:::{list-table} :header-rows: 1 :align: center :widths: 90 10

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 9

  - :::{image}attachment:5a837f06-202c-4c6e-9259-dd7a6dc01c97.png :width: 200px ::: :::

RECAP:

- k-massimi in tempo $O(n * k)$ e $O(k)$ spazio (con ricerca binaria e mantenendo ordinati i k massimi)
- k-massimi in tempo $O(n * log(k))$ e $O(k)$ spazio (con SortedList)
- cenni sulla notazione Big-O

La funzione `zip` per fondere più contenitori

- prende prima i primi elementi di ciascuna lista, poi i secondi ...

Sintassi: **`zip( Contenitore1, Contenitore2, ...)`**

Il risultato è un **generatore** di tuple che contengono mano a mano gli elementi in posizione **i** dei diversi contenitori

- comodissima per scandire in parallelo più liste senza usare indici
- SI FERMA quando uno dei contenitori finisce (il più corto)
- col parametro opzionale **`strict=True`** lancia un errore se non sono della stessa lunghezza

In [1…

```
# Esempio: unisco più sequenze
```

```
L1 = [ 1, 2, 3, 4, 5, 6, 7 ]  # 7 elementi
S2 = 'abcdef'                 # 6 elementi
L3 = ['A', 'B', 'C', 'D']     # 4 elementi

list(zip(L1, S2, L3))         # estraggo le tuple dal generatore e ci riempio una lista
# NOTA: torna una sequenza della lunghezza MINIMA tra le tre
```

Out[1… [(1, 'a', 'A'), (2, 'b', 'B'), (3, 'c', 'C'), (4, 'd', 'D')]

In [2…
```
## Esempio: se invece li stampo su 3 colonne **usando indici**
# (genero un errore perchè la prima è più lunga delle altre)
for i in range(len(L1)):
    print(L1[i], S2[i], L3[i], sep='\t')
```

```
1       a       A
2       b       B
3       c       C
4       d       D
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[2], line 4
      1 ## Esempio: se invece li stampo su 3 colonne **usando indici**
      2 # (genero un errore perchè la prima è più lunga delle altre)
      3 for i in range(len(L1)):
----> 4     print(L1[i], S2[i], L3[i], sep='\t')

IndexError: list index out of range
```

In [3…
```
## Modo corretto per lavorare con gli indici
for i in range(min(len(L1),len(S2),len(L3))):   # prendiamo il minimo
    print(L1[i], S2[i], L3[i], sep='\t')
```

```
1       a       A
2       b       B
3       c       C
4       d       D
```

In [4…
```
## Esempio: li stampo su 3 colonne con zip
for a, b, c in zip(L1, S2, L3):
```

```
    print(a, b, c, sep='\t')
```

```
1       a       A
2       b       B
3       c       C
4       d       D
```

Espressioni condizionate che danno 2 risultati diversi

è una espressione che può avere due risultati diversi a seconda di una condizione.

Invece di scrivere

```
if condizione:
    A = valore_se_vero
else:
    A = valore_se_falso
```

potete scrivere

```
A = valore_se_vero if condizione else valore_se_falso
```

In C/C++ si scriverebbe

```
A = condizione ? valore_se_vero : valore_se_falso ;
```

In [9…

```
# Esempio
B = 34
# A = B>0 ? 23 : 45 ;  # assegnamento condizionato in C
A = 23 if B>0 else 45  # lo stesso assegnamento condizionato in Python

print(A)
```

```
23
```

RECAP: list comprehension
sintassi semplificata per costruire contenitori semplici

```
   <parentesi che indica il tipo di contenitore>
```

```
    <espressione che calcola ciascun elemento>
    <ciclo/i>
    <condizione/i>
    ...
  <parentesi chiusa>
```

In [6…
```
## Esempio complicato con for nidificati e condizionati
[   (x, y)                  # ciascun elemento è una coppia (x,y)
    for x in range(5)       # per ciascun x in 0,1,2,3,4
    if  x%2                 # ma solo se x è dispari
    for y in range(5*x)     # e per ogni y in 0,1,2,3,4... 5*x-1
    if  y%3                 # ma solo se y non è multiplo di 3 (resto != 0)
  ]
```

