FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- E' vietata la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- E' inoltre vietata la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente dall'autore o da Unica

INDUSTRIAL SOFTWARE DEVELOPMENT

Luca Didaci (didaci@unica.it)

Main topic of the second part of the course:

DESIGN PATTERNS: Standard methods for solving recurring problems.

Todays lesson:
Exercises

You can try to solve the exercises using an online IDE (http://repl.it, http://codeboard.com, ...) and send me the link.

Functions

There is a limit to the complexity that a human being can handle. ***We need to divide problems into more straightforward problems***.

The aim of these first lessons is to show that it is useful to **split a problem into small, easy sub-problems** (***divide et impera***[1]) and face with each subproblem a time.

Some guidelines can help us in splitting the original problem.

[1] (from latin: ***divide and conquer***)

EXERCISE

A *palindrome* word is a word that you can read from left to right and vice-versa, and the word doesn't change

Example: 'ABBA', 'EXE', 'ROTOR',...

In the following exercises, let us consider only UPPERCASE characters.

(Solutions: `palindrome.py`)

1. Write a function that receive in input a string and returns `True` if the string is palindrome, `False` otherwise.

2. Write a function that receives in input a string, and returns a PALINDROME string. The function must build the palindrome string **adding characters on the left** of the initial string.

   Example. '**BCBAZ**' ➡ '$ZABCB$**BCBAZ**'

3. as in 2, but the function must add the MINIMUM number of characters on the left

   Example. '**BCBAZ**' ➡ 'ZA**BCBAZ**'

HINT for point 3 (create a palindrome string adding the minimum number of character on the left): split the task in subtasks.

- check if `s` is palindrome;
- split `s` in *palindrome* and *non-palindrome* parts;

  BCBAZ ➡ {**BCB**, *AZ*}
- build the palindrome string inverting the non-palindrome part.

  {**BCB**, *AZ*} ➡ *ZA* **BCB** *AZ*

(Is it **ALWAYS** possible to split `s` in palindrome and non-palindrome parts?)

EXERCISE

Given a list of integers, find the *nearest* pair that sum to a given target. *Nearest* refers to the position in the list.

Example:

```
list `[1, 5, 3, 6, 4, 2]`; target 7
There are three pairs that meets the condition
(1 6) -> distance 3
(5 2) -> distance 4
(3 4) -> distance 2
```

so (3 4) is the nearest pair.

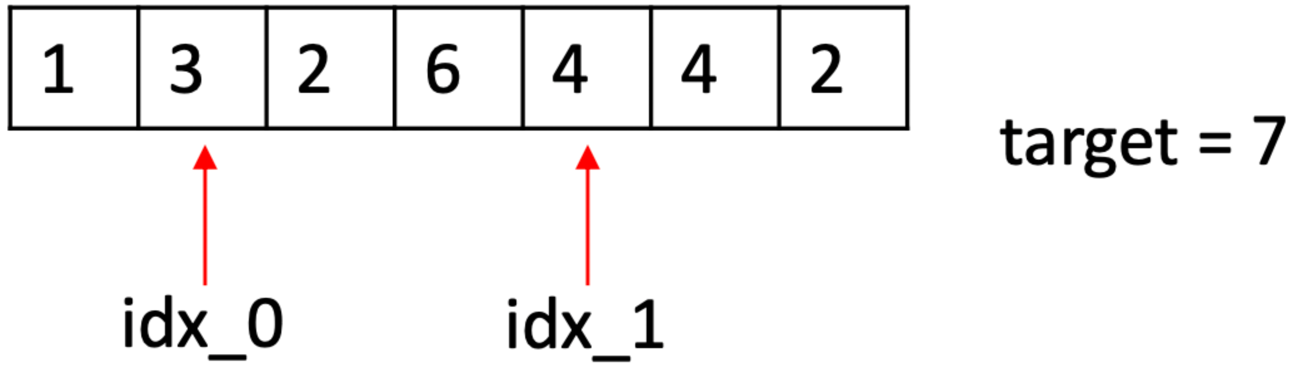
'one-step' solution: `nearest_pair_1.py`. Could be hard to understand, hard to write.

