

# Elementi di Informatica e Programmazione A.A. 2024-25

CANALE A

**Lezione 01**  
**01/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

Università degli Studi di Padova

Scuola di Ingegneria

Corso di Laurea in

Ingegneria Biomedica

Elementi di Informatica  
e Programmazione (EIP)

Canale 1 o A

(matricola con ultima cifra 0-4)

Prof. Marcello Dalpasso

# Sito Web del corso

- ❑ Tutte le **informazioni** relative a questo corso si trovano nel sito di Ateneo «Macroarea STEM»

**<https://stem.elearning.unipd.it>**

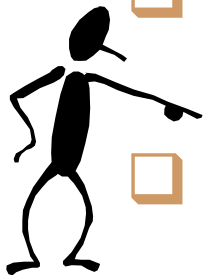
- A volte chiamato "**sito Moodle**", perché Moodle è la tecnologia web open-source su cui si basa
- Attenzione: è **diverso** dal cosiddetto "**sito esami**" (<https://esami.elearning.unipd.it>) che useremo per gli esami
- ❑ Sul sito del corso si troverà anche, giorno per giorno, **tutto il materiale didattico**
  - copia di quanto presentato a lezione (in PDF)
  - progetti per le esercitazioni di laboratorio (con soluzione)
  - questionari periodici di auto-valutazione
  - ...

# Iscrizione in Moodle

☐ Dopo aver fatto login con SSO, **bisogna iscriversi ai singoli insegnamenti** presenti nel sistema

- Chiedere informazioni ai singoli docenti per eventuali password di iscrizione

☐ **Per l'iscrizione a questo corso NON serve la password**



☐ L'iscrizione al corso in Moodle è **NECESSARIA** per

- Accedere al materiale didattico
- Accedere agli esercizi proposti
- Essere aggiornati in merito a eventuali modifiche al calendario delle lezioni o alle modalità d'esame
- Conoscere le istruzioni operative per lo svolgimento degli esami
- ...

# Chi fa parte di questo canale

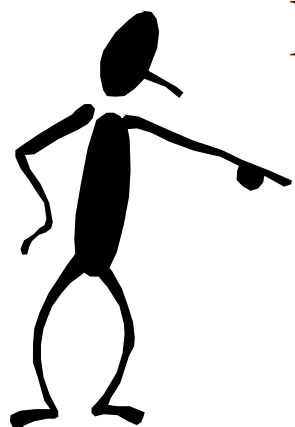
- ❑ Iscritt\* al Corso di Laurea in **Ingegneria Biomedica** con ultima cifra del numero di matricola che termina con **0, 1, 2, 3 o 4**
  - **Altrimenti l'assegnazione prevista è il canale 2 o B, tenuto quest'anno... sempre da me! ☺**
- ❑ Per ottenere il **cambio di canale**, bisogna fare richiesta **motivata** al Presidente del Consiglio di Corso di Studi (Prof.ssa Alessandra Bertoldo) **tramite la Segreteria Didattica del DEI** (NON la Segreteria di Ingegneria)
  - **Nel sito del corso trovate il link alle istruzioni per fare la richiesta**
  - **Chiedere al docente non serve a nulla!**
    - **L'attribuzione del nuovo canale va POI comunicata al docente, inoltrando la mail che riceverete e che approva il cambio di canale**
  - **Solitamente il cambio di canale ha senso solo se viene fatto per tutti gli insegnamenti del semestre (altrimenti gli orari non sono compatibili)**

# Come seguire le lezioni

- ❑ Il corso di laurea in Ingegneria Biomedica si avvale della didattica "**tradizionale**", in aula  
(con l'eventuale eccezione di alcuni insegnamenti)
  - **Le lezioni si possono seguire SOLTANTO in aula, in diretta**
    - **non remotamente, non in differita**
  - Le lezioni **NON** vengono **videotrasmesse** né **videoregistrate**
    - Salvo eventuali casi eccezionali, decisi di volta in volta:  
es. scioperi dei trasporti, condizioni meteo proibitive, ecc.
- ❑ **In questo corso, metto a disposizione il materiale didattico di anni precedenti, anche con registrazioni video**
  - Confrontando le descrizioni delle lezioni nel "Diario delle lezioni" presente nel sito del corso per ciascun anno accademico (oltre a quello attuale, aggiornato giorno per giorno), potrete identificare le registrazioni utili (ad esempio, in caso di vostra assenza)