Out[6…
```
[(1, 1),
 (1, 2),
 (1, 4),
 (3, 1),
 (3, 2),
 (3, 4),
 (3, 5),
 (3, 7),
 (3, 8),
 (3, 10),
 (3, 11),
 (3, 13),
 (3, 14)]
```

In [7…
```
## Tipi di contenitori
[      x*2 for x in range(10) ]  # lista
{      x*2 for x in range(10) }  # insieme    (singolo elemento)
{   x: x*2 for x in range(10) }  # dizionario (coppia chiave : valore)
tuple( x*2 for x in range(10) )  # tupla
```

Out[7…
```
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

In [8…
```
# Possiamo usare espressioni condizionate
# ad esempio nelle list-comprehension
```

```
# per tornare valori diversi per casi diversi
[ x*2 if x%3!=0           #           calcola x*2 se x non è divisibile per 3
      else x*4            # altrimenti calcola x*4
      for x in range(10) ]
```

Out[8… [0, 2, 4, 12, 8, 10, 24, 14, 16, 36]

RECAP: sorted

sorted e **sort**, **min** e **max** ammettono un parametro **key** che accetta una funzione che fa da "criterio di ordinamento"che:

- accetta un solo valore (l'elemento della lista)
- produce un qualcosa che sarà usato da sorted per ordinare i valori (trasformazione di Schwartz)

In [9…
```
## Esempio: ordinare una lista di interi mettendo:
#  - prima i valori pari, in ordine decrescente
#  - poi i valori dispari, in ordine crescente

L = [ 1, 5, 2, 4, 7, 1, 9, 4, 8, 6, ]

# soluzione 1: separo i due gruppi, li ordino, poi li concateno
pari    = [ x for x in L if x%2==0 ]  # estraggo i pari
dispari = [ x for x in L if x%2    ]  # e i dispari
print('pari', pari, 'dispari', dispari)
pari.sort(reverse=True)               # li ordino
dispari.sort()
pari + dispari                        # li concateno
```

```
pari [2, 4, 4, 8, 6] dispari [1, 5, 7, 1, 9]
```

Out[9… [8, 6, 4, 4, 2, 1, 1, 5, 7, 9]

In [1…
```
# Soluzione 2: per ogni valore creo una coppia che contiene
# - per prima cosa l'indicazione se sono pari o dispari che separerà i due gruppi
#   ad esempio il resto della divisione per 2
#   che mette prima i pari (resto 0) e poi i dispari (resto 1)
# - per seconda il valore:
```

```
#   - così com'è per ordinarlo in senso crescente se è dispari
#   - oppure negato per ordinarlo in senso decrescente se era pari
def criterio(x):
    return x%2, (x if x%2 else -x)

print('trasf. di Schwartz', [ criterio(x) for x in L ])
print('Schwartz in ordine', sorted([ criterio(x) for x in L ]))

sorted([ criterio(x) for x in L])
sorted(L, key=criterio)
```

```
trasf. di Schwartz [(1, 1), (1, 5), (0, -2), (0, -4), (1, 7), (1, 1), (1, 9), (0, -4), (0, -8), (0, -6)]
Schwartz in ordine [(0, -8), (0, -6), (0, -4), (0, -4), (0, -2), (1, 1), (1, 1), (1, 5), (1, 7), (1, 9)]
```

Out[1…

```
[8, 6, 4, 4, 2, 1, 1, 5, 7, 9]
```

RECAP: funzioni anonime (espressioni `lambda`)

- accettano uno o più argomenti
- calcolano SOLO una espressione

Sintassi: **`lambda <argomenti>: <espressione>`**

Sono utilissime per piccole trasformazioni dei dati, per sorted ed altre operazioni "funzionali"

In [1…

```
## Esempio:
## ricerca del massimo rispetto ad un ordinamento complicato
## Usiamo il *key* di prima nella funzione *max* come se fosse una *sorted*

sorted(L, key=lambda x: (x%2, x if x%2 else -x))   # le parentesi nella lambda sono necessarie
```

Out[1…

```
[8, 6, 4, 4, 2, 1, 1, 5, 7, 9]
```

Funzioni definite internamente a funzioni

