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

- ▪ Fondamenti di programmazione

  Canale A-L - Prof. Sterbini

  AA 25-26 - Lezione 7

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

In [1…

```
from timeit import timeit
```

RECAP

- input da tastiera
- gestione errori con try/except/finally
- ordinamento e criteri di ordinamento complessi

FDPLEZ7

:::{list-table}

. . FDPLEZ7

- :::{image}attachment:image.png :width: 700px ::: :::

RECALL: Sort secondo criteri complessi

- creiamo una funzione `criterio_di_ordinamento` che:
  - riceve un SOLO elemento della lista
  - ritorna una tupla con i valori che rappresentano i criteri di ordinamento **nell'ordine in cui vanno considerati**

NOTA: la funzione di trasformazione deve ricevere **UN SOLO ARGOMENTO** (l'elemento da trasformare) e in genere produce una **semplice espressione**, e ci serve solo una volta!

(🤪) Sarebbe bello poter definire funzioni "usa e getta" per questo scopo

Funzioni anonime (espressioni *lambda*)

Notate che la funzione `criterio_di_ordinamento` è estremamente semplice ed ha un solo compito:

- riceve dei parametri
- ritorna SOLO una espressione **SENZA ESEGUIRE ALTRE ISTRUZIONI**

Quindi ha una sola istruzione ... e può essere scritta in modo compatto

Quando la funzione calcola **una sola espressione** potete definirla come funzione usa-e-getta

```
lambda argomenti: espressione

# MEGLIO, se producete una tupla
lambda argomenti: (espressione)
```

NOTA l'espressione può produrre una lista o una tupla di valori, che è proprio quello che ci serve per **sorted**

In [2…
```
# quindi la funzione *criterio_di_ordinamento*
# può essere scritta direttamente come

def criterio_di_ordinamento(elemento):
    return (len(elemento),elemento.lower(), elemento)

# OVVERO
criterio_di_ordinamento = lambda elemento: (len(elemento),elemento.lower(), elemento)

# calcoliamo la tupla per l'elemento 'PaPErino'
print(criterio_di_ordinamento('PaPErino'))
```

```
(8, 'paperino', 'PaPErino')
```

In [2…
```
# usiamo la lambda nel sorted (RICORDATEVI LE PARENTESI DELLA TUPLA!!!)
L = [ 'PaPerino', 'plUTO', ' TopoLINO', 'minniE', 'PLUTO', 'papeRINO', 'papEROne', 'gasTONE', 'MInnie']
S = sorted(
    L,                  # lista da ordinare
    key=lambda elemento: (len(elemento),elemento.lower(), elemento), # criterio
    reverse=True)       # in ordine opposto
print(S,'\n\n\n\n')
```

```
['TopoLINO', 'papEROne', 'papeRINO', 'PaPerino', 'gasTONE', 'minniE', 'MInnie', 'plUTO', 'PLUTO']
```

RIASSUNTO: Per ordinare un contenitore secondo criteri complessi

- definiamo la funzione di trasformazione (meglio) o una lambda (meno leggibile) che:
  - riceve l'elemento da trasformare (**UN SOLO ARGOMENTO**)

    - e torna una tupla contenente nell'ordine i dati che servono a rappresentare i criteri complessi
- verrà usata da **sorted** per:
    - trasformare ciascun elemento
    - ordinare la lista di tuple ottenute
    - tornare gli elementi originali nel nuovo ordine

Ordinamenti contrapposti!!!

Esempio: ordiniamo la lista di parole

- in ordine **CRESCENTE** di lunghezza
- e in caso di parità in ordine alfabetico **DECRESCENTE** (Z->A)

qui abbiamo un problema in più, **i due ordinamenti vanno in direzioni opposte!**

Uno dei due va rovesciato, perchè **i confronti tra tuple sono sempre <u>monodirezionali</u>** cioè tutti gli elementi delle tuple vengono confrontati nella stessa direzione

NON POSSO rovesciare l'ordinamento alfabetico

POSSO rovesciare l'ordinamento dei numeri **CAMBIANDOGLI SEGNO**!