# Frequenza alle lezioni

- ❑ La frequenza alle lezioni e alle esercitazioni **NON** è obbligatoria però



- **Tutte le informazioni di carattere organizzativo che vengono fornite A LEZIONE O NEL SITO devono essere note**

- ❑ La frequenza alle **lezioni in aula** è

- **VIVAMENTE CONSIGLIATA**

- ❑ La frequenza in **laboratorio** è

- **VIVAMENTE CONSIGLIATA**



# Come fare domande

- ❑ La maggior parte dei dubbi che sorgono a lezione sono più facilmente oggetto di **domande/risposte in posta elettronica, dopo la lezione**
- ❑ Molto spesso si tratta di quesiti di una certa complessità, ai quali difficilmente si può rispondere "al volo" in modo didatticamente efficace
  - La domanda tipica riguarda codice di programmazione
  - L'elaborazione di una risposta sensata richiede tempo, che vi dedico volentieri al di fuori delle ore di lezione
- ❑ Sembra strano, ma vedrete che è così... **la maggior parte delle interazioni, per questo corso, avviene in posta elettronica** (con eventuali integrazioni audio/video, mie o vostre)
  - È la natura della materia di insegnamento che spinge a questo
  - **Questo NON significa che non si possono fare domande durante la lezione!**

# Docente ([www.dalpasso.net](http://www.dalpasso.net))

□ (Prof. Ing.) Marcello **Dalpasso**

- Professore Associato nel settore dei Sistemi per l'Elaborazione dell'Informazione

□ **DEI - Dip. di Ingegneria dell'Informazione**

- **Via Gradenigo 6/A** (Padova), edificio principale (DEI/G)

□ Ricevimento studenti (anche via ZOOM)  
**solo su appuntamento**

- molto più efficiente e flessibile di un orario settimanale fisso
- l'appuntamento va richiesto tramite posta elettronica

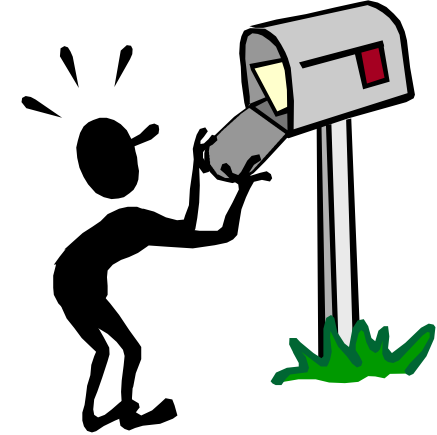
□ Il metodo più efficace di interazione diretta con il docente è la *posta elettronica*

**[marcello.dalpasso@unipd.it](mailto:marcello.dalpasso@unipd.it)**

Attenzione: il mio **cognome** è (stranamente) **una parola unica** (importante nei motori di ricerca)

# Richiesta di colloquio

- ❑ Illustrate **bene** il problema in **posta elettronica**, eventualmente allegando algoritmo o codice, anche in PDF o con immagini/foto/video



Vedremo cos'è il codice sorgente

- Ma **se è codice, allegare IL FILE SORGENTE**
  - **Il punto precedente è MOLTO importante: se avete scritto del codice, non ha senso che mi mandiate una foto dello schermo con il codice... perché poi lo devo ricopiare!**
  - Dovete **sempre** scrivere usando il vostro indirizzo ufficiale di ateneo (...[@studenti.unipd.it](mailto:@studenti.unipd.it))
  - Non sempre è necessario un colloquio, spesso riesco a rispondere adeguatamente in posta elettronica, eventualmente con registrazioni audio/video
- ❑ **Si può contattare il docente anche per problemi non strettamente riguardanti il corso o la didattica**