```
def funzione_esterna(argomenti):
```

```
def funzione_interna(argomenti1):
    codice della funzione_interna
    che può usare sia argomenti1 che argomenti
codice della funzione_esterna
che può usare la funzione_interna (ma non i suoi argomenti)
```

- sono disponibili SOLO all'interno della funzione "esterna" che le contiene
- possono **usare gli argomenti e le variabili** definiti nella funzione "esterna"
  - perchè l'ordine di ricerca è **LEGB**: **L**ocal **E**nclosing **G**lobal **B**uiltin

Sono comodissime quando:

- vogliamo **impedire** che siano usate altrove (**nascono** quando la funzione viene CHIAMATA e scompaiono al return)
- in altre parti del codice vogliamo usare lo stesso nome di funzione e vogliamo evitare conflitti
- vogliamo rendere più **leggibile** il codice dando un nome alla trasformazione
- la trasformazione da applicare ai dati ha bisogno di **più di una** istruzione
- dobbiamo usare **UN SOLO argomento** ma i conti da fare hanno bisogno di altri parametri aggiuntivi disponibili nella funzione esterna

In [9…
```
# Esempio: data una tabella definita come lista di dizionari
agenda = [
    {'nome': 'Paperino','cognome':'Paolino',      'telefono':'555-1313',  'indirizzo': 'via dei Peri 11
    {'nome': 'Gastone', 'cognome':'Paperone',     'telefono':'555-1717',  'indirizzo': 'via dei Baobab 4
    {'nome': 'Paperon', 'cognome':"de' Paperoni", 'telefono':'555-99999', 'indirizzo': 'colle Papero 1',
    {'nome': 'Archimede','cognome':'Pitagorico',  'telefono':'555-11235', 'indirizzo': 'colle degli Inve
    {'nome': 'Pietro',  'cognome':'Gambadilegno', 'telefono':'555-66666', 'indirizzo': 'via dei Ladri 13
    {'nome': 'Trudy',   'cognome':'Gambadilegno', 'telefono':'555-66666', 'indirizzo': 'via dei Ladri 13
    {'nome': 'Topolino','cognome':'Mouse',        'telefono':'555-12345', 'indirizzo': 'via degli Invest
    {'nome': 'Minnie',  'cognome':'Mouse',        'telefono':'555-54321', 'indirizzo': 'via di M.me Curi
    {'nome': 'Pippo',   'cognome':"de' Pippis",   'telefono':'555-33333', 'indirizzo': 'via dei Pioppi 1
    ]
```

In [9…
```
# Se voglio cercare i valore massimo rispetto ad una colonna data come argomento
def cerca_massimo(ag, colonna) :
    # dato che key DEVE essere una funzione *di un solo argomento*
    # ma per estrarre il valore mi serve *anche il nome della colonna*
```

```
    # definisco il criterio come inner function
    def estrai_valore(riga) :
        # la colonna la leggo dal namespace della funzione che contiene estrai_valore (LEGB)
        return riga[colonna]

    return max(ag, key=estrai_valore)

cerca_massimo(agenda, 'nome')
```

Out[9…
```
{'nome': 'Trudy',
 'cognome': 'Gambadilegno',
 'telefono': '555-66666',
 'indirizzo': 'via dei Ladri 13',
 'città': 'Topolinia'}
```

In [1…
```
# per una estrazione così semplice posso usare anche una lambda
# NOTA: anche la lambda ha accesso alle variabili della funzione che la contiene
def cerca_massimo(ag, colonna) :
    return max(ag, key=lambda riga: riga[colonna])