In [4…
```python
# definiamo la funzione di trasformazione adatta
# NON possiamo rovesciare l'ordinamento alfabetico!
# ALLORA rovesciamo l'ordinamento delle lunghezze
def trasforma_elemento(el):
    # lunghezza negativa: valori più grandi saranno più a sinistra e vengono prima
    return -len(el), el
```

In [5…
```python
print('disordinata', L)
# vediamo cosa succede
S = sorted(L, key=trasforma_elemento)
# E' tutto a rovescio rispetto al nostro task!
print('ordinata   ', S)
```

```
disordinata ['PaPerino', 'plUTO', 'TopoLINO', 'minniE', 'PLUTO', 'papeRINO', 'papEROne', 'gasTONE', 'MInnie']
ordinata    ['PaPerino', 'TopoLINO', 'papEROne', 'papeRINO', 'gasTONE', 'MInnie', 'minniE', 'PLUTO', 'plUTO']
```

In [6…
```python
# cambiamo il parametro *reverse* per ordinare le stringhe in direzione opposta
S = sorted(L, key=trasforma_elemento, reverse=True)
# ora va bene
print(S)
```

```
['plUTO', 'PLUTO', 'minniE', 'MInnie', 'gasTONE', 'papeRINO', 'papEROne', 'TopoLINO', 'PaPerino']
```

In [3…
```python
# Possiamo anche riscrivere la trasformazione usando una funzione anonima con lambda
# (notate che sono NECESSARIE le parentesi per creare la tupla
#  ed evitare confusione con gli argomenti di sorted)
S = sorted(L, key=lambda el: (-len(el),el), reverse=True)
print(S,'\n\n\n')
```

```
['plUTO', 'PLUTO', 'minniE', 'MInnie', 'gasTONE', 'papeRINO', 'papEROne', 'TopoLINO', 'PaPerino']
```

In [5…
```python
# Supponiamo di sapere età e genere di alcuni personaggi
L = [ (27, 'M', 'Paperino'), (31, 'M', 'Topolino'),
      (26, 'F', 'Paperina'), (32, 'F', 'Minnie')]
# li voglio in ordine di:
# - lunghezza del nome crescente
# - genere crescente ('F' < 'M')
# - età crescente
def trasforma_elemento(terna):
    eta, genere, nome = terna      # spacchetto la terna
    return len(nome), genere, eta  # calcolo la tupla
sorted(L, key=trasforma_elemento)
```

Out[5…
```
 [(32, 'F', 'Minnie'),
  (26, 'F', 'Paperina'),
  (27, 'M', 'Paperino'),
  (31, 'M', 'Topolino')]
```

La trasformazione di Schwartz dei criteri di ordinamento può essere usata anche per `min` e per `max`

In [9…
```
print(min(L, key=trasforma_elemento))   # estraggo il primo
print(max(L, key=trasforma_elemento))   # e l'ultimo
```

```
(32, 'F', 'Minnie')
(31, 'M', 'Topolino')
```

List comprehension

una sintassi semplificata per costruire contenitori

Spessissimo dobbiamo raccogliere in un contenitore più elementi e scriviamo roba del tipo di

- iniziamo costruendo una lista vuota
- iteriamo su un contenitore di elementi
  - trasformiamo ciascun elemento
  - lo aggiungiamo alla lista
- torniamo la lista ottenuta

Sarebbe bello poter usare una sintassi più leggibile

(che non ci obbliga a **simulare** il codice nella testa, ma che è simile alle espressioni insiemistiche)

In [1…
```
# dati dei valori
lista_di_interi = [12, 34, 61, 2, 4, 7]
# voglio costruire la lista dei cubi
def cubi(numeri):
    lista_di_cubi = []              # inizio con una lista vuota
    for x in numeri:                # scandisco i valori
        lista_di_cubi.append(x**3)  # aggiungo in fondo il cubo del valore corrente
    return lista_di_cubi
```

```
cubi(lista_di_interi)                # ho ottenuto i cubi
```

Out[1… [1728, 39304, 226981, 8, 64, 343]

List-comprehension: sintassi

```
# se inizio con la parentesi quadra sto creando una lista
[ espressione                         # valore calcolato per ciascun elemento del risultato
      for variabile in contenitore   # per ciascun valore di contenitore di dati
]
```