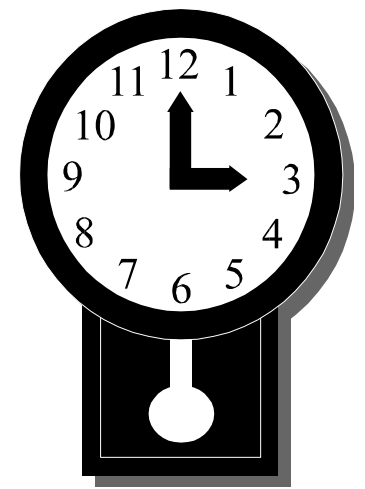
# Interazioni con il docente

- ❑ **Durante il corso, normalmente rispondo alla posta elettronica entro 24 ore** (eventualmente anche soltanto per dirvi "ho ricevuto la domanda e risponderò al più presto")
  - Quindi, dopo avermi scritto, NON scrivetemi di nuovo prima che siano trascorse 24 ore, pensando "forse non ha ricevuto il mio messaggio precedente..."
    - A volte i messaggi di posta elettronica non vengono consegnati correttamente senza che il mittente riceva notifiche (chi si occupa di reti di calcolatori direbbe che si tratta, tecnicamente, di un "servizio non affidabile", come il servizio SMS dei cellulari), ma DOVETE aspettare 24 ore
- ❑ Terminato il corso e la prima sessione d'esami, il tempo di risposta può aumentare, fino a "qualche giorno", al limite una settimana (come se fosse il ricevimento settimanale)
  - Mie eventuali assenze più lunghe saranno segnalate nel sito del corso, con eventuale individuazione di un "sostituto" per le interazioni didattiche

# Curriculum vitæ et studiorum

- ❑ Nato nel 1965 a Ferrara (vivo a Ferrara)
- ❑ Maturità scientifica
- ❑ Laurea in Ingegneria Elettronica a UniBO
  - Esame di Stato per l'abilitazione alla professione di Ingegnere
- ❑ Dottorato di Ricerca in Ingegneria Elettronica e Informatica a Unibo
- ❑ Ricercatore Universitario a UniPD dal 1998
- ❑ Professore Associato a UniPD dal 2004

# Orario



## ❑ Lezioni in aula Ve (ma consultate l'orario)

- Martedì 10.30-12.30
- Mercoledì 10.30-12.30
- Venerdì 16.30-18.30

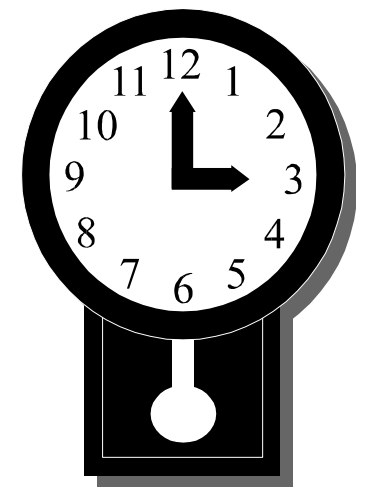
## ❑ Un'ora di didattica universitaria a UniPD ha una durata compresa tra 45 e 60 minuti, "in relazione alle esigenze didattiche della lezione" (cioè "decide il docente"...😊)

- Decisione del Senato Accademico UniPD del 09/11/2015

## ❑ Quindi **2 ore di lezione durano 90-120 minuti**

- **Io arriverò al minuto 30** (ad esempio, 10.30) e inizierò dopo qualche minuto (tempo tecnico di "setup"), facendo solitamente lezione **SENZA** pausa intermedia per **90/100 minuti**
  - Quando il tempo "sta finendo" **NON rumoreggiate** in aula: chi deve/vuole andare via, vada via in silenzio

# Esercitazioni in laboratorio



- Un turno di 4 ore ogni settimana
  - **Venerdì ore 8.30-12.20**
- Esercitazioni **guidate** ma **non assistite**
  - **Il docente NON è presente**  
(perché il laboratorio non è "creditizzato"...),  
ma **propone in Moodle esercizi da svolgere**
- Si svolgono nell'**Aula Didattica Taliercio (ADT)**
  - Fiera di Padova, Padiglione 14, Aula 14L, ingresso da **Via Tommaseo 59**
  - **Durante la prima settimana di lezione non ci saranno esercitazioni**
    - Vi comunicherò se ci saranno **nella seconda settimana (probabilmente sì)**, altrimenti certamente a partire dalla terza settimana
- **Entro venerdì 4 ottobre**, chi vuole partecipare alle esercitazioni fin dall'inizio deve **NECESSARIAMENTE** iscriversi nelle apposite liste presenti nel sito del corso, che ci servono per creare le credenziali di accesso al laboratorio
  - **Chi ha già deciso che non frequenterà il laboratorio, non si iscriva**
  - Chi non ha ancora deciso, potrà iscriversi più avanti per partecipare a partire dalla seconda esercitazione
  - Chi si iscrive ma non partecipa, non avrà penalizzazioni...
  - Per chi «non è in corso» il posto non è garantito, ma di solito c'è posto