cerca_massimo(agenda, 'nome')
```

Out[1…
```
{'nome': 'Trudy',
 'cognome': 'Gambadilegno',
 'telefono': '555-66666',
 'indirizzo': 'via dei Ladri 13',
 'città': 'Topolinia'}
```

NOTA: negli esercizi ricorsivi all'esame NON usate inner functions

Potrebbe venirvi spontaneo di definire inner functions per risolvere i problemi ricorsivi

Però il mio test per riconoscere se avete usato una soluzione ricorsiva riconosce SOLO le funzioni esterne riscorsive

Ve lo ricorderò all'esame più avanti e all'esame

Tipi in Python

Cosa sono i tipi in Python: sono la definizione della rappresentazione dei dati e delle operazioni che si possono fare su di essi (cioè sono le classi degli oggetti usati)

Python NON dichiara nè controlla **nè a compile time nè a runtime** se i tipi dei dati forniti alle funzioni (e ritornati) sono "giusti"

Questo perchè i tipi sono dinamici e gli oggetti riferiti dalle variabili "sanno" quali metodi possono essere applicati, ma solo quando vengono eseguiti

Questo vi fornisce ampia libertà con tutta una serie di controlli eseguiti a run-time

Nei linguaggi compilati (C++, Java, ...) **dobbiamo indicare i tipi** e questo permette di spostare nella fase di compilazione molti controlli e rendere il codice compilato molto più veloce

Esistono tool per controllare i tipi usati nel programma, i due più comuni sono:

- **mypy** che fa una analisi statica del codice (ovvero SENZA eseguirlo)
  e deduce e controlla i tipi usati dal programma
- **typeguard** a run-time (eseguendo il codice)
  controlla che i tipi siano corretti durante l'esecuzione
- entrambi sfruttano le annotazioni di tipo che aggiungete

Aggiungere i tipi alle definizioni delle funzioni, variabili ed argomenti ci fa catturare molti più errori e fa scrivere codice più corretto

Esistono inoltre dei compilatori di Python (ad es. il progetto **Codon** oppure **Cython**) che lo rendono più efficiente e veloce (quasi come il C/C++) se i tipi vengono indicati (ma solo per un sottoinsieme del linguaggio).

Come si aggiungono i tipi al codice

```
# definizione del tipo di una variabile nell'assegnamento
variabile : tipo = valore
# oppure come commento
variabile = valore # type : tipo

# definizione del tipo degli argomenti di una funzione e del suo risultato
def funzione( argomento1 : tipo1, ...) -> return_type :
    codice della funzione
```

```
# oppure come commento
def funzione( argomento1, ...):
    # type:(tipo1, ...) -> return_type
    codice della funzione
```

Queste "annotazioni" finiscono in un attributo dell'oggetto funzione e possono essere esaminate e controllate con `mypy` oppure con `typeguard`

Come si descrivono i tipi

- potete usare direttamente i nomi delle classi corrispondenti
  - **bool, int, float, dict, set, tuple, list, ...**

```
pi : float = 3.14                    # pi    contiene un float
dati : list[int] = [ 3, 6, 12, 45 ]  # dati  contiene una lista di int
```

NOTATE: quando si tratta di tipi composti si usano le parentesi QUADRE `[]`

Per i contenitori possiamo indicare **tra parentesi quadre** `[]` il tipo degli elementi contenuti nel contenitore

esempio	descrizione
list	lista eterogenea
list[int]	lista omogenea di **int** (di lunghezza indefinita)
dict[str, int]	dizionario con chiavi tutte **str** e valori tutti **int**
set[float]	insieme omogeneo di **float**
tuple[int, float]	coppia di valori (**int**, **float**) nell'ordine
tuple[int, ...]	tupla di lunghezza variabile contenente solo **int**

Potete usare sinonimi (alias) al posto di tipi complessi

Questo aiuta a rendere più leggibili le annotazioni

```
# Esempio:
```

```
# una scheda è un dizionario di coppie  'info':'valore'
Scheda = dict[str, str]
# una agendina è una lista di schede
Agenda = list[Scheda]
```

Si tendono ad usare nomi maiuscoli per indicare i tipi

Ci sono tipi speciali nel modulo `typing`

`from typing import ...`

- **None** per indicare il valore **None**
- **Any** per indicare che va bene qualsiasi tipo
- **NoReturn** per funzioni che non tornano valori (lanciano sempre errore)
- **Union[T1, T2]** oppure **T1 | T2** per indicare che sono validi sia **T1** che **T2**

- **Optional[T]** equivale a **T | None** oppure **Union[T, None]**
- **Callable[[...], ReturnType]**
  per indicare una funzione che riceve gli argomenti **[...]** (primo argomento)
  e che torna un **ReturnType** (secondo argomento)
- **TypeVar** per definire tipi parametrici
- ...

In [1…

```
## Esempio di annotazioni di tipo
from typing import Sized
# i Sized sono tutti gli oggetti hanno il metodo __len__
# per cui se ne può calcolare la dimensione usando la funzione len()

# qui ho un criterio di ordinamento che
# - accetta cose che hanno una lunghezza (sono di tipo Sized)
# - e produce coppie (int,Sized)
def criterio(elemento : Sized) -> tuple[int,Sized]:
        return len(elemento), elemento
```