In [1…
```
lista_di_interi = [12, 34, 2, 61, 2, 4, 7]

# Con la sintassi della list-comprehension
lista_di_cubi = [ X**3 for X in lista_di_interi ]

# - le parentesi indicano che tipo di contenitore stiamo costruendo (lista)
# - il *for* indica su quale sequenza di dati stiamo iterando
# - l'espressione all'inizio (X**3) indica come produciamo ogni nuovo valore
lista_di_cubi
```

Out[1… [1728, 39304, 8, 226981, 8, 64, 343]

In [1…
```
# possiamo  costruire altri tipi di contenitore
# INSIEMI, racchiusi tra parentesi graffe
{ X**3 for X in lista_di_interi }
```

Out[1… {8, 64, 343, 1728, 39304, 226981}

In [1…
```
# DIZIONARI che contengono coppie chiave:valore racchiuse tra parentesi graffe
{ X : X**3 for X in lista_di_interi }
```

Out[1… {12: 1728, 34: 39304, 2: 8, 61: 226981, 4: 64, 7: 343}

In [1…
```
# TUPLE (qui ci vuole la parola tuple, non bastano le parentesi)
tuple( X**3 for X in lista_di_interi )
```

Out[1… (1728, 39304, 8, 226981, 8, 64, 343)

e possiamo anche condizionare quali elementi trasformare e quali ignorare

```
[ espressione                       # valore calcolato
     for variabile in contenitore   # per ciascun valore del contenitore
     if condizione                  # ma solo se la condizione è vera
]
```

In [1…
```
# Voglio i cubi MA SOLO dei numeri dispari
{ x : x**3 for x in lista_di_interi if x%2!=0 }
```

Out[1… {61: 226981, 7: 343}

oppure costruire for nidificati

```
[ espressione                 # valore calcolato
     for var1 in contenitore1 # per ciascun valore di contenitore1
     for var2 in contenitore2 # per ciascun valore di contenitore2
]
```

In [7…
```
# Esempio: costruisco l'insieme dei prodotti della tabellina del 10
S = { X*Y                    # ciascun elemento è un prodotto:
        for X in range(1,11) # di  ciascuno dei valori X da 1 a 10
        for Y in range(1,11) # per ciascuno dei valori Y da 1 a 10
    }
print(S)
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 15, 16, 18, 20, 21, 24, 25, 27, 28, 30, 32, 35, 36, 40, 42, 45,
48, 49, 50, 54, 56, 60, 63, 64, 70, 72, 80, 81, 90, 100}
```

oppure possiamo costruire contenitori nidificati

```
[ [ espressione for var1 in contenitore1 ]   # ciascun elemento della lista esterna è una lista
                for var2 in contenitore2     # ne viene creata una per ciascun valore del
contenitore2
]
```

In [9…
```
# Ad esempio creiamo la tabellina del 10 come lista di liste
# (una lista per ogni riga)
```

```
T = [ [ X*Y for X in range(1,11) ]  # per ogni Y costruisco la riga con 10 moltiplicazioni
            for Y in range(1,11) ]
%pprint
T
```

```
Pretty printing has been turned ON
```

Out[9…

```
 [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
  [2, 4, 6, 8, 10, 12, 14, 16, 18, 20],
  [3, 6, 9, 12, 15, 18, 21, 24, 27, 30],
  [4, 8, 12, 16, 20, 24, 28, 32, 36, 40],
  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50],
  [6, 12, 18, 24, 30, 36, 42, 48, 54, 60],
  [7, 14, 21, 28, 35, 42, 49, 56, 63, 70],
  [8, 16, 24, 32, 40, 48, 56, 64, 72, 80],
  [9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
  [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]]
```

FDPLEZ7

:::{list-table}

. . FDPLEZ7

- :::{image}attachment:image.png :width: 700px ::: :::

In [1…

```
## Vediamo la soluzione

alfabeto = "abcdefghijk"
parola   = "bigia"

[ alfabeto[alfabeto.index(X):]   # il pezzo dalla posizione del carattere X in poi
```

```
    for X in parola]          # per ciascuna lettera X di parola
```

Out[1… `['bcdefghijk', 'ijk', 'ghijk', 'ijk', 'abcdefghijk']`

COME analizzare un problema?