# Tutor Junior in laboratorio

□ Durante le esercitazioni di laboratorio saranno presenti **Tutor Junior (TJ)** qualificat\*

- Quattro TJ per ogni canale
- **Non faranno "lezione"** ma **vi aiuteranno** a risolvere gli esercizi assegnati e, soprattutto, ad imparare a risolverli da soli...
- Chi non va in laboratorio avrà comunque accesso a tutto il materiale didattico relativo, ma **non avrà l'aiuto dei/delle TJ**



# Esercitazioni solo in laboratorio?

È sostanzialmente **indispensabile** esercitarsi anche "a casa" con un **computer (fisso o portatile)**

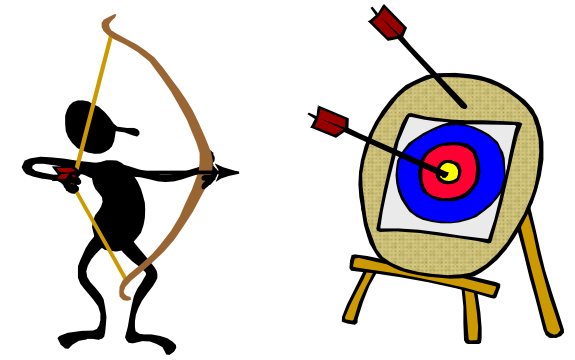
- **Almeno altre 4 ore alla settimana**
- Windows, Linux, Mac... va bene tutto
- **Non serve un computer particolarmente potente**
- Nel sito del corso trovate istruzioni per la configurazione del software (gratuito) che serve
- **Le esercitazioni proposte settimanalmente dal docente richiedono un impegno al calcolatore di circa 8 ore**

# Calendario delle lezioni



- ❑ Lezioni: 30 settembre 2024 – 17 gennaio 2025
  - La durata del corso è di **72 ore di lezione (più le esercitazioni in laboratorio)**, può darsi che finisca (ben) prima del 17 gennaio, almeno per quanto riguarda le lezioni in aula
- ❑ Vacanze natalizie
  - 21 dicembre 2024 (compreso) – 6 gennaio 2025 (compreso)
- ❑ Altre festività:
  - **1 novembre 2024, festa nazionale (venerdì)**
  - 8 dicembre 2024, festa nazionale (domenica)
- ❑ Può succedere che, con il massimo preavviso possibile, alcune lezioni vengano annullate (e, poi, recuperate, anche "fuori orario"), per malattia o altri impegni inderogabili del docente
  - **Consultate sempre il sito e la posta elettronica!!!**

# Obiettivi formativi



□ Il corso ha l'obiettivo di

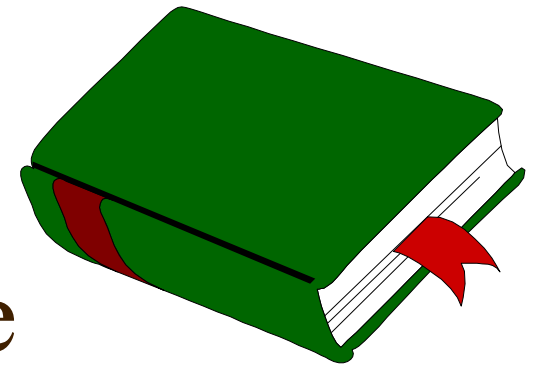
- presentare le **basi teoriche** dell'informatica
- proporre un **approccio ingegneristico** e progettuale alla programmazione
- fornire una **visione professionale** alla risoluzione dei problemi
- **utilizzare concretamente** le caratteristiche del **linguaggio di programmazione Python**

# Programma del Corso



- ❑ Concetti fondamentali della programmazione
- ❑ Sintassi di base del linguaggio Python
  - Operatori, espressioni, strutture di controllo, funzioni, oggetti e classi, ereditarietà e ricorsione
- ❑ Strutture fondamentali per la gestione dei dati
  - variabili, liste, matrici, classi
- ❑ Introduzione alle strutture dati più complesse
  - dizionari, insiemi
- ❑ **Cenni** alla programmazione grafica
- ❑ Algoritmi fondamentali di ordinamento e ricerca
- ❑ **Realizzazione di (semplici) progetti di programmazione nel laboratorio didattico**

