': 'Largo Dordi',
    'Insegna': 'FOTOGRAFICA TRINTINAGLIA UMBERTO S.N.C.',
    'Note': 'esercizio commerciale, attività artigianale',
    'Numero': '3'}
{   'Cap': '38055',
    'Categorie': "['BAR']",
    'Civico': '24',
    'Comune': 'Grigno',
    'Frazione/Località': 'Serafini',
    'Indirizzo': '',
    'Insegna': 'BAR SERAFINI DI MINATI RENZO',
    'Note': 'esercizio commerciale',
    'Numero': '4'}
{   'Cap': '38066',
    'Categorie': '[]',
    'Civico': '35',
    'Comune': 'Riva del Garda',
    'Frazione/Località': '',
    'Indirizzo': 'Via S. Francesco',
    'Insegna': 'SEMBENINI GINO & FIGLI S.R.L.',
    'Note': '',
    'Numero': '6'}
```

6.1.5 3. JSON files

JSON is a more elaborated format, widely used in the world of web applications.

A json is simply a text file, structured as *a tree*. Let's see an example, extracted from the data Bike sharing stations of Lavis municipality as found on dati.trentino :

- Data source: dati.trentino.it[317] - Trasport Service of the Autonomous Province of Trento
- License: CC-BY 4.0[318]

File bike-sharing-lavis.json:

```
[
  {
    "name": "Grazioli",
    "address": "Piazza Grazioli - Lavis",
    "id": "Grazioli - Lavis",
    "bikes": 3,
    "slots": 7,
    "totalSlots": 10,
    "position": [
      46.139732902099794,
      11.111516155225331
    ]
  },
  {
    "name": "Pressano",
    "address": "Piazza della Croce - Pressano",
    "id": "Pressano - Lavis",
    "bikes": 2,
    "slots": 5,
    "totalSlots": 7,
    "position": [
      46.15368174037716,
      11.106601229430453
    ]
  },
  {
    "name": "Stazione RFI",
    "address": "Via Stazione - Lavis",
    "id": "Stazione RFI - Lavis",
    "bikes": 4,
    "slots": 6,
    "totalSlots": 10,
    "position": [
      46.148180371138814,
      11.096753997622727
    ]
  }
]
```

As you can see, the json format is very similar to data structures we already have in Python, such as strings, integer numbers, floats, lists and dictionaries. The only difference are the json `null` fields which become `None` in Python. So the conversion to Python is almost always easy and painless, to perform it you can use the native Python module called `json` by calling the function `json.load`, which interprets the json text file and converts it to a Python data structure:

[317] https://dati.trentino.it/dataset/stazioni-bike-sharing-emotion-trentino
[318] http://creativecommons.org/licenses/by/4.0/deed.it

```
[37]: import json

      with open('bike-sharing-lavis.json',  encoding='utf-8') as f:
          python_content = json.load(f)

      print(python_content)
```

```
[{'address': 'Piazza Grazioli - Lavis', 'totalSlots': 10, 'name': 'Grazioli', 'bikes':
↪ 3, 'slots': 7, 'id': 'Grazioli - Lavis', 'position': [46.139732902099794, 11.
↪111516155225331]}, {'address': 'Piazza della Croce - Pressano', 'totalSlots': 7,
↪'name': 'Pressano', 'bikes': 2, 'slots': 5, 'id': 'Pressano - Lavis', 'position':␣
↪[46.15368174037716, 11.106601229430453]}, {'address': 'Via Stazione - Lavis',
↪'totalSlots': 10, 'name': 'Stazione RFI', 'bikes': 4, 'slots': 6, 'id': 'Stazione␣
↪RFI - Lavis', 'position': [46.148180371138814, 11.096753997622727]}]
```

Notice that what we've just read with the function `json.load` is not simple text anymore, but Python objects. For this json, the most external object is a list (note the square brackets at the file beginning and end). We can check using `type` on `python_content`:

```
[38]: type(python_content)
```

```
[38]: list
```

By looking at the JSON closely, you will see it is a list of dictionaries. Thus, to access the first dictionary (that is, the one at zero-th index), we can write

```
[39]: python_content[0]
```

```
[39]: {'address': 'Piazza Grazioli - Lavis',
       'bikes': 3,
       'id': 'Grazioli - Lavis',
       'name': 'Grazioli',
       'position': [46.139732902099794, 11.111516155225331],
       'slots': 7,
       'totalSlots': 10}
```

We see it's the station in Piazza Grazioli. To get the exact name, we will access the `'address'` key in the first dictionary:

```
[40]: python_content[0]['address']
```

```
[40]: 'Piazza Grazioli - Lavis'
```

To access the position, we will use the corresponding key:

```
[41]: python_content[0]['position']
```

```
[41]: [46.139732902099794, 11.111516155225331]
```

Note how the position is a list itself. In JSON we can have arbitrarily branched trees, without necessarily a regular structure (althouth when we're generating a json it certainly helps maintaining a regualar data scheme).

JSONL

There is a particular JSON file type which is called JSONL[319] (note the *L* at the end), which is a text file containing a sequence of lines, each representing a valid json object.

Let's have a look at the file employees.jsonl:

```
{"name": "Mario", "surname":"Rossi"}
{"name": "Paolo", "surname":"Bianchi"}
{"name": "Luca", "surname":"Verdi"}
```

To read it, we can open the file, separating the text lines and then interpret each of them as a single JSON object:

[42]:
```python
import json

with open('./employees.jsonl', encoding='utf-8',) as f:
    json_texts_list = list(f)       #  converts file text lines into a Python list


# in this case we will have a python content for each row of the original file

i = 0
for json_text in json_texts_list:
    python_content = json.loads(json_text)    # converts json text to a python object
    print('Object ', i)
    print(python_content)
    i = i + 1
```

```
Object  0
{'name': 'Mario', 'surname': 'Rossi'}
Object  1
{'name': 'Paolo', 'surname': 'Bianchi'}
Object  2
{'name': 'Luca', 'surname': 'Verdi'}
```

[ ]:

6.2 Graph formats solutions

6.2.1 Download exercises zip

Browse files online[320]

[319] http://jsonlines.org/
[320] https://github.com/DavidLeoni/softpython-en/tree/master/graph-formats

6.2.2 Introduction

Usual matrices from linear algebra are of great importance in computer science because they are widely used in many fields, for example in machine learning and network analysis. This tutorial will give you an appreciation of the meaning of matrices when considered as networks or, as we call them in computer science, *graphs*. We will also review other formats for storing graphs, such as *adjacency lists* and a have a quick look at a specialized library called Networkx.

What to do

- unzip exercises in a folder, you should get something like this:

```
graph-formats
    graph-formats.ipynb
    graph-formats-sol.ipynb
    jupman.py
    soft.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

- open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `graph-formats/graph-formats.ipynb`

WARNING 2: DO NOT use the *Upload* button in Jupyter, instead navigate in Jupyter browser to the unzipped folder !

- Go on reading that notebook, and follow instuctions inside.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

Required libraries

In order for visualizations to work, you need installed the python library `networkx` and `pydot`. Pydot is an interface to the non-pyhon package GraphViz[321].

Anaconda:

From Anaconda Prompt:

1. Install GraphViz:

```
conda install graphviz
```

2. Install python packages:

```
conda install pydot networkx
```

[321] http://graphviz.org/

Ubuntu

From console:

1. Install PyGraphViz (note: you should use apt to install it, pip might give problems):

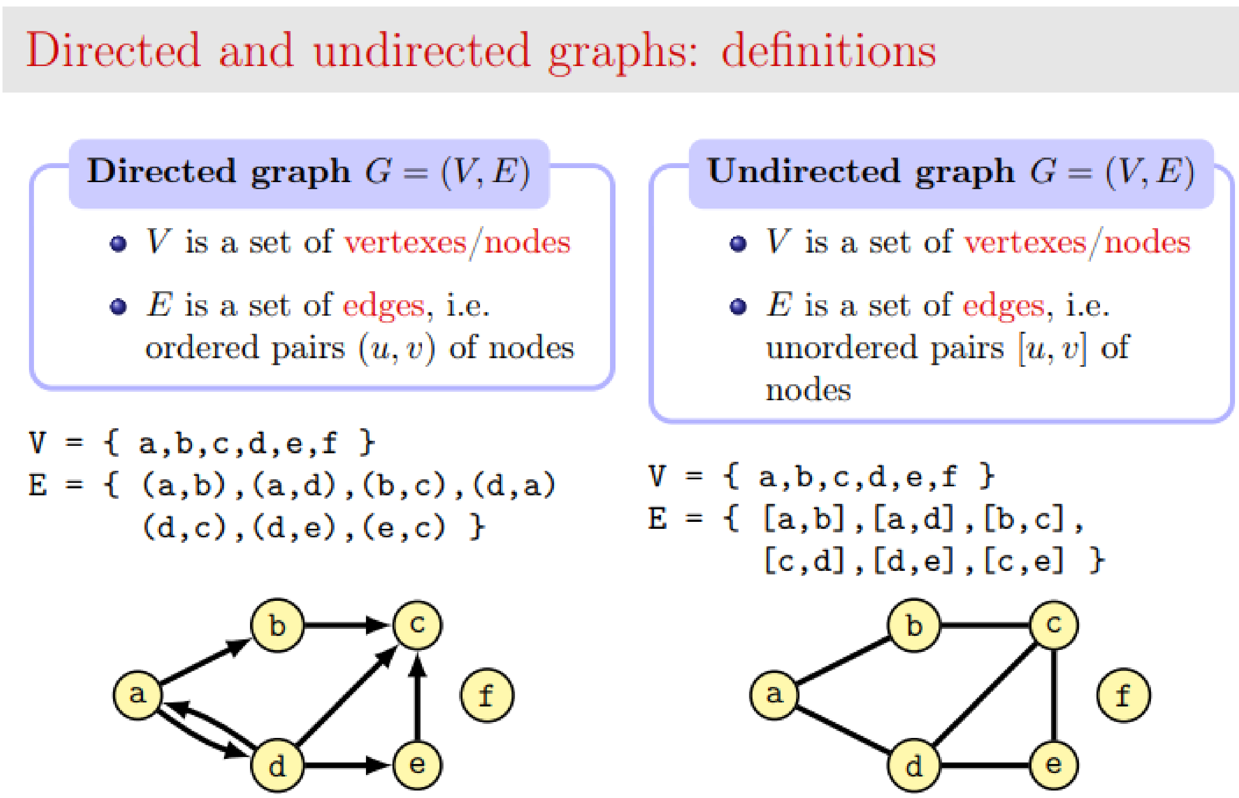
```
sudo apt install python3-pygraphviz
```

2. Install python packages:

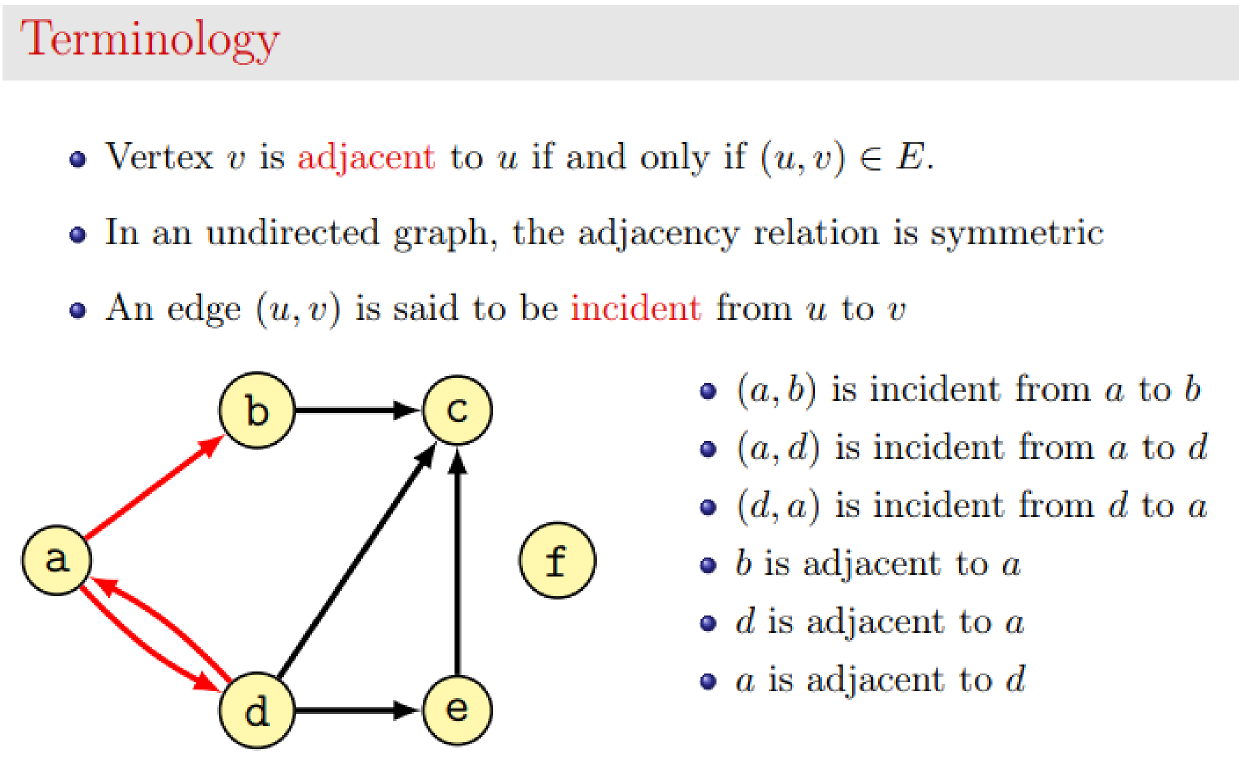
```
python3 -m pip install --user pydot networkx
```

Graph definition

In computer science a *graph* is a set of verteces V (also called *nodes)* linked by a set of edges E. You can visualize nodes as circles and links as lines. If the graph is *undirected,* links are just lines, if the graph is *directed,* links are represented as arrows with a tip to show the direction:



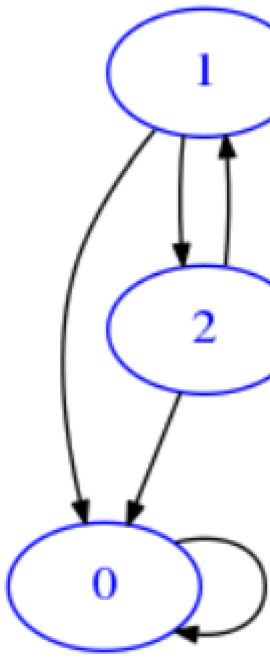
Credits: slide by Dr Alberto Montresor



Credits: slide by Dr Alberto Montresor

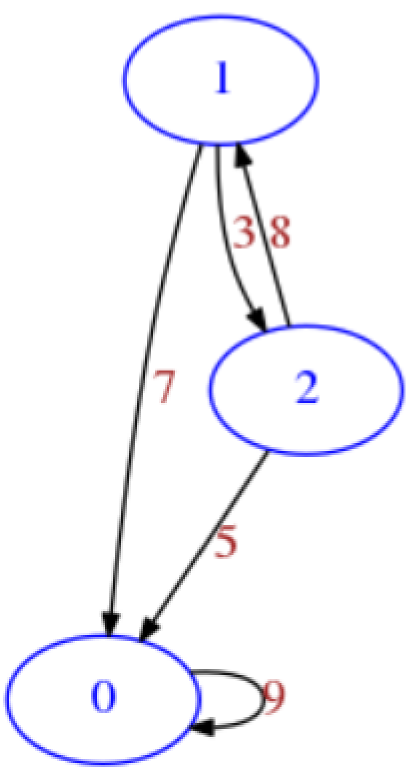
For our purposes, we will consider directed graphs (also called *digraphs).*

Usually we will indicate nodes with numbers going from zero included but optionally they can be labelled. Since we are dealing with directed graphs, we can have an arrow going for example from node `1` to node `2`, but also another arrow going from node `2` to node `1`. Furthemore, a node (for example node `0`) can have a *cap*, that is an edge going to itself:



Edge weights

Optionally, we will sometimes assign a *weight* to the edges, that is a number to be shown over the edges. So we can modify the previous example. Note we can have an arrow going from node `1` to node `2` with a weight which is different from the weight arrow from `2` to `1`:



6.2.3 Matrices

Here we will represent graphs as matrices, which performance-wise is particularly good when the matrix is *dense,* that is, has many entries different from zero. Otherwise, when you have a so-called *sparse* matrix (few non-zero entries), it is best to represent the graph with *adjacency list,* but we will deal with them later.