Metodologia di analisi top-down

- cerchiamo di capire **come lo faremmo su carta**
  - descrivendo gli **input ed output** (cioè argomenti e return)
  - eventuali **controlli da fare sui dati**
  - possibili **errori da generare**
  - possibili **effetti collaterali**
  - possibili **scelte non indicate**

- lo dividiamo in **problemi più piccoli**
  - ripetendo il metodo di analisi
- **continuando a frammentarlo** finchè
  - la sua **descrizione è così semplice**
  - da poter essere **implementato**
- mano a mano **testiamo le sottofunzioni** realizzate
  - **semplificando enormemente il debug** che diventa più **veloce**
  - (all'esame è importantissimo saper debuggare rapidamente)
- è naturale che **gli approcci che già conosciamo** ci faranno da guida
  - cerchiamo di imparare tanti approcci

Come capire quali sono gli input, i conti da fare e gli output?

Esaminate la descrizione del problema :

- input ed output sono normalmente dei **nomi**
- calcoli da fare sono normalmente dei **verbi**
- può essere necessaria della conoscenza di dominio

Esempio: **calcolate** la **media** di un **elenco di altezze**

- **elenco di altezze**: input
- **media**: risultato
- **calcolate la media**: calcolo da svolgere, conoscendo come è definita la media di un gruppo di valori (conoscenza esterna)

Come riconoscere dove suddividere il problema in parti più semplici?

Ad esempio:

- se il problema deve compiere le stesse operazioni su un gruppo di dati
  - possiamo spostare quel calcolo in una funzione separata da chiamare in un FOR o un WHILE
- se il problema è descritto come una successione di passi
  - possiamo realizzare ciascun passo in una funzione diversa
  - e chiamarle nell'ordine
- se dobbiamo generare una struttura dati intermedia
  - in un primo passo creiamo la struttura
  - in un secondo la usiamo
- ...

Esempio di problema: trovare i k massimi di una lista L di valori numerici

Vediamo che parole sono state usate:

- **K**: input
- **lista L di valori numerici**: input
- **k massimi**: output

- **trovare i k massimi**: calcolo da eseguire

- **rappresentazione dei dati**?
  (in questo caso lo sappiamo)
  - INPUT: una lista L di numeri ed un valore intero K
  - OUTPUT: la lista dei K massimi valori

- controlliamo se ci sono **condizioni di validità** dei dati
  - cosa fare se **K negativo**? ERRORE?
  - cosa fare se **K > len(L)**?
    - diamo **errore**?
    - torniamo un numero di elementi **minori di K**?

- ci sono **effetti collaterali**?
  - **modifichiamo distruttivamente** L ?
  - non la modifichiamo?

- vogliamo **caratteristiche particolari** del risultato?
  - **ordinato**?
  - **disordinato**?

Scegliamo ad esempio di modificare **L** distruttivamente

- se K è sbagliato diamo un **errore**
- **modifichiamo distruttivamente** L
- il risultato può essere in **qualsiasi ordine**

Strategia di soluzione

- Per estrarre i **K** massimi dalla lista **L**

- controllo che gli argomenti siano validi altrimenti dò errore
- ripeto **K** volte
  - trovo il massimo di **L** e lo tolgo
  - lo aggiungo al risultato
- ritorno il risultato

NOTA la modifica distruttiva toglie di torno il massimo corrente e facilita il ritrovamento del successivo

Digressione: come 'lanciare' errori

```
raise ClasseDellErrore(messaggio_di_errore)

# oppure (ma così lancio solo AssertionError)
assert condizione_normalmente_vera, messaggio_di_errore_se_falsa
```

Le asserzioni sono molto usate come controllo che durante l'esecuzione tutto vada bene.

NOTA: Possono essere disattivate quindi non vanno usate per controlli sui dati in input (che invece devono essere fatti sempre).