# Libro “di testo”



Cay Horstmann, Rance D. Necaise

*“Concetti di informatica e fondamenti di Python”*

***SECONDA EDIZIONE***

Ed. Maggioli 2019, ISBN 978-88-916-3543-3

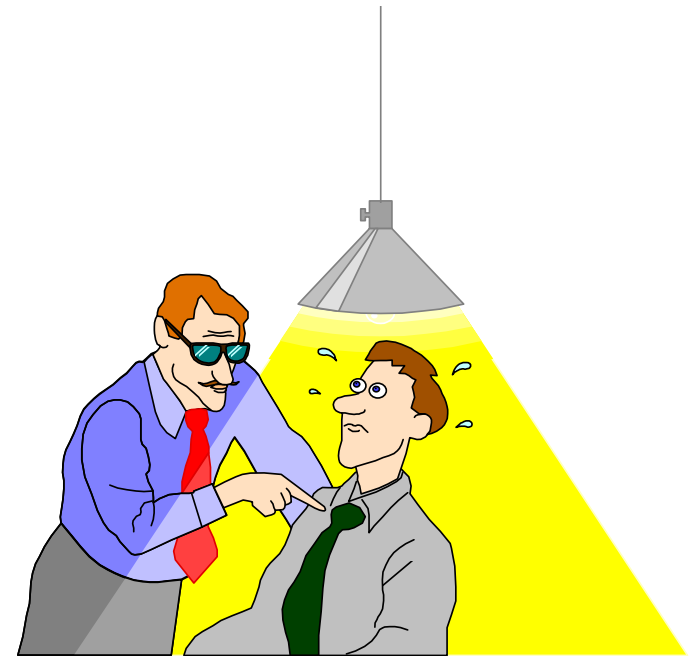
**Non è “obbligatorio”,  
ma NON chiedete se altri libri vanno bene...  
(la prima edizione dello stesso libro è accettabile)**

**Attenzione: siete all’Università...  
il “libro di testo” NON verrà seguito passo-passo e NON  
contiene tutto (e solo) il materiale presentato a lezione, non  
fornirò riferimenti ai numeri di pagina...**

# Modalità d'esame

- ❑ **Due appelli a gennaio/febbraio, un appello a giugno/luglio e un appello a settembre**

Già fissati, vedere il sito del corso

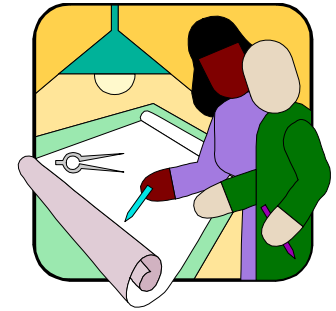


- ❑ L'esame consiste in due prove da sostenere NELLO STESSO APPELLO (non soltanto nella stessa sessione...)
  - Questionario e programmazione
  - Non c'è una prova orale
- ❑ **Altri dettagli sugli esami si possono (anzi, si devono!) consultare sul sito**

# "Compitini" ?

- ❑ In questo corso **NON** ci sono prove di accertamento intermedie

# Qui si progetta!



## ❑ **ATTENZIONE:**

**questo è un corso di PROGETTAZIONE**

- ❑ Conoscere la teoria (cioè "avere studiato") è soltanto una condizione **NECESSARIA** per superare l'esame, non certo sufficiente
  - È (anche) per questo motivo che non c'è l'esame orale
- ❑ **Bisogna dimostrare di saper affrontare e risolvere problemi mai affrontati durante il corso**
  - **anche se ovviamente simili ad altri già visti e risolti**



# Studio "di gruppo"

- ❑ Nella preparazione degli esami di progettazione, come questo, è **molto utile lavorare in gruppo** (non durante l'esame 😊), in particolare per la parte di esercizi
  - Gli ingegneri dovranno certamente lavorare in "team", tanto vale abituarsi fin da subito
- ❑ **Non fate gli esercizi insieme!!**
  - Scegliete un esercizio, risolvervelo **in autonomia**, poi confrontate e discutete insieme le soluzioni trovate, per decidere quale sia la migliore e quali errori ci siano in altre
- ❑ Fate gruppi di piccole dimensioni (4 o 5), possibilmente "ben assortiti": quelli meno bravi trarranno vantaggio da quelli più bravi, che, a loro volta, avranno la soddisfazione di aiutare gli altri e faranno l'utilissimo esercizio di individuare gli errori