If you have a directed graph (digraph) with `n` verteces, you can represent it as an `n x n` matrix by considering each row as vertex:

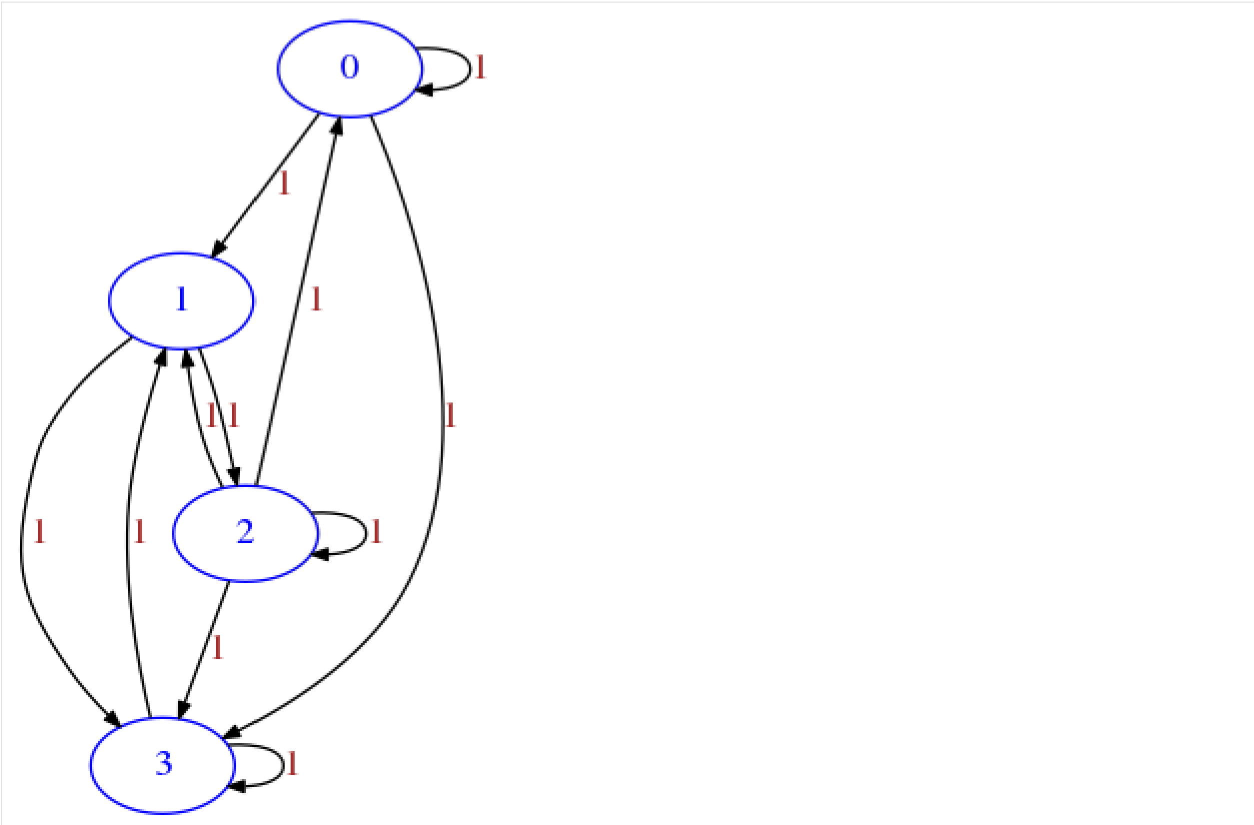
- A row at index `i` represents the outward links from node `i` to the other `n` nodes, with possibly node `i` itself included.
- A value of zero means there is no link to a given node.
- In general, `mat[i][j]` is the weight of the edge between node `i` to node `j`

Visualization examples

We defined a function `soft.draw_mat`to display matrices as graphs (you don't need to understand the internals, for now we won't go into depth about matrix visualizations).

If it doesn't work, see above *Required libraries paragraph*

[2]:
```
# PLEASE EXECUTE THIS CELL TO CHECK IF VISUALIZATION IS WORKING

# notice links with weight zero are not shown)
# all weights are set to 1

# first need to import this
from soft import draw_mat

mat = [
    [1,1,0,1],  # node 0 is linked to node 0 itself, node 1 and node 2
    [0,0,1,1],  # node 1 is linked to node 2 and node 3
    [1,1,1,1],  # node 2 is linked to node 0, node 1, node 2 itself and node 3
    [0,1,0,1]   # node 3 is linked to node 1 and node 3 itself
  ]


draw_mat(mat)
```

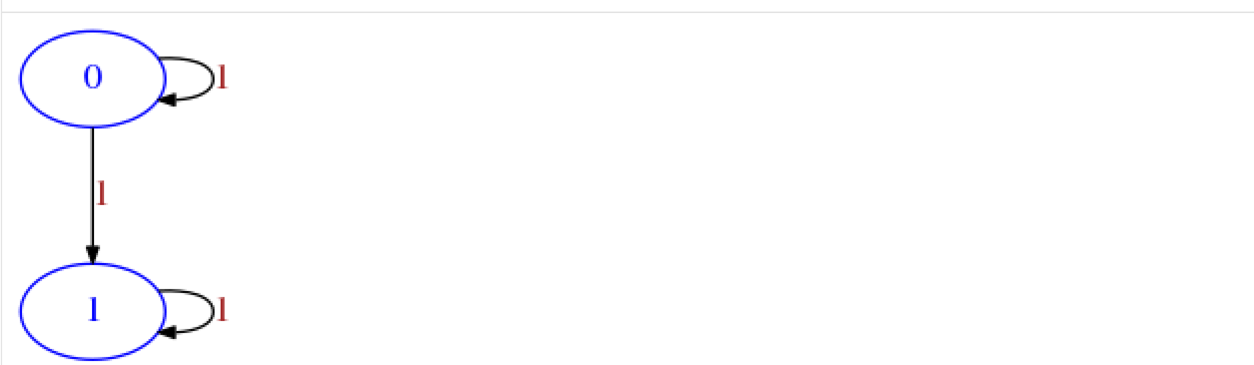


Saving a graph to an image file

If you want (or if you are not using Jupyter), optionally you can save the graph to a `.png` file by specificing the `save_to` filepath:

[3]:
```
mat = [
        [1,1],
        [0,1]
      ]
draw_mat( mat, save_to='example.png')
```

```
Image saved to file:  example.png
```



Saving a graph to an dot file

You can also save a graph to the original *dot* language of GraphViz:

```
[4]: mat = [
             [1,1],
             [0,1]
         ]
     draw_mat( mat, save_to='example.dot')
```

```
Dot saved to file:  example.dot
```

Note no visualization occurs, as you probably might need this kind of output when GraphViz is not installed in your system and you want to display the file elsewhere.

There are lots of websites that take `.dot` and output images, for example https://dreampuf.github.io/GraphvizOnline

We output here the file content, try to copy/paste it in the above website:

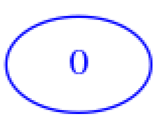
```
[5]: with open('example.dot') as f:
         print(f.read())
```

```
digraph  {
scale=3;
node [fontcolor=blue, color=blue];
edge [splines=curved, fontcolor=brown, arrowsize="0.6"];
0;
1;
0 -> 0  [weight=1, label=1];
0 -> 1  [weight=1, label=1];
1 -> 1  [weight=1, label=1];
}
```

Minimal graph

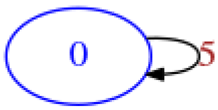
With this representation derived from matrices as we intend them (that is with at least one row and one column), the corresponding minimal graph can have only one node:

```
[6]: minimal = [
         [0]
     ]

     draw_mat(minimal)
```



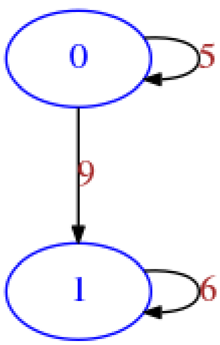
If we set the weight different from zero, the zeroeth node will link to itself (here we put the weight 5 in the link):

```
[7]: minimal = [
         [5]
     ]

     draw_mat(minimal)
```



Graph with two nodes example

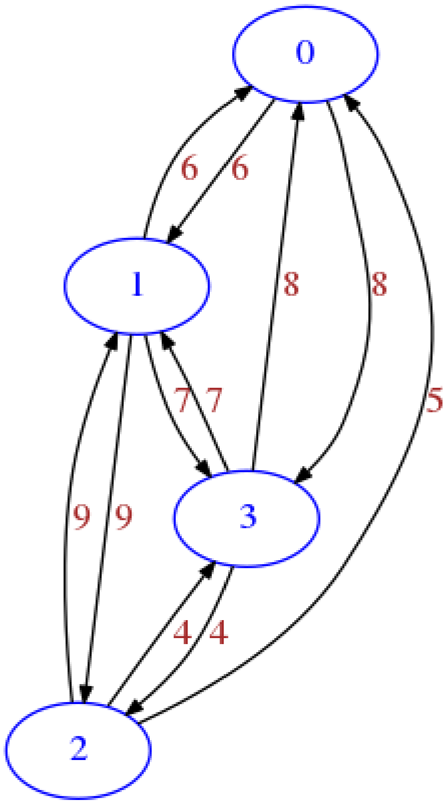
[8]:
```
m = [
    [5,9], # node 0 links to node 0 itself with a weight of 5, and to node 1 with a weight of 9
    [0,6], # node 1 links to node 1 with a weight of 6

]

draw_mat(m)
```



Distance matrix

Depending on the problem at hand, it may be reasonable to change the weights. For example, on a road network the nodes could represent places and the weights could be the distances. If we assume it is possible to travel in both directions on all roads, we get a matrix symmetric along the diagonal, and we can call the matrix a *distance matrix*. Talking about the diagonal, for the special case of going from a place to itself, we set that street length to 0 (which make sense for street length but could give troubles for other purposes, for example if we give the numbers the meaning 'is connected' a place should always be connected to itself)

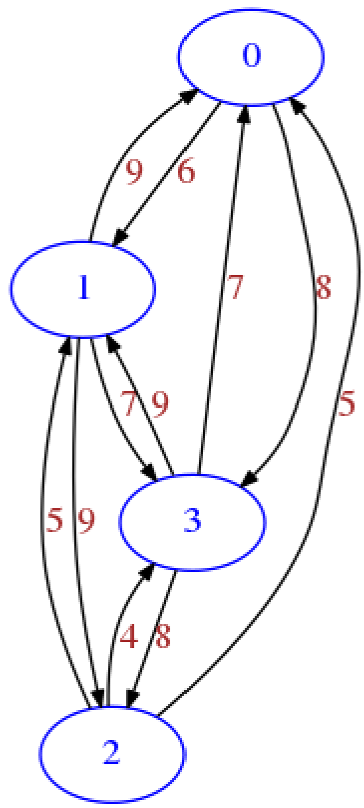
[9]:
```
# distance matrix example

mat = [
        [0,6,0,8],   # place 0 is linked to place 1 and place 2
        [6,0,9,7],   # place 1 is linked to place 0, place 2 and place 3
        [5,9,0,4],   # place 2 is linked to place 0, place 1 and place 3
        [8,7,4,0]    # place 3 is linked to place 0, place 1 and place 2
      ]


draw_mat(mat)
```



More realistic traffic road network, where going in one direction might take actually longer than going back, because of one-way streets and different routing times.

[10]:

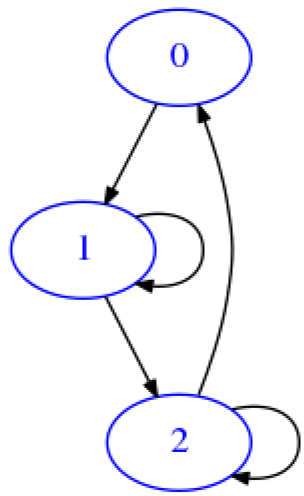
```
mat = [
        [0,6,0,8],   # place 0 is linked to place 1 and place 2
        [9,0,9,7],   # place 1 is linked to place 0, place 2 and place 3
        [5,5,0,4],   # place 2 is linked to place 0, place 1 and place 3
        [7,9,8,0]    # place 3 is linked to place 0, place 1, place 2
      ]


draw_mat(mat)
```



Boolean matrix example

If we are not interested at all in the weights, we might use only zeroes and ones as we did before. But this could have implications when doing operations on matrices, so some times it is better to use only `True` and `False`

[11]:
```
mat = [
    [False, True, False],
    [False, True, True],
    [True, False, True],

]
draw_mat(mat)
```



Matrix exercises

We are now ready to start implementing the following functions. Before even start implementation, for each try to interpret the matrix as a graph, drawing it on paper. When you're done implementing try to use `draw_mat` on the results. Notice that since `draw_mat` is a generic display function and knows nothing about the nature of the graph, sometimes it will not show the graph in the optimal way we humans would use.

Exercise - line

⊛⊛ This function is similar to `diag`. As that one, you can implement it in two ways: you can use a double `for`, or a single one (much more efficient). What would be the graph representation of `line` ?

RETURN a matrix as lists of lists where node `i` must have an edge to node `i + 1` with weight `1`

- Last node points to nothing
- `n` must be `>= 1`, otherwise raises `ValueError`

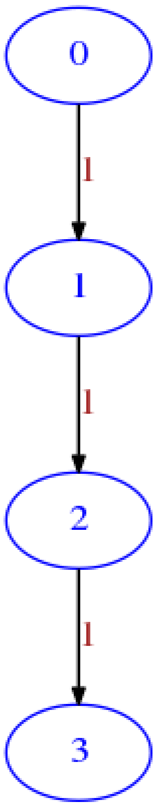
Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[12]:
```python
def line(n):

    if n < 1:
        raise ValueError("Invalid n %s" % n)
    ret = [[0]*n for i in range(n)]
    for i in range(n-1):
        ret[i][i+1] = 1
    return ret


assert line(1) == [ [0] ]

assert line(2) == [ [0,1],
                    [0,0] ]
assert line(3) == [ [0,1,0],
                    [0,0,1],
```

(continues on next page)

(continued from previous page)

```
                    [0,0,0] ]

assert line(4) == [ [0,1,0,0],
                    [0,0,1,0],
                    [0,0,0,1],
                    [0,0,0,0] ]
draw_mat(line(4))
```



</div>

[12]:

```
def line(n):
    raise Exception('TODO IMPLEMENT ME !')

assert line(1) == [ [0] ]

assert line(2) == [ [0,1],
                    [0,0] ]
assert line(3) == [ [0,1,0],
                    [0,0,1],
                    [0,0,0] ]

assert line(4) == [ [0,1,0,0],
                    [0,0,1,0],
                    [0,0,0,1],
                    [0,0,0,0] ]
draw_mat(line(4))
```

Exercise - cross

⊛⊛ RETURN a nxn matrix filled with zeros except on the crossing lines.