Per semplicità userò `assert`

In [1…

```
# per estrarre i k massimi da L
def k_massimi_distruttivo(L, k):
    # controllo che k sia valido e altrimenti do errore con un messaggio esplicativo
    if not L:
        raise ValueError("L è vuota")
    # oppure
    assert len(L)>0,     "L è vuota"

    if not 0<k<=len(L):
        raise ValueError(f"K={k} non è compreso tra 1 e len(L)={len(L)}")
    # oppure
    assert 0<k<=len(L), f"K={k} non è compreso tra 1 e len(L)={len(L)}"

    risultato = []              # definisco la variabile risultato e la inizializzo = []
    for _ in range(k):          # ripeto per k volte (l'indice non mi interessa)
        M = estrai_massimo(L)   # estraggo un massimo da L eliminandolo dalla lista
```

```
        risultato.append(M)     # lo aggiungo al risultato
    return risultato            # torno i k massimi raccolti

    # oppure, usando una list-comprehension
    # return [ estrai_massimo(L) for _ in range(k) ]
```

In [2…
```
# Se lo voglio scrivere con una list-comprehension ....
```

- per estrarre ed eliminare il massimo
  - lo cerco (con **max**)
  - lo elimino (con **remove**)
  - lo torno come risultato

In [2…
```
# per estrarre ed eliminare un massimo da una lista
def estrai_massimo(L):
    assert L, "La lista è vuota, non posso cercare il massimo"
    M = max(L)    # L NON deve essere vuota
    L.remove(M)   # sono sicuro che ci sia
    return M

# Esempio
X = [1, 5, 2, 89, 2, 23]
estrai_massimo(X), X
```

Out[2…
```
(89, [1, 5, 2, 2, 23])
```

In [2…
```
# costruiamo una lista di valori casuali (con ripetizioni)
from random import choices
help(choices)
L = list(choices(range(1_000_000), k=10))
print(L)
```

```
Help on method choices in module random:

choices(population, weights=None, *, cum_weights=None, k=1) method of random.Random instance
    Return a k sized list of population elements chosen with replacement.

    If the relative weights or cumulative weights are not specified,
    the selections are made with equal probability.

[81925, 221359, 25440, 821184, 741682, 451459, 263122, 217079, 142674, 294367]
```

In [2…

```
## vediamo quanto tempo impiega per diversi valori di K e per N=100_000
LL = list(choices(range(1_000_000), k=100_000))
'''
L = LL.copy()
%timeit k_massimi_distruttivo(L, 3)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 30)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 300)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 1000)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 3000)
L = LL.copy()
%timeit k_massimi_distruttivo(L, 5000)
'''
None
help(timeit)
```

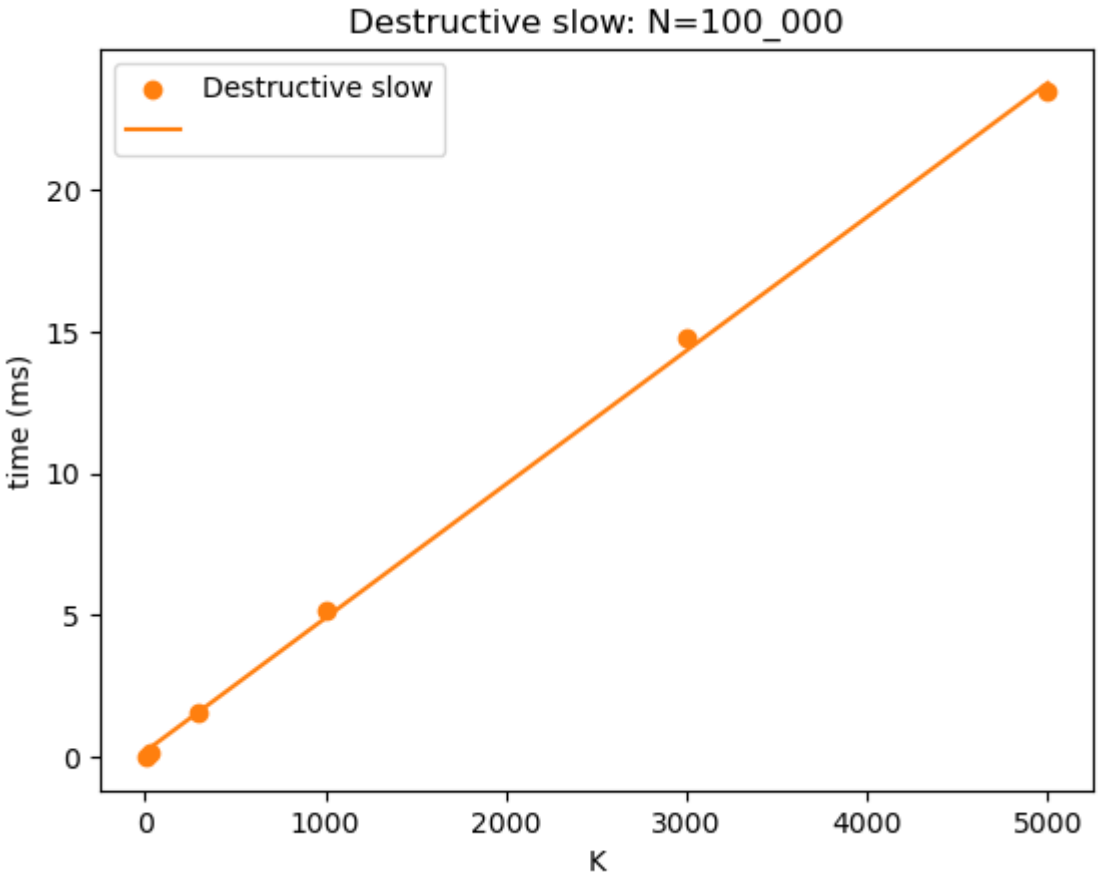
```
Help on function timeit in module timeit:

timeit(stmt='pass', setup='pass', timer=<built-in function perf_counter>, number=1000000, globals=None)
    Convenience function to create Timer object and call timeit method.
```