```
# i tipi sono inseriti nell'attributo __annotations__ dell'oggetto funzione creato
criterio.__annotations__
```

Out[1…

```
{'elemento': typing.Sized, 'return': tuple[int, typing.Sized]}
```

In [1…

```
# provo a eseguire un test su sorted
def test_sorted() -> list:
    L        : list[Sized] = [ '932', '1', '23', '045', 2 ] # <== ATTENZIONE intero!!!
    expected : list[Sized] = ['1', '2', '23', '045', '932']
    result = sorted(L,key=criterio)
    assert result == expected
    return result

if __name__ == '__main__':
    test_sorted()
# scopriamo l'errore SOLO A RUNTIME!!!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 10
      7     return result
      9 if __name__ == '__main__':
---> 10     test_sorted()
     11 # scopriamo l'errore SOLO A RUNTIME!!!

Cell In[16], line 5, in test_sorted()
      3 L        : list[Sized] = [ '932', '1', '23', '045', 2 ] # <== ATTENZIONE intero!!!
      4 expected : list[Sized] = ['1', '2', '23', '045', '932']
----> 5 result = sorted(L,key=criterio)
      6 assert result == expected
      7 return result

Cell In[15], line 10, in criterio(elemento)
      9 def criterio(elemento : Sized) -> tuple[int,Sized]:
---> 10         return len(elemento), elemento

TypeError: object of type 'int' has no len()
```

come verificare i tipi staticamente? (senza eseguire il codice)

`mypy --pretty sorted.py`

mypy scopre l'errore di tipo già dall'assegnamento!!!

In [9…
```
!mypy --pretty sorted.py
```

```
sorted.py:7: error: List item 4 has incompatible type "int"; expected "Sized"
[list-item]
        L        : list[Sized] = [ '932', '1', '23', '045', 2 ]
                                                            ^
Found 1 error in 1 file (checked 1 source file)
```

MA è anche utile verificare i tipi a runtime

- possono dipendere da dati esterni
- **mypy** non è detto che abbia tutte le info necessarie
  - moduli non tipati o tipati male
  - alcune deduzioni ancora non sono implementate

Run-time type-checking

Non si fa così: `python sorted.py`

(lo scopriamo all'ultimo momento, eseguendo **len**)

In [1…
```
!python sorted.py
```

```
python: can't open file '/Users/andrea/Documents/Uni/Didattica/Prog1/2025-26/Lezioni/lezione09/sorted.p
y': [Errno 2] No such file or directory
```

Run-time type-checking

NON si fa così: `pytest sorted.py`

```
E           TypeError: object of type 'int' has no len()
```

Si è scoperto solo perchè **len(int)** non funziona

In [2…
```
!pytest sorted.py
```

```
============================= test session starts ==============================
platform darwin -- Python 3.12.11, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/andrea/Documents/Uni/Didattica/Prog1/2025-26/Lezioni/lezione09
plugins: anyio-4.11.0, hypothesis-6.140.3, typeguard-4.4.4
collected 0 items

============================ no tests ran in 0.10s =============================
ERROR: file or directory not found: sorted.py
```

Molto meglio: col modulo `typeguard`

Che si installa nell'Anaconda prompt col comando

`conda install typeguard -c conda-forge`

PS: è già installato nella VM

Si usa eseguendo in Anaconda prompt il comando

`pytest sorted.py --typeguard-packages=sorted`

`E              typeguard.TypeCheckError : argument "elemento" (int) is not an instance of collections.abc.Sized`

BENE: adesso **typeguard** si è accorto che **2** non è di tipo **Sized** quando ha controllato l'assegnamento

In [9…
```
!pytest sorted.py --typeguard-packages=sorted
```

```
============================= test session starts ==============================
platform darwin -- Python 3.12.11, pytest-8.4.2, pluggy-1.6.0
rootdir: /Users/andrea/Documents/Uni/Didattica/Prog1/2025-26/Lezioni/lezione09
plugins: anyio-4.11.0, hypothesis-6.140.3, typeguard-4.4.4
collected 1 item

sorted.py F                                                              [100%]