- `n` must be >=1 and odd, otherwise a `ValueError` is thrown

Example for `n=7` :

```
0001000
0001000
0001000
1111111
0001000
0001000
0001000
```

Try to figure out how the resulting graph would look like (try to draw on paper, also notice that `draw_mat` will probably not draw the best possible representation)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[13]:
```python
def cross(n):

    if n < 1 or n % 2 == 0:
        raise ValueError("Invalid n %s" % n)
    ret = [[0]*n for i in range(n)]
    for i in range(n):
```

(continues on next page)

(continued from previous page)

```
        ret[n//2 ][i] = 1
        ret[i][n//2] = 1
    return ret


assert cross(1) == [
    [1]
]
assert cross(3) == [ [0,1,0],
                     [1,1,1],
                     [0,1,0] ]

assert cross(5) == [ [0,0,1,0,0],
                     [0,0,1,0,0],
                     [1,1,1,1,1],
                     [0,0,1,0,0],
                     [0,0,1,0,0] ]
```

</div>

[13]:
```
def cross(n):
    raise Exception('TODO IMPLEMENT ME !')

assert cross(1) == [
    [1]
]
assert cross(3) == [ [0,1,0],
                     [1,1,1],
                     [0,1,0] ]

assert cross(5) == [ [0,0,1,0,0],
                     [0,0,1,0,0],
                     [1,1,1,1,1],
                     [0,0,1,0,0],
                     [0,0,1,0,0] ]
```

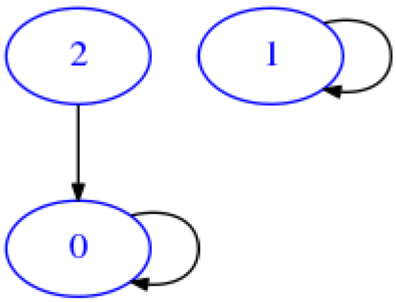
union

When we talk about the *union* of two graphs, we intend the graph having union of verteces of both graphs and having as edges the union of edges of both graphs. In this exercise, we have two graphs as list of lists with boolean edges. To simplify we suppose they have the same vertices but possibly different edges, and we want to calculate the union as a new graph.

For example, if we have a graph `ma` like this:

[14]:
```

ma =   [
            [True, False, False],
            [False, True, False],
            [True, False, False]
       ]

```

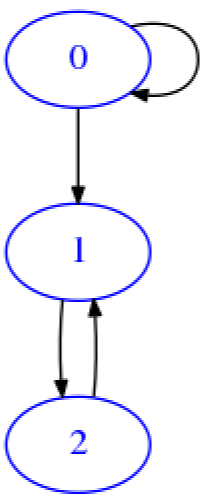
[15]:
```
draw_mat(ma)
```



And another `mb` like this:

```
[16]: mb =  [
              [True, True, False],
              [False, False, True],
              [False, True, False]

      ]
```

```
[17]: draw_mat(mb)
```

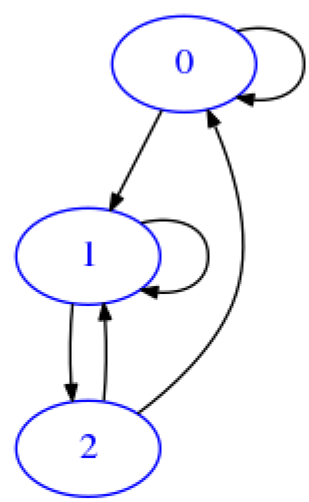


The result of calling `union(ma, mb)` will be the following:

```
[18]:
res = [[True, True, False], [False, True, True], [True, True, False]]
```

which will be displayed as

```
[19]: draw_mat(res)
```



So we get same verteces and edges from both `ma` and `mb`

Exercise - union

⊕⊕ Takes two graphs represented as nxn matrices of lists of lists with boolean edges, and RETURN a NEW matrix which is the union of both graphs

- if `mata` row number is different from `matb`, raises `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[20]:
```python
def union(mata, matb):


    if len(mata) != len(matb):
        raise ValueError("mata and matb have different row number a:%s b:%s!" \
                          % (len(mata), len(matb)))

    n = len(mata)

    ret = []
    for i in range(n):
        row = []
        ret.append(row)
        for j in range(n):
            row.append(mata[i][j] or matb[i][j])
    return ret


try:
    union([[False],[False]], [[False]])
    raise Exception("Shouldn't arrive here !")
except ValueError:
    "test passed"

try:
    union([[False]], [[False],[False]])
```

(continues on next page)

(continued from previous page)

```
    raise Exception("Shouldn't arrive here !")
except ValueError:
    "test passed"

ma1 =  [ [False] ]
mb1 =  [ [False] ]

assert union(ma1, mb1) == [ [False] ]

ma2 =  [ [False] ]
mb2 =  [ [True] ]

assert union(ma2, mb2) == [ [True] ]

ma3 =  [ [True] ]
mb3 =  [ [False] ]

assert union(ma3, mb3) == [ [True] ]


ma4 =  [ [True] ]
mb4 =  [ [True] ]

assert union(ma4, mb4) == [ [True] ]

ma5 =  [ [False, False, False],
         [False, False, False],
         [False, False, False] ]
mb5 =  [ [True, False, True],
         [False, True, True],
         [False, False, False] ]

assert union(ma5, mb5) == [ [True, False, True],
                            [False, True, True],
                            [False, False, False] ]

ma6 =  [ [True, False, True],
         [False, True, True],
         [False, False, False] ]
mb6 =  [ [False, False, False],
         [False, False, False],
         [False, False, False] ]

assert union(ma6, mb6) == [ [True, False, True],
                            [False, True, True],
                            [False, False, False] ]

ma7 =  [ [True, False, False],
         [False, True, False],
         [True, False, False] ]
mb7 =  [ [True, True, False],
         [False, False, True],
         [False, True, False] ]

assert union(ma7, mb7) == [ [True, True, False],
                            [False, True, True],
                            [True, True, False] ]
```

</div>

[20]:
```
def union(mata, matb):
    raise Exception('TODO IMPLEMENT ME !')

try:
    union([[False],[False]], [[False]])
    raise Exception("Shouldn't arrive here !")
except ValueError:
    "test passed"

try:
    union([[False]], [[False],[False]])
    raise Exception("Shouldn't arrive here !")
except ValueError:
    "test passed"

ma1 =  [ [False] ]
mb1 =  [ [False] ]

assert union(ma1, mb1) == [ [False] ]

ma2 =  [ [False] ]
mb2 =  [ [True] ]

assert union(ma2, mb2) == [ [True] ]

ma3 =  [ [True] ]
mb3 =  [ [False] ]

assert union(ma3, mb3) == [ [True] ]


ma4 =  [ [True] ]
mb4 =  [ [True] ]

assert union(ma4, mb4) == [ [True] ]

ma5 =  [ [False, False, False],
         [False, False, False],
         [False, False, False] ]
mb5 =  [ [True, False, True],
         [False, True, True],
         [False, False, False] ]

assert union(ma5, mb5) == [ [True, False, True],
                            [False, True, True],
                            [False, False, False] ]

ma6 =  [ [True, False, True],
         [False, True, True],
         [False, False, False] ]
mb6 =  [ [False, False, False],
         [False, False, False],
         [False, False, False] ]

assert union(ma6, mb6) == [ [True, False, True],
                            [False, True, True],
```
(continues on next page)

(continued from previous page)

```
                                [False, False, False] ]

ma7 =  [ [True, False, False],
         [False, True, False],
         [True, False, False] ]
mb7 =  [ [True, True, False],
         [False, False, True],
         [False, True, False] ]

assert union(ma7, mb7) == [ [True, True, False],
                            [False, True, True],
                            [True, True, False] ]
```

Subgraph

If we interpret a matrix as graph, we may wonder when a graph A is a subgraph of another graph B, that is, when A nodes are a subset of B nodes and when A edges are a subset of B edges. For convenience, here we only consider graphs having the same `n` nodes both in A and B. Edges may instead vary. Graphs are represented as boolean matrices.

Exercise - is_subgraph

⊛⊛ RETURN True is A is a subgraph of B, that is, some or all of its edges also belong to B. A and B are boolean matrices of size nxn.

- If sizes don't match, raises `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[21]:
```
def is_subgraph(mata, matb):

    n = len(mata)
    m = len(matb)
    if n != m:
        raise ValueError("A size %s and B size %s  should match !" % (n,m))
    for i in range(n):
        for j in range(n):
            if mata[i][j] and not matb[i][j]:
                return False
    return True


# the set of edges is empty
ma = [ [False] ]
# the set of edges is empty
mb = [ [False] ]
# an empty set is always a subset of an empty set
assert is_subgraph(ma, mb) == True

# the set of edges is empty
ma = [ [False] ]
# the set of edges contains one element
mb = [ [True] ]
# an empty set is always a subset of any set, so function gives True
```

(continues on next page)

(continued from previous page)

```
assert is_subgraph(ma, mb) == True

ma = [ [True] ]
mb = [ [True] ]
assert is_subgraph(ma, mb) == True

ma = [ [True] ]
mb = [ [False] ]
assert is_subgraph(ma, mb) == False

ma = [ [True, False],
       [True, False] ]
mb = [ [True, False],
       [True, True] ]
assert is_subgraph(ma, mb) == True

ma = [ [False, False, True],
       [True, True,True],
       [True, False,True] ]
mb = [ [True, False, True],
       [True, True,True],
       [True, True,True] ]
assert is_subgraph(ma, mb) == True
```

</div>

[21]:
```
def is_subgraph(mata, matb):
    raise Exception('TODO IMPLEMENT ME !')

# the set of edges is empty
ma = [ [False] ]
# the set of edges is empty
mb = [ [False] ]
# an empty set is always a subset of an empty set
assert is_subgraph(ma, mb) == True

# the set of edges is empty
ma = [ [False] ]
# the set of edges contains one element
mb = [ [True] ]
# an empty set is always a subset of any set, so function gives True
assert is_subgraph(ma, mb) == True

ma = [ [True] ]
mb = [ [True] ]
assert is_subgraph(ma, mb) == True

ma = [ [True] ]
mb = [ [False] ]
assert is_subgraph(ma, mb) == False

ma = [ [True, False],
       [True, False] ]
mb = [ [True, False],
       [True, True] ]
assert is_subgraph(ma, mb) == True
```

(continues on next page)

(continued from previous page)

```
ma = [ [False, False, True],
       [True, True,True],
       [True, False,True] ]
mb = [ [True, False, True],
       [True, True,True],
       [True, True,True] ]
assert is_subgraph(ma, mb) == True
```

Exercise - remove_node

⊛⊛ Here the function text is not so precise, as it is talking about nodes but you have to operate on a matrix. Can you guess exactly what you have to do ? In your experiments, try to draw the matrix before and after executing `remove_node`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[22]:
```
def remove_node(mat, i):
    """ MODIFIES mat by removing node i.
    """

    del mat[i]
    for row in mat:
        del row[i]


m  = [ [3,5,2,5],
       [6,2,3,7],
       [4,2,1,2],
       [7,2,2,6] ]

remove_node(m,2)

assert len(m) == 3
for i in range(3):
    assert len(m[i]) == 3
```

</div>

[22]:
```
def remove_node(mat, i):
    """ MODIFIES mat by removing node i.
    """
    raise Exception('TODO IMPLEMENT ME !')

m  = [ [3,5,2,5],
       [6,2,3,7],
       [4,2,1,2],
       [7,2,2,6] ]

remove_node(m,2)

assert len(m) == 3
for i in range(3):
    assert len(m[i]) == 3
```

Exercise - utriang

⊛⊛⊛ You will try to create an upper triangular matrix of side `n`. What could possibly be the graph interpretation of such a matrix? Since `draw_mat` is a generic drawing function doesn't provide the best possible representation, try to draw on paper a more intuitive one.

RETURN a matrix of size `nxn` which is upper triangular, that is, has all nodes below the diagonal `0`, while all the other nodes are set to `1`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[23]:
```python
def utriang(n):

    ret = []
    for i in range(n):
        row = []
        for j in range(n):
            if j < i:
                row.append(0)
            else:
                row.append(1)
        ret.append(row)
    return ret


assert utriang(1) == [ [1] ]
assert utriang(2) == [ [1,1],
                       [0,1] ]
assert utriang(3) == [ [1,1,1],
                       [0,1,1],
                       [0,0,1] ]
assert utriang(4) == [ [1,1,1,1],
                       [0,1,1,1],
                       [0,0,1,1],
                       [0,0,0,1] ]
```

</div>

[23]:
```python
def utriang(n):
    raise Exception('TODO IMPLEMENT ME !')

assert utriang(1) == [ [1] ]
assert utriang(2) == [ [1,1],
                       [0,1] ]
assert utriang(3) == [ [1,1,1],
                       [0,1,1],
                       [0,0,1] ]
assert utriang(4) == [ [1,1,1,1],
                       [0,1,1,1],
                       [0,0,1,1],
                       [0,0,0,1] ]
```

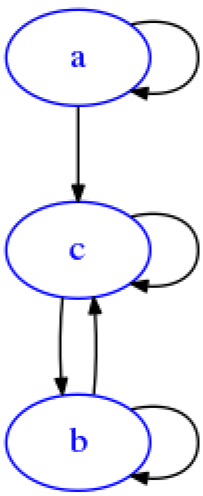
Edge difference

The *edge difference* of two graphs `ediff(da,db)` is a graph with the edges of the first except the edges of the second. For simplicity, here we consider only graphs having the same verteces but possibly different edges. This time we will try operate on graphs represented as dictionaries of adjacency lists.

For example, if we have

[24]:
```
da =  {
          'a':['a','c'],
          'b':['b', 'c'],
          'c':['b','c']
        }
```

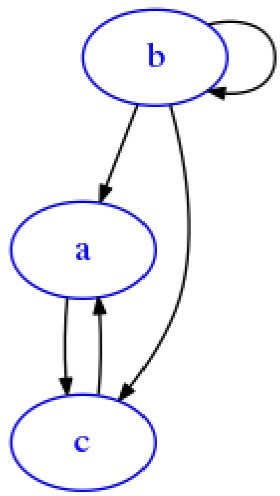
[25]:
```
draw_adj(da)
```



and

[26]:
```
db =  {
          'a':['c'],
          'b':['a','b', 'c'],
          'c':['a']
        }
```

[27]:
```
draw_adj(db)
```

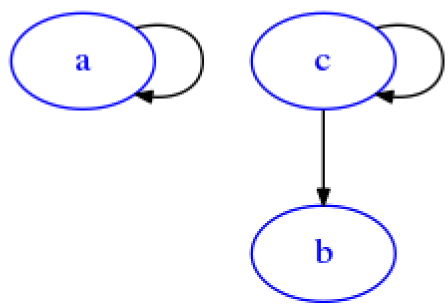


The result of calling `ediff(da,db)` will be:

[28]:
```
res = {
          'a':['a'],
          'b':[],
          'c':['b','c']
      }
```

Which can be shown as

[29]:
```
draw_adj(res)
```



Exercise - ediff

⊛⊛⊛ Takes two graphs as dictionaries of adjacency lists da and db, and RETURN a NEW graph as dictionary of adjacency lists, containing the same vertices of da, and the edges of da except the edges of db.