Divide and conquer approach

Or you can split the problem in two easier subproblems:

1. Given a list `v` and an index `idx_0`, find the first index `idx_1>idx_0` for which `v[idx_0]+v[idx_1]=target`

2. Use the solution (1) to solve the original problem.



1	3	2	6	4	4	2

target = 7

idx_0

idx_1

solution: `nearest_pair_2.py`

Principles/guidelines

- avoid 'magic numbers'
- **DRY**: don't repeat yourself!
- **Single-responsibility principle**
- anticipation of change
- divide and conquer

FizzBuzz Exercise

Fizz Buzz is a classic simple problem in computer science, often used as an exercise in interviews **(assigned time: 3 minutes)**.

Write a function that accepts an integer `i`, and

- if `i` is multiple of 3, print "**Fizz**" instead of the number.
- if `i` is multiple of 5, print "**Buzz**" instead of the number.
- if `i` is multiple of both 3 and 5, print "**FizzBuzz**" instead of the number.
- if no conditions are met, print the number `i`.

Test the function with all the numbers between 1 ad N.
(Naive solution: `fizzbuzz/fizzbuzz_1.py`)

Try to improve the code

- avoid 'magic numbers' - use identifiers with meaningful names
- **DRY**: don't repeat yourself! If there are duplicate parts of the code, we can put them together.

- Follow the **Single-responsibility principle**: "every function (or, in general, every module or class) should have responsibility for a single part of the problem".
  A function should have only one reason to change.

The proposed code is responsible for *two* aspects of the problem: ***compute conditions*** and ***use conditions***. -> There are *two* reasons to change the function:
- if the way conditions ***are used*** changes (i.e. print hello word instead of fizzbuzz) or
- if the ***conditions themselves*** changes (i.e. `greater than` istead of `divisible`)

If a function makes several things, it is hard to ***test*** and ***debug*** the code. If the function does not work correctly, we do not know if the error is in the part that **calculates** the conditions, or in the part that **uses** the conditions.

Use a function to compute the condition and another function to perform the action.

(Improved solution: `fizzbuzz/fizzbuzz_2.py`)

Anticipation of change

Change is unavoidable in software systems.

- user requirements may not be fully understood in the initial phase of the project
- customer needs change
- environment changes
- we must improve the software because we have to beat the competition.

We need to identify
- changes that will probably happen in the near future
- plan for change

Generalizes the problem

Initially the user asked us a software to discriminate numbers that are multiple of 3 and multiple of 5, but the user could (AND probably WILL) change his/her mind.

Example of new requirements:

Multiple of	print
3	A
5	B
7	C

Greater than	print
10	X
20	Y
30	W

Write a generalized **fizzbuzz** function `gfb()` that accepts as input
- an integer value `i`
- a `dictionary` containing a **number** (on which to test the condition) and a **value** to print.
- a function `f()` to evaluate the condition between `i` and the number.

Example:

```
dict_of_cond_1 = {3: 'A', 5: 'B', 7: 'C'}
dict_of_cond_2 = {10: 'X', 20: 'Y', 30: 'Z'}

print(gfb(21, dict_of_cond_1, is_multiple_of))   # AC
print(gfb(8, dict_of_cond_1, is_multiple_of))    # 8
print(gfb(7, dict_of_cond_2, is_greater_than))   # 7
print(gfb(20, dict_of_cond_2, is_greater_than))  # X
```

key point: python functions are first class citizen https://en.wikipedia.org/wiki/First-class_function. You can assign a function to a variable and pass a function as a input argument to another function. (Generalized solution: `fizzbuzz/fizzbuzz_3.py`)

We have considered the following **software design principles**:

- avoid 'magic numbers' - use identifiers with meaningful names.
- **DRY**: don't repeat yourself
- **Anticipation** of change
- **Single responsability** principle

In this example, `is_multiple_of()` and `is_greater_than()` are responsible for *computing* the conditions. `gfb()` is responsible for *using* the conditions.

- **Open-Closed** principle (part of *SOLID* principles): software entities should be open for extension, but closed for modification. We want to extend our software's functionality by **adding** 'parts' **without changing** the existing code.

In this example we can **add** new functions to manage new conditions **without changing** the old ones.