=================================== FAILURES ===================================
_____________________________________ test_sorted _____________________________________

    def test_sorted() -> None :
        L        : list[Sized] = [ '932', '1', '23', '045', 2 ]
        expected : list[Sized] = [ '1', '23', '045', '932' ]
>       assert sorted(L, key=criterio_sort) == expected
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^

sorted.py:9:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
sorted.py:3: in criterio_sort
    def criterio_sort(elemento : Sized) -> tuple[int, Sized]:
/Users/andrea/micromamba/envs/F25/lib/python3.12/site-packages/typeguard/_functions.py:137: in check_arg
ument_types
    check_type_internal(value, annotation, memo)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

value = 2, annotation = typing.Sized
memo = <typeguard.TypeCheckMemo object at 0x1063d6c00>

    def check_type_internal(
        value: Any,
        annotation: Any,
        memo: TypeCheckMemo,
    ) -> None:
        """
        Check that the given object is compatible with the given type annotation.
```

```
This function should only be used by type checker callables. Applications should use
:func:`~.check_type` instead.

:param value: the value to check
:param annotation: the type annotation to check against
:param memo: a memo object containing configuration and information necessary for
    looking up forward references
"""

if isinstance(annotation, ForwardRef):
    try:
        annotation = evaluate_forwardref(annotation, memo)
    except NameError:
        if memo.config.forward_ref_policy is ForwardRefPolicy.ERROR:
            raise
        elif memo.config.forward_ref_policy is ForwardRefPolicy.WARN:
            warnings.warn(
                f"Cannot resolve forward reference {annotation.__forward_arg__!r}",
                TypeHintWarning,
                stacklevel=get_stacklevel(),
            )

        return

if annotation is Any or annotation is SubclassableAny or isinstance(value, Mock):
    return

# Skip type checks if value is an instance of a class that inherits from Any
if not isclass(value) and SubclassableAny in type(value).__bases__:
    return

extras: tuple[Any, ...]
origin_type = get_origin(annotation)
if origin_type is Annotated:
    annotation, *extras_ = get_args(annotation)
    extras = tuple(extras_)
    origin_type = get_origin(annotation)
```

```
        else:
            extras = ()

        if origin_type is not None:
            args = get_args(annotation)

            # Compatibility hack to distinguish between unparametrized and empty tuple
            # (tuple[()]), necessary due to https://github.com/python/cpython/issues/91137
            if origin_type in (tuple, Tuple) and annotation is not Tuple and not args:
                args = ((),)
        else:
            origin_type = annotation
            args = ()

        for lookup_func in checker_lookup_functions:
            checker = lookup_func(origin_type, args, extras)
            if checker:
                checker(value, origin_type, args, memo)
                return

        if isclass(origin_type):
            if not isinstance(value, origin_type):
>               raise TypeCheckError(f"is not an instance of {qualified_name(origin_type)}")
E               typeguard.TypeCheckError: argument "elemento" (int) is not an instance of collections.abc.Sized

/Users/andrea/micromamba/envs/F25/lib/python3.12/site-packages/typeguard/_checkers.py:965: TypeCheckError
============================ short test summary info =============================
FAILED sorted.py::test_sorted - typeguard.TypeCheckError: argument "elemento" (int) is not an instance of c...
============================== 1 failed in 0.14s ===============================
```

In [9…

```
!cat sorted2.py
!echo '========================================'
!python sorted2.py
```

```
### Oppure (sconsigliato): MODIFICANDO IL CODICE
from typeguard import typechecked
from typing import Sized

@typechecked          # decoratore che controlla i tipi in ingresso e uscita
def criterio(el : Sized) -> tuple[int,Sized]:
    return len(el), el