- As order of elements within the adjacency lists, use the same order as found in da.
- We assume all verteces in da and db are represented in the keys (even if they have no outgoing edge), and that da and db have the same keys

EXAMPLE:

```
da =  {  'a':['a','c'],
         'b':['b', 'c'],
         'c':['b','c']
```

(continues on next page)

(continued from previous page)

```
      }

db =  { 'a':['c'],
        'b':['a','b', 'c'],
        'c':['a']
      }

assert ediff(da, db) == { 'a':['a'],
                          'b':[],
                          'c':['b','c']
                        }
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[30]:
```
def ediff(da,db):


    ret = {}
    for key in da:
        ret[key] = []
        for target in da[key]:
            # not efficient but works for us
            # using sets would be better, see https://stackoverflow.com/a/6486483
            if target not in db[key]:
                ret[key].append(target)
    return ret






da1 =  { 'a': [] }
db1 =  { 'a': [] }

assert ediff(da1, db1) ==   { 'a': [] }

da2 =  { 'a': [] }

db2 =  { 'a': ['a'] }

assert ediff(da2, db2) == { 'a': [] }

da3 =  { 'a': ['a'] }
db3 =  { 'a': [] }

assert ediff(da3, db3) ==   { 'a': ['a'] }


da4 =  { 'a': ['a'] }
db4 =  { 'a': ['a'] }

assert ediff(da4, db4) == { 'a': [] }
da5 =  { 'a':['b'],
         'b':[]
       }
```

(continues on next page)

(continued from previous page)

```
db5 =  { 'a':['b'],
         'b':[] }

assert ediff(da5, db5) == { 'a':[],
                             'b':[]
                           }

da6 =  { 'a':['b'],
         'b':[]
       }
db6 =  { 'a':[],
         'b':[]
       }

assert ediff(da6, db6) == {
                             'a':['b'],
                             'b':[]
                           }

da7 =  { 'a':['a','b'],
         'b':[]
       }
db7 =  { 'a':['a'],
         'b':[]
       }

assert ediff(da7, db7) == { 'a':['b'],
                             'b':[]
                           }


da8 =  { 'a':['a','b'],
         'b':['a']
        }
db8 =  { 'a':['a'],
         'b':['b']
       }

assert ediff(da8, db8) == { 'a':['b'],
                             'b':['a']
                           }

da9 =  { 'a':['a','c'],
         'b':['b', 'c'],
         'c':['b','c']
       }

db9 =  { 'a':['c'],
         'b':['a','b', 'c'],
         'c':['a']
       }

assert ediff(da9, db9) == { 'a':['a'],
                             'b':[],
                             'c':['b','c']
                           }
```

</div>

[30]:
```python
def ediff(da,db):
    raise Exception('TODO IMPLEMENT ME !')




da1 =  { 'a': [] }
db1 =  { 'a': [] }

assert ediff(da1, db1) ==   { 'a': [] }

da2 =  { 'a': [] }

db2 =  { 'a': ['a'] }

assert ediff(da2, db2) == { 'a': [] }

da3 =  { 'a': ['a'] }
db3 =  { 'a': [] }

assert ediff(da3, db3) ==   { 'a': ['a'] }


da4 =  { 'a': ['a'] }
db4 =  { 'a': ['a'] }

assert ediff(da4, db4) == { 'a': [] }
da5 =  { 'a':['b'],
         'b':[]
       }
db5 =  { 'a':['b'],
         'b':[] }

assert ediff(da5, db5) == { 'a':[],
                            'b':[]
                          }

da6 =  { 'a':['b'],
         'b':[]
       }
db6 =  { 'a':[],
         'b':[]
       }

assert ediff(da6, db6) == {
                            'a':['b'],
                            'b':[]
                          }

da7 =  { 'a':['a','b'],
         'b':[]
       }
db7 =  { 'a':['a'],
         'b':[]
       }
```

(continues on next page)

(continued from previous page)

```
assert ediff(da7, db7) == { 'a':['b'],
                            'b':[]
                          }


da8 =  { 'a':['a','b'],
         'b':['a']
        }
db8 =  { 'a':['a'],
         'b':['b']
       }

assert ediff(da8, db8) == { 'a':['b'],
                            'b':['a']
                          }

da9 =  { 'a':['a','c'],
         'b':['b', 'c'],
         'c':['b','c']
       }

db9 =  { 'a':['c'],
         'b':['a','b', 'c'],
         'c':['a']
       }

assert ediff(da9, db9) == { 'a':['a'],
                            'b':[],
                            'c':['b','c']
                          }
```

Exercise - pyramid

⊛⊛⊛ The following function requires to create a matrix filled with non-zero numbers. Even if don't know exactly the network meaning, with this fact we can conclude that all nodes are linked to all others. A graph where this happens is called a *clique* (the Italian name is *cricca)*

Takes an odd number `n >= 1` and RETURN a matrix as list of lists containing numbers displaced like this example for a pyramid of square `7`:

```
1111111
1222221
1233321
1234321
1233321
1222221
1111111
```

- if `n` is even, raises `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[31]:
```
def pyramid(n):
```

(continues on next page)

(continued from previous page)

```
    if n % 2 == 0:
        raise ValueError("n should be odd, found instead %s" % n)
    ret = [[0]*n for i in range(n)]
    for i in range(n//2 + 1):
        for j in range(n//2 +1):
            ret[i][j] = min(i, j) + 1
            ret[i][n-j-1] = min(i, j) + 1
            ret[n-i-1][j] = min(i, j) + 1
            ret[n-i-1][n-j-1] = min(i, j) + 1

    ret[n//2][n//2] = n // 2 + 1
    return ret


try:
    pyramid(4)
    raise Exception("SHOULD HAVE FAILED!")
except ValueError:
    "passed test"

assert pyramid(1) == [
                        [1]
                    ]

assert pyramid(3) == [
                       [1,1,1],
                       [1,2,1],
                       [1,1,1]
                    ]

assert pyramid(5) == [
                        [1, 1, 1, 1, 1],
                        [1, 2, 2, 2, 1],
                        [1, 2, 3, 2, 1],
                        [1, 2, 2, 2, 1],
                        [1, 1, 1, 1, 1]
                    ]
```

</div>

[31]:
```
def pyramid(n):
    raise Exception('TODO IMPLEMENT ME !')

try:
    pyramid(4)
    raise Exception("SHOULD HAVE FAILED!")
except ValueError:
    "passed test"

assert pyramid(1) == [
                        [1]
                    ]

assert pyramid(3) == [
                       [1,1,1],
                       [1,2,1],
```

(continues on next page)

(continued from previous page)

```
                    [1,1,1]
                ]

assert pyramid(5) == [
                        [1, 1, 1, 1, 1],
                        [1, 2, 2, 2, 1],
                        [1, 2, 3, 2, 1],
                        [1, 2, 2, 2, 1],
                        [1, 1, 1, 1, 1]
                ]
```

6.2.4 Adjacency lists

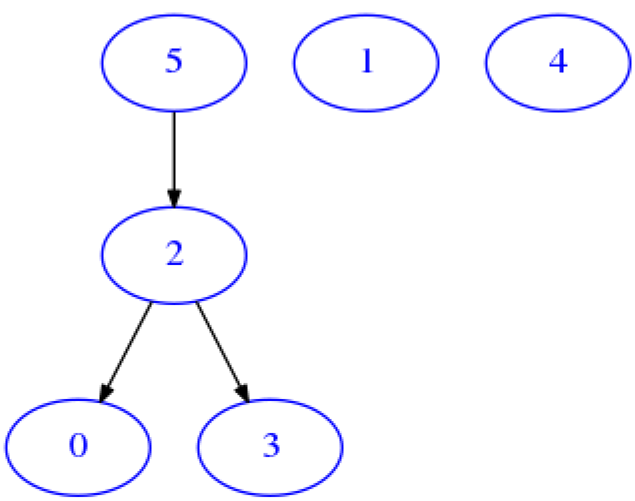
So far, we represented graphs as matrices, saying they are good when the graph is dense, that is any given node is likely to be connected to almost all other nodes - or equivalently, many cell entries in the matrix are different from zero. But if this is not the case, other representations might be needed. For example, we can represent a graph as a *adjacency lists*.

Let's look at this 6x6 boolean matrix:

[32]:
```
m = [
        [False, False, False, False, False, False],
        [False, False, False, False, False, False],
        [True,  False, False, True,  False, False],
        [False, False, False, False, False, False],
        [False, False, False, False, False, False],
        [False, False, True,  False, False, False]
]
```

We see just a few `True`, so by drawing it we don't expect to see many edges:

[33]:
```
draw_mat(m)
```



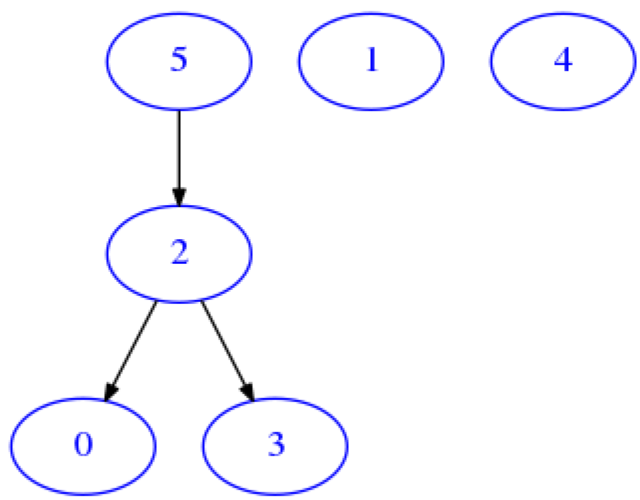
As a more compact representation, we might represent the data as a dictionary of *adjacency lists* where the keys are the node indexes and the to each node we associate a list with the target nodes it points to.

To reproduce the example above, we can write like this:

[34]:
```
d = {
        0: [],      # node 0 links to nothing
        1: [],      # node 1 links to nothing
        2: [0,3],   # node 2 links to node 0 and 3
        3: [],      # node 3 links to nothing
        4: [],      # node 4 links to nothing
        5: [2]      # node 5 links to node 2
    }
```

In `soft.py`, we provide also a function `soft.draw_adj` to quickly inspect such data structure:

[35]:
```
from soft import draw_adj

draw_adj(d)
```



As expected, the resulting graph is the same as for the equivalent matrix representation.

Exercise - mat_to_adj

✪✪ Implement a function that takes a boolean nxn matrix and RETURN the equivalent representation as dictionary of adjacency lists. Remember that to create an empty dict you have to write `dict()`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[36]:
```
def mat_to_adj(bool_mat):

    ret = dict()
    n = len(bool_mat)
    for i in range(n):
        ret[i] = []
        for j in range(n):
            if bool_mat[i][j]:
                ret[i].append(j)
    return ret
```

(continues on next page)

(continued from previous page)

```
m1 = [ [False] ]

d1 =  { 0:[] }

assert mat_to_adj(m1) == d1

m2 = [ [True] ]

d2 = { 0:[0] }

assert mat_to_adj(m2) == d2


m3 = [ [False,False],
       [False,False] ]

d3 =  { 0:[],
        1:[]
      }


assert mat_to_adj(m3) == d3


m4 = [ [True,True],
       [True,True] ]

d4 =  { 0:[0,1],
        1:[0,1]
      }


assert mat_to_adj(m4) == d4

m5 = [ [False,False],
       [False,True] ]

d5 =  { 0:[],
        1:[1]
      }


assert mat_to_adj(m5) == d5


m6 = [ [True,False,False],
       [True, True,False],
       [False,True,False] ]

d6 =  { 0:[0],
        1:[0,1],
        2:[1]
      }

assert mat_to_adj(m6) == d6
```

</div>

[36]:
```python
def mat_to_adj(bool_mat):
    raise Exception('TODO IMPLEMENT ME !')

m1 = [ [False] ]

d1 =  { 0:[] }

assert mat_to_adj(m1) == d1

m2 = [ [True] ]

d2 = { 0:[0] }

assert mat_to_adj(m2) == d2


m3 = [ [False,False],
       [False,False] ]

d3 =  { 0:[],
        1:[]
      }


assert mat_to_adj(m3) == d3


m4 = [ [True,True],
       [True,True] ]

d4 =  { 0:[0,1],
        1:[0,1]
      }


assert mat_to_adj(m4) == d4

m5 = [ [False,False],
       [False,True] ]

d5 =  { 0:[],
        1:[1]
      }


assert mat_to_adj(m5) == d5


m6 = [ [True,False,False],
       [True, True,False],
       [False,True,False] ]

d6 =  { 0:[0],
        1:[0,1],
        2:[1]
      }
```

(continues on next page)

(continued from previous page)

```
assert mat_to_adj(m6) == d6
```

Exercise - mat_ids_to_adj

⊛⊛ Implement a function that takes a boolean nxn matrix and a list of immutable identifiers for the nodes, and RETURN the equivalent representation as dictionary of adjacency lists.

- If matrix is not `nxn` or `ids` length does not match `n`, raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[37]:
```
def mat_ids_to_adj(bool_mat, ids):


    ret = dict()
    n = len(bool_mat)
    m = len(bool_mat[0])
    if n != m:
        raise ValueError('matrix is not nxn !')
    if n != len(ids):
        raise ValueError("Identifiers quantity is different from matrix size!" )
    for i in range(n):
        ret[ids[i]] = []
        for j in range(n):
            if bool_mat[i][j]:
                ret[ids[i]].append(ids[j])
    return ret



try:
    mat_ids_to_adj([[False, True]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    mat_ids_to_adj([[False]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

m1 = [ [False] ]

d1 =  { 'a':[] }
assert mat_ids_to_adj(m1, ['a']) == d1

m2 = [ [True] ]

d2 =  { 'a':['a'] }
assert mat_ids_to_adj(m2, ['a']) == d2


m3 = [ [False,False],
```

(continues on next page)

(continued from previous page)

```
       [False,False] ]

d3 =  { 'a':[],
        'b':[]
      }
assert mat_ids_to_adj(m3,['a','b']) == d3


m4 = [ [True,True],
       [True,True] ]

d4 =  { 'a':['a','b'],
        'b':['a','b']
      }

assert mat_ids_to_adj(m4, ['a','b']) == d4

m5 = [ [False,False],
       [False,True] ]

d5 =  { 'a':[],
        'b':['b']
      }


assert mat_ids_to_adj(m5,['a','b']) == d5


m6 = [ [True,False,False],
       [True, True,False],
       [False,True,False] ]

d6 =  {  'a':['a'],
         'b':['a','b'],
         'c':['b']
      }


assert mat_ids_to_adj(m6,['a','b','c']) == d6
```

</div>

[37]:
```
def mat_ids_to_adj(bool_mat, ids):
    raise Exception('TODO IMPLEMENT ME !')


try:
    mat_ids_to_adj([[False, True]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    mat_ids_to_adj([[False]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"
```

(continues on next page)

(continued from previous page)