In [2…

```
%matplotlib inline
import matplotlib.pyplot as plt
from scipy import stats
```

```
xdata = [3, 30, 300, 1000, 3000, 5000]
ydata_slow = [3.09, 32, 318, 1030, 2700, 4020 ] # old data
ydata_slow = [timeit(lambda : k_massimi_distruttivo(LLL,X), number=5)
              for X in xdata
              for LLL in [LL.copy()]
             ]
slope, intercept, *_ = stats.linregress(xdata, ydata_slow)
plt.figure()
plt.title("Destructive slow: N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow',''])
None
```



Come vedete il tempo di esecuzione è proporzionale a **K**.

Per essere più precisi, se **N = len(L)**, nel caso peggiore:

- per **K** volte
  - trovo il massimo (scandendo la lista, tempo proporzionale a **N**)
  - lo elimino (scandendo la lista, tempo proporzionale a **N**)
  - lo aggiungo al risultato (tempo costante)

Quindi il tempo nel caso peggiore è proporzionale a **K * N**

E se invece volessimo lasciare L invariata?

Basta copiarla

In [2…
```
# per calcolare i k_massimi SENZA modificare L
def k_massimi(L, k):
    # copio la lista                tempo proporzionale a N
    L1 = L.copy()
    # uso k_massimi sulla copia     tempo proporzionale a k*N
    return k_massimi_distruttivo(L1, k)
# Il tempo di questa implementazione è proporzionale a k*N nel caso peggiore
```

In [2…
```
# di nuovo vediamo i tempi al variare di K e/o N
L = LL.copy()
'''
%timeit k_massimi(L, 3)
%timeit k_massimi(L, 30)
%timeit k_massimi(L, 300)
%timeit k_massimi(L, 1000)
%timeit k_massimi(L, 3000)
%timeit k_massimi(L, 5000)
'''
None
```

In [2…
```
xdata = [3, 30, 300, 1000, 3000, 5000]
```

```
ydata_slow2 = [3.54, 32, 317, 1040, 3090, 5060 ] # old data
ydata_slow2 = [timeit(lambda:k_massimi(L,X), number=5) for X in xdata]
slope2, intercept2, *_ = stats.linregress(xdata, ydata_slow2)
plt.figure()
plt.title("Destructive slow (orange, faster)\nNon-destructive (green, slower): N=100_000")
plt.scatter(xdata, ydata_slow, color='tab:orange')
plt.plot(xdata, [x*slope+intercept for x in xdata], color='tab:orange')
plt.scatter(xdata, ydata_slow2, color='tab:green')
plt.plot(xdata, [x*slope2+intercept2 for x in xdata], color='tab:green')
plt.xlabel('K')
plt.ylabel('time (ms)')
plt.legend(['Destructive slow','','Non-destructive',''])
None
```