LL : list[Sized] = [ 'uno', 2 ]
sorted(LL, key=criterio)     # <== viene scovato qui
========================================
Traceback (most recent call last):
  File "/Users/andrea/Documents/Uni/Didattica/Prog1/2025-26/Lezioni/lezione09/sorted2.py", line 10, in <
module>
    sorted(LL, key=criterio)     # <== viene scovato qui
    ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/andrea/Documents/Uni/Didattica/Prog1/2025-26/Lezioni/lezione09/sorted2.py", line 6, in cr
iterio
    def criterio(el : Sized) -> tuple[int,Sized]:
  File "/Users/andrea/micromamba/envs/F25/lib/python3.12/site-packages/typeguard/_functions.py", line 13
7, in check_argument_types
    check_type_internal(value, annotation, memo)
  File "/Users/andrea/micromamba/envs/F25/lib/python3.12/site-packages/typeguard/_checkers.py", line 965
, in check_type_internal
    raise TypeCheckError(f"is not an instance of {qualified_name(origin_type)}")
typeguard.TypeCheckError: argument "el" (int) is not an instance of collections.abc.Sized
```

Cosa vi consiglio di fare

Definite i tipi degli argomenti e dei risultati delle vostre funzioni (e i tipi delle vostre variabili)

Chiamate, senza modificare il codice

- **`mypy --pretty programma.py`**
- **`pytest programma.py --typeguard-packages=programma`**

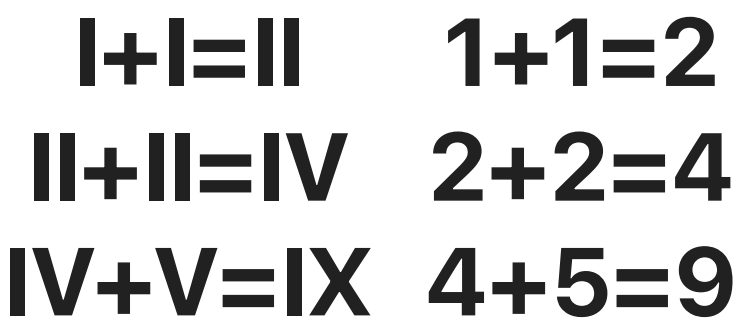
Oppure potete installare i plugin per il vostro IDE:

- per PyCharm `Mypy (Official)`
- per VSCode `Mypy Type Checker`
- per Spyder `pyls-mypy`

INTERVALLO

Un vecchio Homework (AA 22-23)

dal blog XKCD di Randall Munroe



Dato un numero romano (ad es. 'MCMLXI' = 1961) composto dei simboli

lettera	peso	lettera	peso	lettera	peso	lettera	peso	lettera
M	1000	**D**	500	**C**	100	**L**	50	**X**

| **X** | 10 | **V** | 5 | **I** | 1

Supponiamo di scriverlo concatenando direttamente i valori di ciascuna lettera

```
'MCMLXI' => 1000 100 1000 50 10 1
         => '1000100100050101'
```

Chiamiamo questo formato "formato XKCD" degli interi da 1 a 3999

Obiettivo dello HW

Bisognava implementare le 4 funzioni:

```python
def decode_XKCD_tuple( xkcd_values : tuple[str, ...], k : int) -> list[int]:
    "decodifica una tupla di interi in formato XKCD e ne estrae i K massimi"
def decode_value( xkcd : str ) -> int:
    "decodifica un valore dal formato XKCD all'intero corrispondente"
def xkcd_to_list_of_weights( xkcd : str) -> list[int]:
    "trasforma un valore in formato XKCD nella lista di pesi interi"
def list_of_weights_to_number( weigths : list[int] ) -> int:
    "data una lista di pesi ottiene il numero intero corrispondente"
```

E poi estrarre da una lista di stringhe in "formato XKCD" i **k** valori massimi.

- convertendo ciascuna stringa nella lista di pesi
- convertendo ciascuna lista di pesi nel valore intero

Perchè questo HW era così semplice?

- fornisce già l'analisi (per chi è alle prime armi)
- controlla che si usino i dati giusti a runtime con **typeguard** (e che non vengano tolti)
- controlla che non si scriva codice troppo intricato con **radon**

- per obbligare a suddividere i programmi in funzioni piccole

Gli HW seguenti NON erano specificati a questo livello di dettaglio (l'analisi del problema era a carico degli studenti)

In [2…
```
# carico un modulo che applica mypy alle celle di questo notebook
%load_ext nb_mypy
# attivo la verifica ad ogni salvataggio
%nb_mypy Off
```