```
m1 = [ [False] ]

d1 =  { 'a':[] }
assert mat_ids_to_adj(m1, ['a']) == d1

m2 = [ [True] ]

d2 =  { 'a':['a'] }
assert mat_ids_to_adj(m2, ['a']) == d2


m3 = [ [False,False],
       [False,False] ]

d3 =  { 'a':[],
        'b':[]
      }
assert mat_ids_to_adj(m3,['a','b']) == d3


m4 = [ [True,True],
       [True,True] ]

d4 =  { 'a':['a','b'],
        'b':['a','b']
      }

assert mat_ids_to_adj(m4, ['a','b']) == d4

m5 = [ [False,False],
       [False,True] ]

d5 =  { 'a':[],
        'b':['b']
      }


assert mat_ids_to_adj(m5,['a','b']) == d5


m6 = [ [True,False,False],
       [True, True,False],
       [False,True,False] ]

d6 =  {  'a':['a'],
         'b':['a','b'],
         'c':['b']
      }


assert mat_ids_to_adj(m6,['a','b','c']) == d6
```

Exercise - adj_to_mat

Try now conversion from dictionary of adjacency list to matrix (this is a bit hard).

To solve this, the general idea is that you have to fill an nxn matrix to return. During the filling of a cell at row `i` and column `j`, you have to decide whether to put a `True` or a `False`. You should put `True` if in the `d` list value corresponding to the `i-th` key, there is contained a number equal to `j`. Otherwise, you should put `False`.

If you look at the tests, as inputs we are passing `OrderedDict`. The reason is that when we check the output matrix of your function, we want to be sure the matrix rows are ordered in a certain way.

But you have to assume d can contain arbitrary ids with no precise ordering, so:

1. first you should scan the dictionary and lists to save the mapping between indexes to ids in a separate list

NOTE: `d.keys()` is not exactly a list (does not allow access by index), so you must convert to list with this: `list(d.keys())`

2. then you should build the matrix to return, using the previously built list when needed.

⊛⊛⊛ Now implement a function that takes a dictionary of adjacency lists with arbitrary ids and RETURN its representation as an nxn boolean matrix

- assume all nodes are present as keys
- assume `d` is a simple dictionary (not necessarily an `OrderedDict`)

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[38]:
```python
def adj_to_mat(d):

    ret = []
    n = len(d)
    ids_to_row_indexes = dict()
    # first maps row indexes to keys
    row_indexes_to_ids = list(d.keys()) # because d.keys() is *not* indexable !
    i = 0
    for key in d:
        row = []
        ret.append(row)
        for j in range(n):
            if  row_indexes_to_ids[j] in d[key]:
                row.append(True)
            else:
                row.append(False)
        i += 1
    return ret


from collections import OrderedDict
od1 = OrderedDict([ ('a',[]) ])
m1 = [ [False] ]
assert adj_to_mat(od1) == m1

od2 = OrderedDict([ ('a',['a']) ])
m2 = [ [True] ]

assert adj_to_mat(od2) == m2

od3 = OrderedDict([ ('a',['a','b']),
```

(continues on next page)

(continued from previous page)

```
                    ('b',['a','b']) ])
m3 = [  [True, True],
        [True, True] ]
assert adj_to_mat(od3) == m3

od4 = OrderedDict([ ('a',[]),
                    ('b',[]) ])

m4 = [ [False, False],
       [False, False] ]
assert adj_to_mat(od4) == m4

od5 = OrderedDict([ ('a',['a']),
                    ('b',['a','b']) ])

m5 = [ [True, False],
       [True, True] ]
assert adj_to_mat(od5) == m5


od6 = OrderedDict([ ('a',['a','c']),
                    ('b',['c']),
                    ('c',['a','b']) ])
m6 = [ [True, False, True],
       [False, False, True],
       [True, True, False] ]
assert adj_to_mat(od6) == m6
```

</div>

[38]:
```
def adj_to_mat(d):
    raise Exception('TODO IMPLEMENT ME !')

from collections import OrderedDict
od1 = OrderedDict([ ('a',[]) ])
m1 = [ [False] ]
assert adj_to_mat(od1) == m1

od2 = OrderedDict([ ('a',['a']) ])
m2 = [ [True] ]

assert adj_to_mat(od2) == m2

od3 = OrderedDict([ ('a',['a','b']),
                    ('b',['a','b']) ])
m3 = [  [True, True],
        [True, True] ]
assert adj_to_mat(od3) == m3

od4 = OrderedDict([ ('a',[]),
                    ('b',[]) ])

m4 = [ [False, False],
       [False, False] ]
assert adj_to_mat(od4) == m4

od5 = OrderedDict([ ('a',['a']),
```

(continues on next page)

(continued from previous page)

```
                    ('b',['a','b']) ])

m5 = [ [True, False],
       [True, True] ]
assert adj_to_mat(od5) == m5


od6 = OrderedDict([ ('a',['a','c']),
                    ('b',['c']),
                    ('c',['a','b']) ])
m6 = [ [True, False, True],
       [False, False, True],
       [True, True, False] ]
assert adj_to_mat(od6) == m6
```

Exercise - table_to_adj

Suppose you have a table expressed as a list of lists with headers like this:

```
[39]: m0 =    [
          ['Identifier','Price','Quantity'],
          ['a',1,1],
          ['b',5,8],
          ['c',2,6],
          ['d',8,5],
          ['e',7,3]
        ]
```

where a, b, c etc are the row identifiers (imagine they represent items in a store), Price and Quantity are properties they might have. **NOTE**: here we put two properties, but they might have n properties !

We want to transform such table into a graph-like format as a dictionary of lists, which relates store items as keys to the properties they might have. To include in the list both the property identifier and its value, we will use tuples. So you need to write a function that transforms the above input into this:

```
[40]: res0 =  {
            'a':[('Price',1),('Quantity',1)],
            'b':[('Price',5),('Quantity',8)],
            'c':[('Price',2),('Quantity',6)],
            'd':[('Price',8),('Quantity',5)],
            'e':[('Price',7),('Quantity',3)]
        }
```

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[41]: def table_to_adj(table):

    ret = {}
    headers = table[0]

    for row in table[1:]:
        lst = []
        for j in range(1, len(row)):
```

(continues on next page)

(continued from previous page)

```
            lst.append((headers[j], row[j]))
        ret[row[0]] = lst
    return ret


m0 = [ ['I','P','Q'] ]
res0 = {}

assert res0 == table_to_adj(m0)

m1 =     [
            ['Identifier','Price','Quantity'],
            ['a',1,1],
            ['b',5,8],
            ['c',2,6],
            ['d',8,5],
            ['e',7,3]
        ]
res1 = {
            'a':[('Price',1),('Quantity',1)],
            'b':[('Price',5),('Quantity',8)],
            'c':[('Price',2),('Quantity',6)],
            'd':[('Price',8),('Quantity',5)],
            'e':[('Price',7),('Quantity',3)]
        }

assert res1 == table_to_adj(m1)

m2 =     [
            ['I','P','Q'],
            ['a','x','y'],
            ['b','w','z'],
            ['c','z','x'],
            ['d','w','w'],
            ['e','y','x']
        ]
res2 =  {
            'a':[('P','x'),('Q','y')],
            'b':[('P','w'),('Q','z')],
            'c':[('P','z'),('Q','x')],
            'd':[('P','w'),('Q','w')],
            'e':[('P','y'),('Q','x')]
        }

assert res2 == table_to_adj(m2)

m3 = [
        ['I','P','Q', 'R'],
        ['a','x','y', 'x'],
        ['b','z','x', 'y'],
]

res3 = {
            'a':[('P','x'),('Q','y'), ('R','x')],
            'b':[('P','z'),('Q','x'), ('R','y')],

}
```

(continues on next page)

(continued from previous page)

```
assert res3 == table_to_adj(m3)
```

</div>

[41]:
```
def table_to_adj(table):
    raise Exception('TODO IMPLEMENT ME !')

m0 = [ ['I','P','Q'] ]
res0 = {}

assert res0 == table_to_adj(m0)

m1 =    [
            ['Identifier','Price','Quantity'],
            ['a',1,1],
            ['b',5,8],
            ['c',2,6],
            ['d',8,5],
            ['e',7,3]
        ]
res1 = {
            'a':[('Price',1),('Quantity',1)],
            'b':[('Price',5),('Quantity',8)],
            'c':[('Price',2),('Quantity',6)],
            'd':[('Price',8),('Quantity',5)],
            'e':[('Price',7),('Quantity',3)]
        }

assert res1 == table_to_adj(m1)

m2 =    [
            ['I','P','Q'],
            ['a','x','y'],
            ['b','w','z'],
            ['c','z','x'],
            ['d','w','w'],
            ['e','y','x']
        ]
res2 =  {
            'a':[('P','x'),('Q','y')],
            'b':[('P','w'),('Q','z')],
            'c':[('P','z'),('Q','x')],
            'd':[('P','w'),('Q','w')],
            'e':[('P','y'),('Q','x')]
        }

assert res2 == table_to_adj(m2)

m3 = [
        ['I','P','Q', 'R'],
        ['a','x','y', 'x'],
        ['b','z','x', 'y'],
]
```

(continues on next page)

(continued from previous page)

```
res3 = {
            'a':[('P','x'),('Q','y'), ('R','x')],
            'b':[('P','z'),('Q','x'), ('R','y')],

}


assert res3 == table_to_adj(m3)
```

6.2.5 Networkx

Before continuing, make sure to have installed the *required libraries*

Networkx is a library to perform statistics on networks. For now, it will offer us a richer data structure where we can store the properties we want in nodes and also edges.

You can initialize networkx objects with the dictionary of adjacency lists we've alredy seen:

[42]:
```
import networkx as nx

# notice with networkx if nodes are already referenced to in an adjacency list
# you do not need to put them as keys:

G=nx.DiGraph({
    'a':['b','c'],        # node a links to b and c
    'b':['b','c', 'd']    # node b links to b itself, c and d
})
```

The resulting object is not a simple dict, but something more complex:

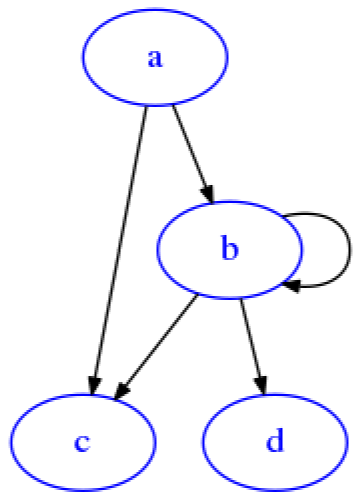
[43]:
```
G
```

[43]:
```
<networkx.classes.digraph.DiGraph at 0x7ff40e249cf8>
```

To display it in a way uniform with the rest of the course, we developed a function called `soft.draw_nx` :

[44]:
```
from soft import draw_nx
```

[45]:
```
draw_nx(G)
```



From the picture above, we notice there are no weights displayed, because in networkx they are just considered optional attributes of edges.

To see all the attributes of an edge, you can write like this:

```
[46]: G['a']['b']
```

```
[46]: {}
```

This graph has no attributes for the node, so we get back an empty dict. If we wanted to add a weight of `123` to that particular `a b` edge, you could write like this:

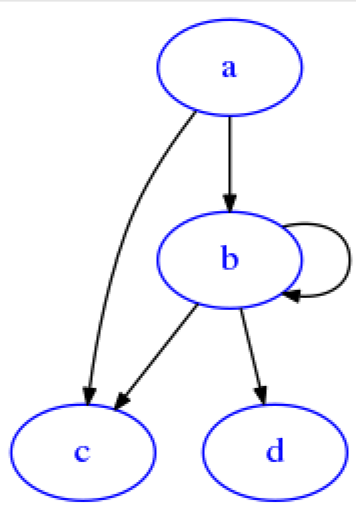
```
[47]: G['a']['b']['weight'] = 123
```

```
[48]: G['a']['b']
```

```
[48]: {'weight': 123}
```

Let's try to display it:

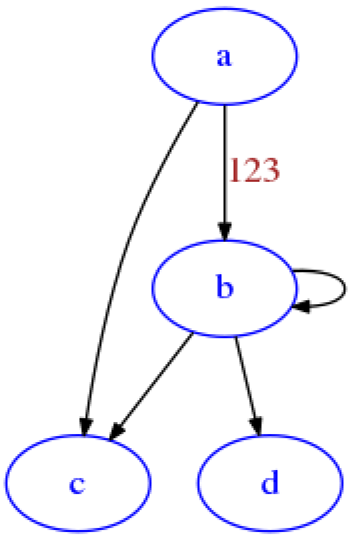
```
[49]: draw_nx(G)
```



We still don't see the weight as weight can be one of many properties: the only thing that gets displayed is the propery `label`. So let's set label equal to the weight:

[50]:
```
G['a']['b']['label'] = 123
```

[51]:
```
draw_nx(G)
```



Converting networkx graphs

If you try to just output the string representation of the graph, networkx will give the empty string:

[52]:
```
print(G)
```

[53]:
```
str(G)
```

[53]:
```
''
```

[54]:
```
repr(G)
```

[54]:
```
'<networkx.classes.digraph.DiGraph object at 0x7ff40e249cf8>'
```

To convert to the dict of adjacency lists we know, you can use this method:

[55]:
```
nx.to_dict_of_lists(G)
```

[55]:
```
{'a': ['b', 'c'], 'b': ['d', 'b', 'c'], 'c': [], 'd': []}
```

The above works, but it doesn't convert additional edge info. For a complete conversion, use `nx.to_dict_of_dicts`

[56]:
```
nx.to_dict_of_dicts(G)
```

[56]:
```
{'a': {'b': {'label': 123, 'weight': 123}, 'c': {}},
 'b': {'b': {}, 'c': {}, 'd': {}},
 'c': {},
 'd': {}}
```

Exercise - mat_to_nx

⊛⊛ Now try by yourself to convert a matrix as list of lists along with node ids (like *you did before*) into a networkx object.

This time, don't create a dictionary to pass it to `nx.DiGraph` constructor: instead, use networkx methods like `.add_edge` and `add_node`. For usage example, check the networkx tutorial[322]. Do you need to explicitly call `add_node` before referring to some node with `add_edge` ?

Implement a function that given a real-valued nxn matrix as list of lists and a list of immutable identifiers for the nodes, RETURN the corresponding graph in networkx format (as nx.DiGraph).

If matrix is not `nxn` or `ids` length does not match `n`, raise `ValueError`

- DON'T transform into a dict, use add_ methods from networkx object!
- WARNING: Remember to set the labels to the weights AS STRINGS!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[57]:
```python
def mat_to_nx(mat, ids):


    G = nx.DiGraph()
    n = len(mat)
    m = len(mat[0])
    if n != m:
        raise ValueError('matrix is not nxn !')
    if n != len(ids):
        raise ValueError("Identifiers quantity is different from matrix size!" )
    for i in range(n):
        G.add_node(ids[i])
        for j in range(n):
            if mat[i][j] != 0:
                G.add_edge(ids[i], ids[j])
                G[ids[i]][ids[j]]['weight'] = mat[i][j]
                G[ids[i]][ids[j]]['label'] = str(mat[i][j])
    return G


try:
    mat_ids_to_adj([[0, 3]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    mat_ids_to_adj([[0]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

m1 = [ [0] ]

d1 =  {'a': {}}

assert nx.to_dict_of_dicts(mat_to_nx(m1, ['a'])) == d1
```

(continues on next page)

[322] https://networkx.github.io/documentation/stable/tutorial.html

(continued from previous page)

```
m2 = [ [7] ]

d2 =  {'a': {'a': {'weight': 7, 'label': '7'}}}
assert nx.to_dict_of_dicts(mat_to_nx(m2, ['a'])) == d2


m3 = [ [0,0],
       [0,0] ]

d3 =  { 'a':{},
        'b':{}
      }
assert nx.to_dict_of_dicts(mat_to_nx(m3,['a','b'])) == d3


m4 = [ [7,9],
       [8,6] ]

d4 =  { 'a':{'a': {'weight':7,'label':'7'},
             'b' : {'weight':9,'label':'9'},
            },
        'b':{'a': {'weight':8,'label':'8'},
             'b' : {'weight':6,'label':'6'},
            }

     }

assert nx.to_dict_of_dicts(mat_to_nx(m4, ['a','b'])) == d4

m5 = [ [0,0],
       [0,7] ]

d5 =  { 'a':{},
        'b':{
                'b' : {'weight':7,'label':'7'},
            }

      }

assert nx.to_dict_of_dicts(mat_to_nx(m5,['a','b'])) == d5


m6 = [ [7,0,0],
       [7,9,0],
       [0,7,0]  ]

d6 =  { 'a':{
                'a' : {'weight':7,'label':'7'},
            },
        'b': {
                'a':  {'weight':7,'label':'7'},
                'b' : {'weight':9,'label':'9'}
             },

         'c':{
                'b' : {'weight':7,'label':'7'}
             }
```

(continues on next page)

(continued from previous page)

```
     }


assert nx.to_dict_of_dicts(mat_to_nx(m6,['a','b','c'])) == d6
```

</div>

[57]:
```
def mat_to_nx(mat, ids):
    raise Exception('TODO IMPLEMENT ME !')

try:
    mat_ids_to_adj([[0, 3]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    mat_ids_to_adj([[0]], ['a','b'])
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

m1 = [ [0] ]

d1 =  {'a': {}}

assert nx.to_dict_of_dicts(mat_to_nx(m1, ['a'])) == d1

m2 = [ [7] ]

d2 =  {'a': {'a': {'weight': 7, 'label': '7'}}}
assert nx.to_dict_of_dicts(mat_to_nx(m2, ['a'])) == d2


m3 = [ [0,0],
       [0,0] ]

d3 =  { 'a':{},
        'b':{}
      }
assert nx.to_dict_of_dicts(mat_to_nx(m3,['a','b'])) == d3


m4 = [ [7,9],
       [8,6] ]

d4 =  { 'a':{'a': {'weight':7,'label':'7'},
             'b' : {'weight':9,'label':'9'},
            },
        'b':{'a': {'weight':8,'label':'8'},
             'b' : {'weight':6,'label':'6'},
            }

     }

assert nx.to_dict_of_dicts(mat_to_nx(m4, ['a','b'])) == d4
```

(continues on next page)

(continued from previous page)

```
m5 = [ [0,0],
       [0,7] ]

d5 =  { 'a':{},
        'b':{
                'b' : {'weight':7,'label':'7'},
            }

      }

assert nx.to_dict_of_dicts(mat_to_nx(m5,['a','b'])) == d5


m6 = [ [7,0,0],
       [7,9,0],
       [0,7,0]   ]

d6 =  { 'a':{
                'a' : {'weight':7,'label':'7'},
            },
        'b': {
                'a':  {'weight':7,'label':'7'},
                'b' : {'weight':9,'label':'9'}
             },

         'c':{
                'b' : {'weight':7,'label':'7'}
             }
     }


assert nx.to_dict_of_dicts(mat_to_nx(m6,['a','b','c'])) == d6
```

6.2.6 Simple statistics

We will now compute simple statistics abour graphs. More advanced stuff will be done in Part B notebook about graph algorithms[323].

Outdegrees and indegrees

The *out-degree* $\deg^{+}(v)$ of a node v is the number of edges going out from it, while the *in-degree* $\deg^{-}(v)$ is the number of edges going into it.

NOTE: the out-degree and in-degree are *not* the sum of weights ! They just count presence or absence of edges.

For example, consider this graph:

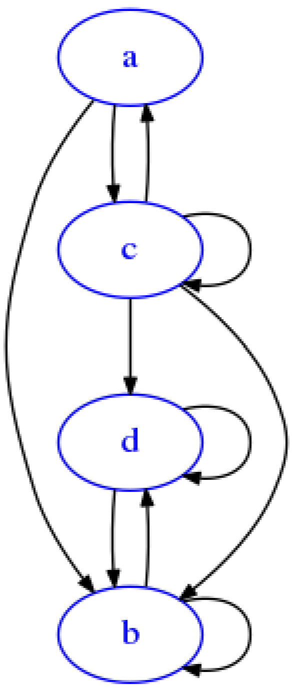
[58]:
```
from soft import draw_adj

d = {
    'a' : ['b','c'],
    'b' : ['b','d'],
```

(continues on next page)

[323] https://sciprog.davidleoni.it/graph-algos/graph-algos.html

(continued from previous page)

```
    'c' : ['a','b','c','d'],
    'd' : ['b','d']
}


draw_adj(d)
```



The out-degree of `d` is 2, because it has one outgoing edge to `b` but also an outgoing edge to itself. The indegree of `d` is 3, because it has an edge coming from `b`, one from `c` and one self-loop from `d` itself.

Exercise - outdegree_adj

⊕ RETURN the outdegree of a node from graph `d` represented as a dictionary of adjacency lists

- If `v` is not a vertex of `d`, raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[59]:
```
def outdegree_adj(d, v):

    if v not in d:
        raise ValueError("Vertex %s is not in %s" % (v, d))

    return len(d[v])


try:
    outdegree_adj({'a':[]},'b')
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"
```

(continues on next page)

(continued from previous page)

```
assert outdegree_adj({
        'a':[]
},'a') == 0

assert outdegree_adj({
        'a':['a']
},'a') == 1

assert outdegree_adj({
        'a':['a','b'],
        'b':[]
},'a') == 2

assert outdegree_adj({
        'a':['a','b'],
        'b':['a','b','c'],
        'c':[]
},'b') == 3
```

</div>

[59]:
```
def outdegree_adj(d, v):
    raise Exception('TODO IMPLEMENT ME !')

try:
    outdegree_adj({'a':[]},'b')
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

assert outdegree_adj({
        'a':[]
},'a') == 0

assert outdegree_adj({
        'a':['a']
},'a') == 1

assert outdegree_adj({
        'a':['a','b'],
        'b':[]
},'a') == 2

assert outdegree_adj({
        'a':['a','b'],
        'b':['a','b','c'],
        'c':[]
},'b') == 3
```

Exercise - outdegree_mat

⊛⊛ RETURN the outdegree of a node `i` from a graph boolean matrix nxn represented as a list of lists

- If `i` is not a node of the graph, raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[60]:
```python
def outdegree_mat(mat, i):

    n = len(mat)
    if i < 0 or i > n:
        raise ValueError("i %s is not a row of matrix %s" % (i, mat))
    ret = 0
    for j in range(n):
        if mat[i][j]:
            ret += 1
    return ret


try:
    outdegree_mat([[False]],7)
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    outdegree_mat([[False]],-1)
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"


assert outdegree_mat(
        [
            [False]
        ]
,0) == 0

assert outdegree_mat( [ [True] ],
                      0) == 1

assert outdegree_mat( [ [True, True],
                        [False, False] ],
                      0) == 2

assert outdegree_mat([ [True, True, False],
                       [True, True, True],
                       [False, False, False] ],1) == 3
```

</div>

[60]:
```python
def outdegree_mat(mat, i):
    raise Exception('TODO IMPLEMENT ME !')

try:
    outdegree_mat([[False]],7)
```

(continues on next page)

(continued from previous page)

```
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

try:
    outdegree_mat([[False]],-1)
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"


assert outdegree_mat(
        [
            [False]
        ]
,0) == 0

assert outdegree_mat( [ [True] ],
                      0) == 1

assert outdegree_mat( [ [True, True],
                        [False, False] ],
                      0) == 2

assert outdegree_mat([ [True, True, False],
                       [True, True, True],
                       [False, False, False] ],1) == 3
```

Exercise - outdegree_avg

⊛⊛ RETURN the average outdegree of nodes in graph `d`, represented as dictionary of adjacency lists.

- Assume all nodes are in the keys.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[61]:
```
def outdegree_avg(d):

    s = 0
    for k in d:
        s += len(d[k])
    return s / len(d)


assert outdegree_avg({
        'a':[]
}) == 0

assert round( outdegree_avg({
                    'a':['a']
                })
            ,2) == 1.00 / 1.00

assert round( outdegree_avg({
                    'a':['a','b'],
```

(continues on next page)

(continued from previous page)

```
                    'b':[]
                })
            ,2) == (2 + 0) / 2

assert round( outdegree_avg({
                    'a':['a','b'],
                    'b':['a','b','c'],
                    'c':[]
                })
        ,2) == round( (2 + 3) / 3 , 2)
```

</div>

```
[61]: def outdegree_avg(d):
    raise Exception('TODO IMPLEMENT ME !')

assert outdegree_avg({
        'a':[]
}) == 0

assert round( outdegree_avg({
                    'a':['a']
                })
            ,2) == 1.00 / 1.00

assert round( outdegree_avg({
                    'a':['a','b'],
                    'b':[]
                })
            ,2) == (2 + 0) / 2

assert round( outdegree_avg({
                    'a':['a','b'],
                    'b':['a','b','c'],
                    'c':[]
                })
        ,2) == round( (2 + 3) / 3 , 2)
```

Exercise - indegree_adj

The indegree of a node `v` is the number of edges going into it.

⊛⊛ RETURN the indegree of node `v` in graph `d`, represented as a dictionary of adjacency lists

- If `v` is not a node of the graph, raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

```
[62]: def indegree_adj(d, v):

    if v not in d:
        raise ValueError("Vertex %s is not in %s" % (v, d))
    ret = 0
    for k in d:
        if v in d[k]:
```

(continues on next page)

(continued from previous page)

```
            ret += 1
    return ret


try:
    indegree_adj({'a':[]},'b')
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"


assert indegree_adj({
        'a':[]
},'a') == 0

assert indegree_adj({'a':['a']},'a') == 1

assert indegree_adj({ 'a':['a','b'],
                      'b':[]},
                    'a') == 1

assert indegree_adj({ 'a':['a','b'],
                      'b':['a','b','c'],
                      'c':[]},
                    'b') == 2
```

</div>

[62]:

```
def indegree_adj(d, v):
    raise Exception('TODO IMPLEMENT ME !')

try:
    indegree_adj({'a':[]},'b')
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"


assert indegree_adj({
        'a':[]
},'a') == 0

assert indegree_adj({'a':['a']},'a') == 1

assert indegree_adj({ 'a':['a','b'],
                      'b':[]},
                    'a') == 1

assert indegree_adj({ 'a':['a','b'],
                      'b':['a','b','c'],
                      'c':[]},
                    'b') == 2
```

Exercise - indegree_mat

⊛⊛ RETURN the indegree of a node `i` from a graph boolean matrix nxn represented as a list of lists

- If `i` is not a node of the graph, raise `ValueError`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[63]:
```python
def indegree_mat(mat, i):

    n = len(mat)
    if i < 0 or i > n:
        raise ValueError("i %s is not a row of matrix %s" % (i, mat))
    ret = 0
    for k in range(n):
        if mat[k][i]:
            ret += 1
    return ret


try:
    indegree_mat([[False]],7)
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

assert indegree_mat(
        [
            [False]
        ]
,0) == 0

assert indegree_mat( [[True]],0) == 1

assert indegree_mat( [ [True, True],
                       [False, False] ],0) == 1

assert indegree_mat( [ [True, True, False],
                       [True, True, True],
                       [False, False, False] ],
                     1) == 2
```

</div>

[63]:
```python
def indegree_mat(mat, i):
    raise Exception('TODO IMPLEMENT ME !')

try:
    indegree_mat([[False]],7)
    raise Exception("SHOULD HAVE FAILED !")
except ValueError:
    "passed test"

assert indegree_mat(
        [
            [False]
        ]
```

(continues on next page)

(continued from previous page)

```
,0) == 0

assert indegree_mat( [[True]],0) == 1

assert indegree_mat( [ [True, True],
                       [False, False] ],0) == 1

assert indegree_mat( [ [True, True, False],
                       [True, True, True],
                       [False, False, False] ],
                     1) == 2
```

Exercise - indegree_avg

⊛⊛ RETURN the average indegree of nodes in graph `d`, represented as dictionary of adjacency lists.

- Assume all nodes are in the keys

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[64]:
```
def indegree_avg(d):

    s = 0
    for k in d:
        s += len(d[k])
    return s / len(d)


assert indegree_avg({
        'a':[]
}) == 0

assert round( indegree_avg({ 'a':['a'] }),
                           2) == 1.00 / 1.00

assert round( indegree_avg({ 'a':['a','b'],
                             'b':[]}),
                           2) == (1 + 1) / 2

assert round( indegree_avg({ 'a':['a','b'],
                             'b':['a','b','c'],
                             'c':[]}),
                           2) == round( (2 + 2 + 1) / 3 , 2)
```

</div>

[64]:
```
def indegree_avg(d):
    raise Exception('TODO IMPLEMENT ME !')

assert indegree_avg({
        'a':[]
}) == 0

assert round( indegree_avg({ 'a':['a'] }),
                           2) == 1.00 / 1.00
```

(continues on next page)

(continued from previous page)

```
assert round( indegree_avg({ 'a':['a','b'],
                             'b':[]}),
                           2) == (1 + 1) / 2

assert round( indegree_avg({ 'a':['a','b'],
                             'b':['a','b','c'],
                             'c':[]}),
                           2) == round( (2 + 2 + 1) / 3 , 2)
```

Was it worth it?

QUESTION: Is there any difference between the results of `indegree_avg` and `outdegree_avg` ?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: They give the same result. Think about what you did: for `outdegree_avg` you summed over all rows and then divided by `n`. For `indegree_avg` you summed over all columns, and then divided by `n`.

More formally, we have that the so-called *degree sum formula* holds (see Wikipedia[324] for more info):

$\sum_{v \in V} \deg^{-}(v) = \sum_{v \in V} \deg^{+}(v) = |A|$

</div>

Exercise - min_outdeg

Difficulty: ⊛⊛⊛

Takes a graph as matrix of list of lists and RETURN the minimum outdegree of nodes with row index between indeces start (included) and end included

- **IMPORTANT**: This function MUST be recursive, so it must call itself.
- **HINT**: REMEMBER to put `return` instructions in all `if` branches!