```
Version 1.0.6
```

Realizziamo lo HW!

- **decode_XKCD_tuple**: per decodificare la tupla e ottenere i k massimi
  - trasformiamo ciascuna stringa nell'intero (sottoproblema)
  - e gestiamo i k massimi come si è visto (sottoproblema)

In [9…
```
def decode_XKCD_tuple( xkcd_values : tuple[str, ...], k : int) -> list[int]:
    'decodifica una tupla di interi in formato XKCD e ne estrae i K massimi'
    interi = [ decode_value(X) for X in xkcd_values ]
    return sorted(interi, reverse=True)[:k]    # semplice ma non la più veloce
# Esempio
decode_XKCD_tuple(('1101000','150','1000100100050101',), 2)
```

Out[9…
```
[1961, 989]
```

- **decode_value**: per trasformare una stringa in un intero
  - trasformiamo la stringa in una lista di pesi (sottoproblema)
  - otteniamo l'intero dalla lista di pesi (sottoproblema)

In [2…
```
def decode_value( xkcd : str ) -> int:
    "decodifica un valore dal formato XKCD all'intero corrispondente"
    pesi = xkcd_to_list_of_weights(xkcd)
    return list_of_weights_to_number(pesi)
```

- **xkcd_to_list_of_weights**: per trasformare la stringa in una lista di pesi
  - scandiamo la stringa cercando i pesi in ordine di lunghezza decrescente (da '1000' a '1')
  - oppure osserviamo che ciascun peso inizia con 5 o con 1

Esempio: 1961: '1000100100050101' -> [1000, 100, 1000, 50, 1]

In [2…
```python
def xkcd_to_list_of_weights( xkcd : str) -> list[int]:
    '''trasforma un valore in formato XKCD nella lista di pesi interi'''
    pesi : list[str] = ['1000','500','100','50','10','5','1']
    risultato : list[int] = []
    while xkcd:                              # se c'è almeno un carattere da convertire
        for peso in pesi:                    # per ciascun numero "romano" in ordine decrescente di lung
            if xkcd.startswith(peso):        # se i primi caratteri gli corrispondono
                risultato.append(int(peso))  # aggiungo il peso al risultato come intero
                xkcd = xkcd[len(peso):]      # accorcio la stringa togliendo la parte trovata
                break                        # smetto il for e continuo il while
    return risultato
# Esempio
xkcd_to_list_of_weights('1000100100050101')
```

Out[2…
```
[1000, 100, 1000, 50, 10, 1]
```

In [2…
```python
# altrimenti se notiamo che tutti i pesi iniziano per 1 oppure per 5
# inserisco spazio prima di 1 e di 5 e uso split
def xkcd_to_list_of_weights( xkcd : str) -> list[int]:
    '''trasforma un valore in formato XKCD nella lista di pesi interi'''
    spaziato  : str       = xkcd.replace('1', ' 1')     # inserisco spazio prima di 1
    spaziato  : str       = spaziato.replace('5', ' 5') # inserisco spazio prima di 5
    frammenti : list[str] = spaziato.split()            # spezzo la stringa sugli spazi
    return [ int(X) for X in frammenti ]                # e converto in interi i frammenti
# Esempio
xkcd_to_list_of_weights('1000100100050101')
```

Out[2…
```
[1000, 100, 1000, 50, 10, 1]
```

- **list_of_weights_to_number**: per trasformare la lista di pesi in un intero
  - applichiamo le regole dei numeri romani:
  - ovvero se un peso è seguito da un peso maggiore (es. 9 = IX)
    - va sottratto
    - altrimenti va sommato

Esempio: [1000, 100, 1000, 50, 1] -> (1000 - 100 + 1000 + 50 + 1) = 1961

In [2…
```python
def list_of_weights_to_number( weigths : list[int] ) -> int:
    'data una lista di pesi ottiene il numero intero corrispondente'
    somma = 0
    for i in range(len(weigths)-1): # per ciascun valore fino al *penultimo*
        if weigths[i]<weigths[i+1]: # se è *seguito* da un valore maggiore
            somma -= weigths[i]     # va sottratto
        else:                       # altrimenti
            somma += weigths[i]     # va sommato
    return somma + weigths[-1]      # aggiungo l'ultimo
# Esempio
list_of_weights_to_number([1000, 100, 1000, 50, 10, 1])
```

Out[2…
```
1961
```

In [2…
```python
# oppure sfrutto zip per scandire i pesi e i pesi sfalzati di 1 posto
def list_of_weights_to_number( weigths : list[int] ) -> int:
    'implementazione usando una list comprehension e zip'
    return sum( -X if X<Y else X              # sommo X oppure lo sottraggo se  minore del seguente
                for X,Y in zip(weigths,             # scandendo in parallelo i pesi
                               weigths[1:]+[0]) )   # e i pesi sfalzati di 1 e con uno 0 in fondo
list_of_weights_to_number([1000, 100, 1000, 50, 10, 1])
```

Out[2…
```
1961
```

PAUSA