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[65]:

```
def helper(mat, start, end):

    n = len(mat)
    if start == end:
        return mat[start].count(True)
    else:
        half = (start + end) // 2
        min_left = helper(mat, 0, half)
        min_right = helper(mat, half+1, end)
        return min(min_left, min_right)


def min_outdeg(mat):
    """ Takes a graph as matrix of list of lists  and RETURN the minimum
```

(continues on next page)

[324] https://en.wikipedia.org/wiki/Directed_graph#Indegree_and_outdegree

(continued from previous page)

```
        outdegree of nodes by calling function helper.
        min_outdeg function is *not* recursive, only function helper is.
    """

    n = len(mat)
    return helper(mat, 0, len(mat) - 1)


assert min_outdeg( [ [False] ]) == 0
assert min_outdeg([ [True] ]) == 1

assert min_outdeg( [ [False, True],
                     [True, False] ]) == 1

assert min_outdeg( [ [True, True, False],
                     [True, True, True],
                     [False, True, True] ]) == 2

assert min_outdeg( [ [True, True, False],
                     [True, True, True],
                     [False, True, False] ]) == 1

assert min_outdeg( [ [True, True, True],
                     [True, True, True],
                     [False, True, False] ]) == 1
```

</div>

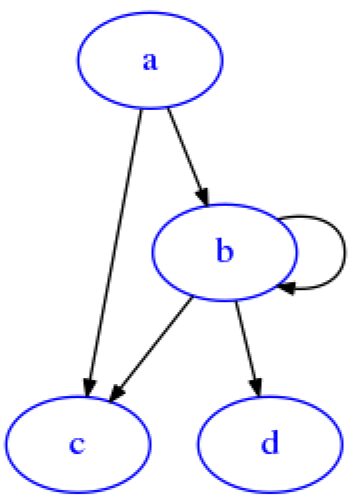
[65]:

```
def helper(mat, start, end):
    raise Exception('TODO IMPLEMENT ME !')

def min_outdeg(mat):
    """ Takes a graph as matrix of list of lists  and RETURN the minimum
        outdegree of nodes by calling function helper.
        min_outdeg function is *not* recursive, only function helper is.
    """
    raise Exception('TODO IMPLEMENT ME !')

assert min_outdeg( [ [False] ]) == 0
assert min_outdeg([ [True] ]) == 1

assert min_outdeg( [ [False, True],
                     [True, False] ]) == 1

assert min_outdeg( [ [True, True, False],
                     [True, True, True],
                     [False, True, True] ]) == 2

assert min_outdeg( [ [True, True, False],
                     [True, True, True],
                     [False, True, False] ]) == 1

assert min_outdeg( [ [True, True, True],
                     [True, True, True],
                     [False, True, False] ]) == 1
```

networkx Indegrees and outdegrees

With Networkx we can easily calculate indegrees and outdegrees of a node:

[66]:
```
import networkx as nx

# notice with networkx if nodes are already referenced to in an adjacency list
# you do not need to put them as keys:

G=nx.DiGraph({
    'a':['b','c'],         # node a links to b and c
    'b':['b','c', 'd']     # node b links to b itself, c and d
})

draw_nx(G)
```



[67]:
```
G.out_degree('a')
```

[67]: 2

QUESTION: What is the outdegree of `'b'` ? Try to think about it and then confirm your thoughts with networkx:

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[68]:
```
# write here
#print("indegree  b:  %s" % G.in_degree('b'))
#print("outdegree b:  %s" % G.out_degree('b'))
```

</div>

[68]:
```
# write here
```

QUESTION: We defined *indegree* and *outdegree*. Can you guess what the *degree* might be ? In particular, for a self pointing node like `'b'`, what could it be? Try to use `G.degree('b')` methods to validate your thoughts.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[69]:
```
# write here
#print("degree  b:  %s" % G.degree('b'))
```

</div>

[69]:
```
# write here
```

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: it is the sum of indegree and outdegree. In presence of a self-loop like for `'b'`, we count the self-loop twice, once as outgoing edge and one as incident edge

</div>

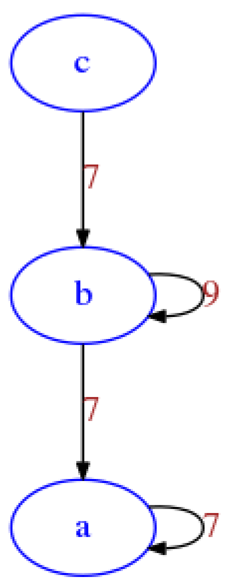
Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[70]:
```
# write here
#G.degree('b')
```

</div>

[70]:
```
# write here
```

[71]:
```
draw_nx(mat_to_nx([
        [7,0,0],
        [7,9,0],
        [0,7,0]
    ], ['a','b','c']))
```



6.3 Binary relations solutions

6.3.1 Download exercises zip

Browse files online[325]

We can use graphs to model relations of many kinds, like *isCloseTo, isFriendOf, loves,* etc. Here we review some of them and their properties.

Before going on, make sure to have read the chapter Graph formats[326]

6.3.2 What to do

- unzip exercises in a folder, you should get something like this:

```
binary-relations
    binary-relations.ipynb
    binary-relations-sol.ipynb
    jupman.py
    soft.py
```

WARNING: to correctly visualize the notebook, it MUST be in an unzipped folder !

- open Jupyter Notebook from that folder. Two things should open, first a console and then browser. The browser should show a file list: navigate the list and open the notebook `binary-relations/binary-relations.ipynb`

WARNING 2: DO NOT use the *Upload* button in Jupyter, instead navigate in Jupyter browser to the unzipped folder !

- Go on reading that notebook, and follow instuctions inside.

Shortcut keys:

- to execute Python code inside a Jupyter cell, press `Control + Enter`
- to execute Python code inside a Jupyter cell AND select next cell, press `Shift + Enter`
- to execute Python code inside a Jupyter cell AND a create a new cell aftwerwards, press `Alt + Enter`
- If the notebooks look stuck, try to select `Kernel -> Restart`

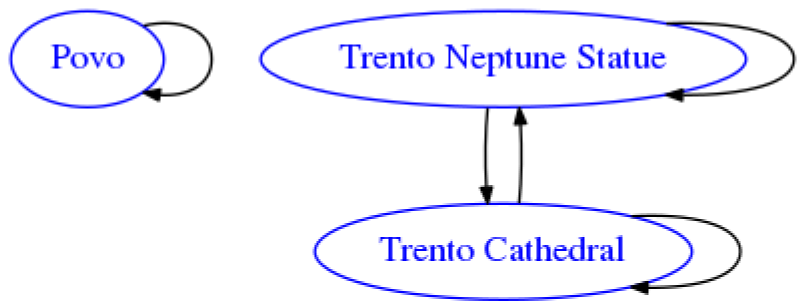
[325] https://github.com/DavidLeoni/sciprog-ds/tree/master/binary-relations
[326] https://sciprog.davidleoni.it/graph-formats/graph-formats-sol.html

6.3.3 Reflexive relations

A graph is reflexive when each node links to itself.

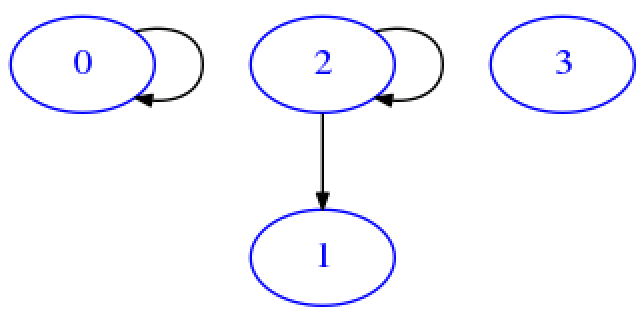
In real life, the typical reflexive relation could be "is close to" , supposing "close to" means being within a 100 meters distance. Obviously, any place is always close to itself, let's see an example (Povo is a small town around Trento):

[2]:
```python
from soft import draw_adj

draw_adj({
    'Trento Cathedral' : ['Trento Cathedral', 'Trento Neptune Statue'],
    'Trento Neptune Statue' : ['Trento Neptune Statue', 'Trento Cathedral'],
    'Povo' : ['Povo'],
})
```



Some relations might not always be necessarily reflexive, like "did homeworks for". You should always do your own homeworks, but to our dismay, university intelligence services caught some of you cheating. In the following example we expose the situation - due to privacy concerns, we identify students with numbers starting from zero included:

[3]:
```python
from soft import draw_mat

draw_mat(
    [
        [True, False, False, False],
        [False, False, False, False],
        [False, True, True, False],
        [False, False, False, False],
    ]

)
```



From the graph above, we see student 0 and student 2 both did their own homeworks. Student 3 did no homerworks at all. Alarmingly, we notice student 2 did the homeworks for student 1. Resulting conspiration shall be severely punished with a one year ban from having spritz at Emma's bar.

Exercise - is_reflexive_mat

⊛⊛ Implement a function that RETURN `True` if nxn boolean matrix mat as list of lists is reflexive, `False` otherwise.

A graph is *reflexive* when all nodes point to themselves.

- Please at least try to make the function efficient

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[4]:
```python
def is_reflexive_mat(mat):

    n = len(mat)
    for i in range(n):
        if not mat[i][i]:
            return False
    return True


assert is_reflexive_mat([ [False] ]) == False    # m1
assert is_reflexive_mat([ [True] ]) == True   # m2

assert is_reflexive_mat([ [False, False],
                          [False, False] ]) == False   # m3

assert is_reflexive_mat([ [True, True],
                          [True, True] ]) == True   # m4

assert is_reflexive_mat([ [True, True],
                          [False, True] ]) == True   # m5

assert is_reflexive_mat([ [True, False],
                          [True, True] ]) == True   # m6

assert is_reflexive_mat([ [True, True],
                          [True, False] ]) == False   # m7

assert is_reflexive_mat([ [False, True],
                          [True, True] ]) == False   # m8

assert is_reflexive_mat([ [False, True],
                          [True, False] ]) == False   # m9

assert is_reflexive_mat([ [False, False],
                          [True, False] ]) == False     # m10

assert is_reflexive_mat([ [False, True, True],
                          [True, False, False],
                          [True, True, True] ]) == False     # m11

assert is_reflexive_mat([ [True, True, True],
                          [True, True, True],
                          [True, True, True] ]) == True     # m12
```

</div>

[4]:
```python
def is_reflexive_mat(mat):
    raise Exception('TODO IMPLEMENT ME !')
```

(continues on next page)

(continued from previous page)

```
assert is_reflexive_mat([ [False] ]) == False   # m1
assert is_reflexive_mat([ [True] ]) == True  # m2

assert is_reflexive_mat([ [False, False],
                          [False, False] ]) == False  # m3

assert is_reflexive_mat([ [True, True],
                          [True, True] ]) == True  # m4

assert is_reflexive_mat([ [True, True],
                          [False, True] ]) == True  # m5

assert is_reflexive_mat([ [True, False],
                          [True, True] ]) == True  # m6

assert is_reflexive_mat([ [True, True],
                          [True, False] ]) == False  # m7

assert is_reflexive_mat([ [False, True],
                          [True, True] ]) == False  # m8

assert is_reflexive_mat([ [False, True],
                          [True, False] ]) == False  # m9

assert is_reflexive_mat([ [False, False],
                          [True, False] ]) == False    # m10

assert is_reflexive_mat([ [False, True, True],
                          [True, False, False],
                          [True, True, True] ]) == False    # m11

assert is_reflexive_mat([ [True, True, True],
                          [True, True, True],
                          [True, True, True] ]) == True    # m12
```

Exercise - is_reflexive_adj

⊛⊛ Implement now the same function for dictionaries of adjacency lists:

RETURN `True` if provided graph as dictionary of adjacency lists is reflexive, `False` otherwise.

- A graph is *reflexive* when all nodes point to themselves.
- Please at least try to make the function efficient.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[5]:
```
def is_reflexive_adj(d):


    for v in d:
        if not v in d[v]:
            return False
    return True
```

(continues on next page)

(continued from previous page)

```
assert is_reflexive_adj({ 'a':[] }) == False   # d1
assert is_reflexive_adj({ 'a':['a'] }) == True  # d2

assert is_reflexive_adj({ 'a':[],
                          'b':[]
                        }) == False  # d3

assert is_reflexive_adj({ 'a':['a'],
                          'b':['b']
                        }) == True  # d4

assert is_reflexive_adj({ 'a':['a','b'],
                          'b':['b']
                        }) == True  # d5

assert is_reflexive_adj({ 'a':['a'],
                          'b':['a','b']
                        }) == True  # d6


assert is_reflexive_adj({ 'a':['a','b'],
                          'b':['a']
                        }) == False  # d7

assert is_reflexive_adj({ 'a':['b'],
                          'b':['a','b']
                        }) == False  # d8

assert is_reflexive_adj({ 'a':['b'],
                          'b':['a']
                        }) == False  # d9

assert is_reflexive_adj({ 'a':[],
                          'b':['a']
                        }) == False    # d10

assert is_reflexive_adj({ 'a':['b','c'],
                          'b':['a'],
                          'c':['a','b','c']
                        }) == False    # d11

assert is_reflexive_adj({ 'a':['a','b','c'],
                          'b':['a','b','c'],
                          'c':['a','b','c']
                        }) == True    # d12
```

</div>

[5]:
```
def is_reflexive_adj(d):
    raise Exception('TODO IMPLEMENT ME !')

assert is_reflexive_adj({ 'a':[] }) == False   # d1
assert is_reflexive_adj({ 'a':['a'] }) == True  # d2

assert is_reflexive_adj({ 'a':[],
```

(continues on next page)

(continued from previous page)

```
                              'b':[]
                            }) == False   # d3

assert is_reflexive_adj({ 'a':['a'],
                              'b':['b']
                            }) == True   # d4

assert is_reflexive_adj({ 'a':['a','b'],
                              'b':['b']
                            }) == True   # d5

assert is_reflexive_adj({ 'a':['a'],
                              'b':['a','b']
                            }) == True   # d6


assert is_reflexive_adj({ 'a':['a','b'],
                              'b':['a']
                            }) == False   # d7

assert is_reflexive_adj({ 'a':['b'],
                              'b':['a','b']
                            }) == False   # d8

assert is_reflexive_adj({ 'a':['b'],
                              'b':['a']
                            }) == False   # d9

assert is_reflexive_adj({ 'a':[],
                              'b':['a']
                            }) == False     # d10

assert is_reflexive_adj({ 'a':['b','c'],
                              'b':['a'],
                              'c':['a','b','c']
                            }) == False     # d11

assert is_reflexive_adj({ 'a':['a','b','c'],
                              'b':['a','b','c'],
                              'c':['a','b','c']
                            }) == True     # d12
```

6.3.4 Symmetric relations

A graph is symmetric when for all nodes, if a node A links to another node B, there is a also a link from node B to A.

In real life, the typical symmetric relation is "is friend of". If you are friend to somene, that someone should be also be your friend.

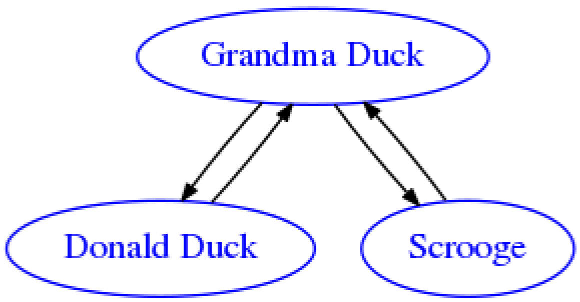
For example, since Scrooge typically is not so friendly with his lazy nephew Donald Duck, but certainly both Scrooge and Donald Duck enjoy visiting the farm of Grandma Duck, we can model their friendship relation like this:

[6]:
```
from soft import draw_adj

draw_adj({
```

(continues on next page)

(continued from previous page)

```
    'Donald Duck' : ['Grandma Duck'],
    'Scrooge' : ['Grandma Duck'],
    'Grandma Duck' : ['Scrooge', 'Donald Duck'],
})
```



Not that Scrooge is not linked to Donald Duck, but this does not mean the whole graph cannot be considered symmetric. If you pay attention to the definition above, there is *if* written at the beginning: *if* a node A links to another node B, there is a also a link from node B to A.

QUESTION: Looking purely at the above definition (so do *not* consider 'is friend of' relation), should a symmetric relation be necessarily reflexive?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: No, in a symmetric relation some nodes can be linked to themseves, while some other nodes may have no link to themselves. All we care about to check symmetry is links from a node to *other* nodes.

</div>

QUESTION: Think about the semantics of the specific "is friend of" relation: can you think of a social network where the relation is not shown as reflexive?

Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: In the particular case of "is friend to" relation is interesting, as it prompts us to think about the semantic meaning of the relation: obviously, everybody *should* be a friend of himself/herself - but if were to implement say a social network service like Facebook, it would look rather useless to show in your your friends list the information that you are a friend of yourself.

</div>

QUESTION: Always talking about the specific semantics of "is friend of" relation: can you think about some case where it should be meaningful to store information about individuals *not* being friends of themselves ?

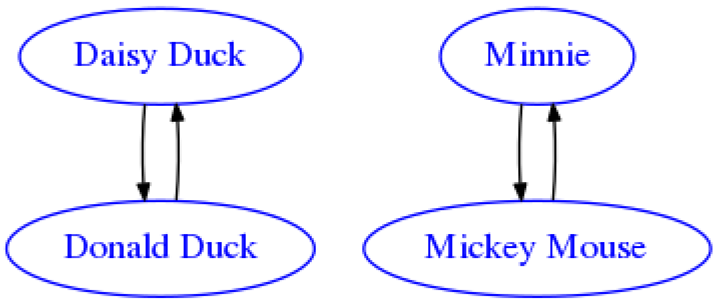
Show answer<div class="jupman-sol jupman-sol-question" style="display:none">

ANSWER: in real life it may always happen to find fringe cases - suppose you are given the task to model a network of possibly depressed people with self-harming tendencies. So always be sure your model correctly fits the problem at hand.

</div>

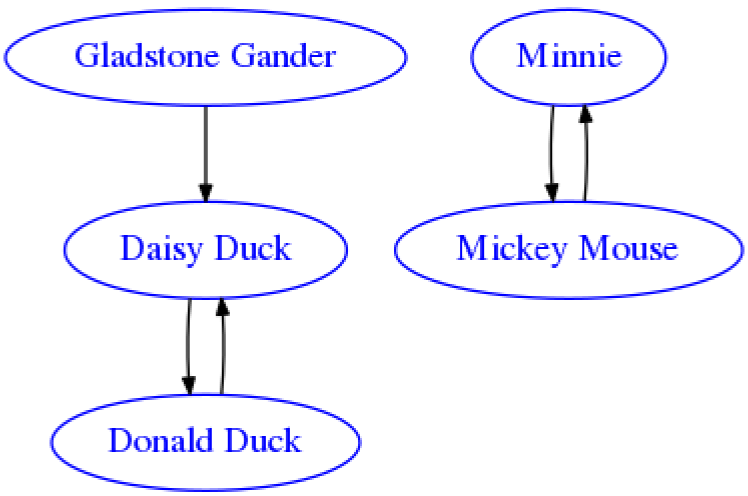
Some relations sometimes may or not be symmetric, depending on the graph at hand. Think about the relation *loves*. It is well known that Mickey Mouse lovel Minnie and the sentiment is reciprocal, and Donald Duck loves Daisy Duck and the sentiment is reciprocal. We can conclude this particular graph is symmetrical:

[7]:
```python
from soft import draw_adj

draw_adj({
    'Donald Duck' : ['Daisy Duck'],
    'Daisy Duck' : ['Donald Duck'],
    'Mickey Mouse' : ['Minnie'],
    'Minnie' : ['Mickey Mouse']

})
```



But what about this one? Donald Duck is not the only duck in town and sometimes a contender shows up: Gladstone Gander[327] (Gastone in Italian) also would like the attention of Daisy (never mind in some comics he actually gets it when Donald Duck messes up big time):

[8]:
```python
from soft import draw_adj

draw_adj({
    'Donald Duck' : ['Daisy Duck'],
    'Daisy Duck' : ['Donald Duck'],
    'Mickey Mouse' : ['Minnie'],
    'Minnie' : ['Mickey Mouse'],
    'Gladstone Gander' : ['Daisy Duck']

})
```



[327] https://en.wikipedia.org/wiki/Gladstone_Gander

Exercise - is_symmetric_mat

✪✪ Implement an automated procedure to check whether or not a graph is symmetrical, which given a matrix as a list of lists that RETURN `True` if `nxn` boolean matrix mat as list of lists is symmetric, `False` otherwise.

- A graph is symmetric when for all nodes, if a node A links to another node B, there is a also a link from node B to A.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[9]:

```
def is_symmetric_mat(mat):

    n = len(mat)
    for i in range(n):
        for j in range(n):
            if mat[i][j] and not mat[j][i]:
                return False
    return True


assert is_symmetric_mat([ [False] ]) == True    # m1
assert is_symmetric_mat([ [True] ]) == True   # m2

assert is_symmetric_mat([ [False, False],
                          [False, False] ]) == True   # m3

assert is_symmetric_mat([ [True, True],
                          [True, True] ]) == True   # m4

assert is_symmetric_mat([ [True, True],
                          [False, True] ]) == False   # m5

assert is_symmetric_mat([ [True, False],
                          [True, True] ]) == False   # m6

assert is_symmetric_mat([ [True, True],
                          [True, False] ]) == True   # m7

assert is_symmetric_mat([ [False, True],
                          [True, True] ]) == True   # m8

assert is_symmetric_mat([ [False, True],
                          [True, False] ]) == True   # m9

assert is_symmetric_mat([ [False, False],
                          [True, False] ]) == False     # m10

assert is_symmetric_mat([ [False, True, True],
                          [True, False, False],
                          [True, True, True] ]) == False     # m11

assert is_symmetric_mat([ [False, True, True],
                          [True, False, True],
                          [True, True, True] ]) == True     # m12
```

</div>

[9]:

```
def is_symmetric_mat(mat):
    raise Exception('TODO IMPLEMENT ME !')

assert is_symmetric_mat([ [False] ]) == True    # m1
assert is_symmetric_mat([ [True] ]) == True   # m2

assert is_symmetric_mat([ [False, False],
                          [False, False] ]) == True   # m3

assert is_symmetric_mat([ [True, True],
                          [True, True] ]) == True   # m4

assert is_symmetric_mat([ [True, True],
                          [False, True] ]) == False   # m5

assert is_symmetric_mat([ [True, False],
                          [True, True] ]) == False   # m6

assert is_symmetric_mat([ [True, True],
                          [True, False] ]) == True   # m7

assert is_symmetric_mat([ [False, True],
                          [True, True] ]) == True   # m8

assert is_symmetric_mat([ [False, True],
                          [True, False] ]) == True   # m9

assert is_symmetric_mat([ [False, False],
                          [True, False] ]) == False     # m10

assert is_symmetric_mat([ [False, True, True],
                          [True, False, False],
                          [True, True, True] ]) == False     # m11

assert is_symmetric_mat([ [False, True, True],
                          [True, False, True],
                          [True, True, True] ]) == True     # m12
```

Exercise - is_symmetric_adj

⊛⊛ Now implement the same as before but for a dictionary of adjacency lists:

RETURN `True` if given dictionary of adjacency lists is symmetric, `False` otherwise.

- Assume all the nodes are represented in the keys.
- A graph is symmetric when for all nodes, if a node A links to another node B, there is a also a link from node B to A.

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[10]:

```
def is_symmetric_adj(d):

    for k in d:
```

(continues on next page)

(continued from previous page)

```
        for v in d[k]:
            if not k in d[v]:
                return False
    return True


assert is_symmetric_adj({ 'a':[] }) == True    # d1
assert is_symmetric_adj({ 'a':['a'] }) == True   # d2

assert is_symmetric_adj({ 'a' : [],
                          'b' : []
                        }) == True  # d3

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['a','b']
                        }) == True  # d4

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['b']
                        }) == False  # d5

assert is_symmetric_adj({ 'a' : ['a'],
                          'b' : ['a','b']
                        }) == False  # d6

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['a']
                        }) == True  # d7

assert is_symmetric_adj({ 'a' : ['b'],
                          'b' : ['a','b']
                        }) == True  # d8

assert is_symmetric_adj({ 'a' : ['b'],
                          'b' : ['a']
                        }) == True  # d9

assert is_symmetric_adj({ 'a' : [],
                          'b' : ['a']
                        }) == False    # d10

assert is_symmetric_adj({ 'a' : ['b', 'c'],
                          'b' : ['a'],
                          'c' : ['a','b','c']
                        }) == False    # d11

assert is_symmetric_adj({ 'a' : ['b', 'c'],
                          'b' : ['a','c'],
                          'c' : ['a','b','c']
                        }) == True    # d12
```

</div>

[10]:

```
def is_symmetric_adj(d):
    raise Exception('TODO IMPLEMENT ME !')
```

(continues on next page)

(continued from previous page)

```
assert is_symmetric_adj({ 'a':[] }) == True    # d1
assert is_symmetric_adj({ 'a':['a'] }) == True  # d2

assert is_symmetric_adj({ 'a' : [],
                          'b' : []
                        }) == True  # d3

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['a','b']
                        }) == True  # d4

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['b']
                        }) == False  # d5

assert is_symmetric_adj({ 'a' : ['a'],
                          'b' : ['a','b']
                        }) == False  # d6

assert is_symmetric_adj({ 'a' : ['a','b'],
                          'b' : ['a']
                        }) == True  # d7

assert is_symmetric_adj({ 'a' : ['b'],
                          'b' : ['a','b']
                        }) == True  # d8

assert is_symmetric_adj({ 'a' : ['b'],
                          'b' : ['a']
                        }) == True  # d9

assert is_symmetric_adj({ 'a' : [],
                          'b' : ['a']
                        }) == False    # d10

assert is_symmetric_adj({ 'a' : ['b', 'c'],
                          'b' : ['a'],
                          'c' : ['a','b','c']
                        }) == False    # d11

assert is_symmetric_adj({ 'a' : ['b', 'c'],
                          'b' : ['a','c'],
                          'c' : ['a','b','c']
                        }) == True    # d12
```

6.3.5 Surjective relations

If we consider a graph as a nxn binary relation where the domain is the same as the codomain, such relation is called *surjective* if every node is reached by *at least* one edge.

For example, `G1` here is surjective, because there is at least one edge reaching into each node (self-loops as in 0 node also count as incoming edges)

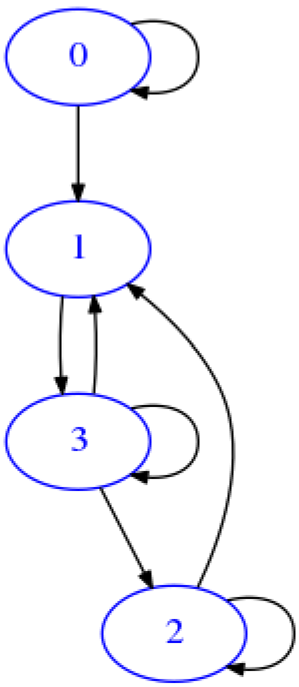
[11]:
```
G1 = [
        [True, True, False, False],
```

(continues on next page)

(continued from previous page)

```
        [False, False,  False, True],
        [False, True, True, False],
        [False, True, True, True],

     ]
```

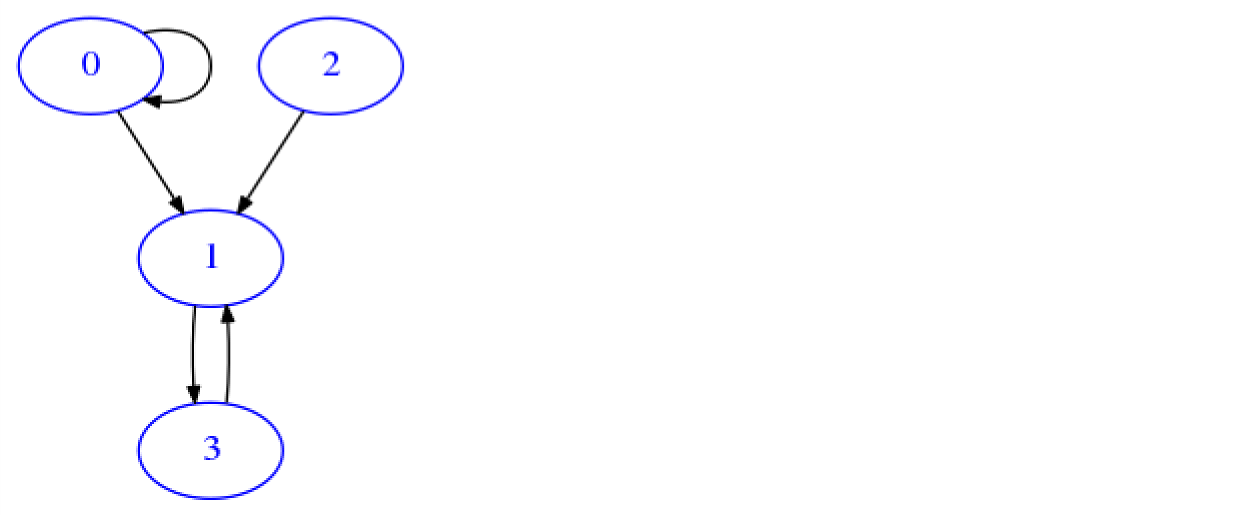
[12]:
```
draw_mat(G1)
```



G2 down here instead does not represent a surjective relation, as there is *at least* one node (2 in our case) which does not have any incoming edge:

[13]:
```
G2 = [
        [True, True, False, False],
        [False, False,  False, True],
        [False, True, False, False],
        [False, True, False, False],

     ]
```

[14]:
```
draw_mat(G2)
```



Exercise - surjective

⊛⊛ RETURN `True` if provided graph `mat` as list of boolean lists is an `nxn` surjective binary relation, otherwise return `False`

Show solution<div class="jupman-sol jupman-sol-code" style="display:none">

[15]:
```python
def surjective(mat):

    n = len(mat)
    c = 0   # number of incoming edges found
    for j in range(len(mat)):      # go column by column
        for i in range(len(mat)):  # go row by row
            if mat[i][j]:
                c += 1
                break    # as you find first incoming edge, increment c and stop␣
↪search for that column
    return c == n




m1 =  [ [False] ]

assert surjective(m1) == False


m2 =  [ [True] ]

assert surjective(m2) == True

m3 =  [ [True, False],
        [False, False] ]

assert surjective(m3) == False
```

(continues on next page)

(continued from previous page)

```
m4 =  [ [False, True],
        [False, False] ]

assert surjective(m4) == False

m5 =  [ [False, False],
        [True, False] ]

assert surjective(m5) == False

m6 =  [ [False, False],
        [False, True] ]

assert surjective(m6) == False


m7 =  [ [True, False],
        [True, False] ]

assert surjective(m7) == False

m8 =  [ [True, False],
        [False, True] ]

assert surjective(m8) == True


m9 =  [ [True, True],
        [False, True] ]

assert surjective(m9) == True


m10 = [ [True, True, False, False],
        [False, False,  False, True],
        [False, True, False, False],
        [False, True, False, False] ]
assert surjective(m10) == False

m11 = [ [True, True, False, False],
        [False, False,  False, True],
        [False, True, True, False],
        [False, True, True, True] ]
assert surjective(m11) == True
```

</div>

[15]:
```
def surjective(mat):
    raise Exception('TODO IMPLEMENT ME !')



m1 =  [ [False] ]

assert surjective(m1) == False
```

(continues on next page)

(continued from previous page)

```
m2 =  [ [True] ]

assert surjective(m2) == True

m3 =  [ [True, False],
        [False, False] ]

assert surjective(m3) == False


m4 =  [ [False, True],
        [False, False] ]

assert surjective(m4) == False

m5 =  [ [False, False],
        [True, False] ]

assert surjective(m5) == False

m6 =  [ [False, False],
        [False, True] ]

assert surjective(m6) == False


m7 =  [ [True, False],
        [True, False] ]

assert surjective(m7) == False

m8 =  [ [True, False],
        [False, True] ]

assert surjective(m8) == True


m9 =  [ [True, True],
        [False, True] ]

assert surjective(m9) == True


m10 = [ [True, True, False, False],
        [False, False,  False, True],
        [False, True, False, False],
        [False, True, False, False] ]
assert surjective(m10) == False

m11 = [ [True, True, False, False],
        [False, False,  False, True],
        [False, True, True, False],
        [False, True, True, True] ]
assert surjective(m11) == True
```

6.3.6 Further resources

- Rule based design[328] by Lex Wedemeijer, Stef Joosten, Jaap van der woude: a very readable text on how to represent information using only binary relations with boolean matrices. This a theorical book with no python exercise so it is not a mandatory read, it only gives context and practical applications for some of the material on graphs presented during the course

```
[ ]:
```

[328] https://www.researchgate.net/profile/Stef_Joosten/publication/327022933_Rule_Based_Design/links/5b7321be45851546c903234a/Rule-Based-Design.pdf

CHAPTER

SEVEN

CHAPTER

EIGHT

INDEX