# Quanto bisogna studiare?

❑ Risposte banali

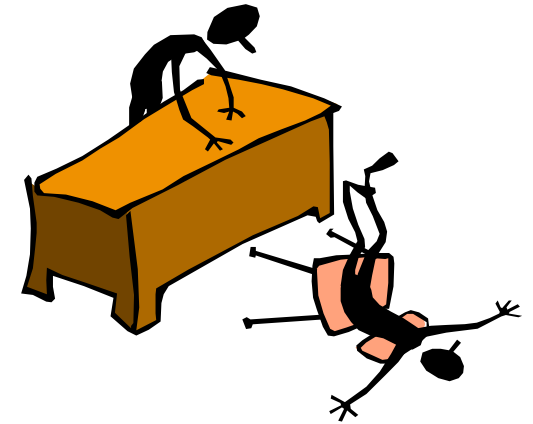
- **MOLTO, ABBASTANZA, POCO...**

❑ Risposta da ingegnere  
per (aspiranti) ingegneri

- dipende dai crediti

❑ Questo è un esame da **9 crediti**,  
quindi facciamo qualche calcolo...

# Quanto bisogna studiare?



- Un credito = 25 ore di lavoro
  - totale corso EIP =  $25 \times 9 = 225$  ore di lavoro
  - in aula + laboratorio =  $72 + 48 = 120$  ore
  - studio finale prima dell'esame (esempio) = 21 ore
  - **studio settimanale (per 12 settimane) = 7 ore**
- Lo studio **settimanale** comprende anche **pratica al calcolatore (OLTRE al laboratorio)**
  - **circa 3 ore di studio e 4 ore di pratica**
- Attenzione: vale per uno “studente medio”
  - **Non è possibile che siate TUTTI migliori dello studente medio... studiate statistica!!!**
- Attenzione: vale per “superare l'esame”, non per prendere 30 !!

# Convienne rifiutare un voto?

❑ **Quanto influisce il voto  $V$  di un singolo esame sul voto di laurea  $F$  ?**

■ **Dipende da**

- **Numero di crediti  $C$  del corso**
- **Media  $M$  (pesata per crediti) degli altri esami**
- **$k$  = eventuali bonus previsti per il corso di laurea**

❑  **$F_V = k + (110/30) * [(180 - C) * M + C * V] / 180$**

❑  **$F_{V_1} - F_{V_2} = (110/30) * C * (V_1 - V_2) / 180$**

■ **Non dipende da  $M$  degli altri esami...**

■  **$\Delta F = \Delta V * C * 0.02037$**

■  **$\Delta F = \Delta V * 0.1833$  (per  $C = 9$ )**

# Convienne rifiutare un voto?

- ❑ Passando **da 18 a 30** nel voto di questo corso, il voto finale di laurea **aumenta di 2.2**
  - Ma di solito l'alternativa non è tra 18 e 30...
- ❑ Il voto finale di laurea aumenta di 0.1833 per ogni punto in più nel voto di un esame da 9 crediti
  - Rifiutando 22 e prendendo poi 25 si aumenta il voto finale di laurea di poco più di mezzo punto
    - con l'arrotondamento all'intero più vicino, si aumenta o di un punto o di... niente
  - Ma rifiutando 22 si può anche poi prendere 19...

# Qualche altro consiglio...

- ❑ Per chi **non** ha esperienza di programmazione
  - **Questo corso non ha prerequisiti**, chiunque lo può seguire con profitto, serve solo... impegno!
  - È fondamentale frequentare regolarmente le lezioni **e svolgere le esercitazioni**
  - È **necessario** dedicare alle esercitazioni **più tempo delle 4 ore previste per le esercitazioni**
    - Svolgere **con regolarità**, settimana dopo settimana, gli esercizi proposti, non accumularli... anche perché servono per seguire le lezioni successive con profitto

# Qualche altro consiglio...

- ❑ Per chi sa già programmare (o pensa di...)
  - **Attenzione a non sottovalutare l'impegno richiesto per il corso**
  - Questo corso **non** ha l'obiettivo di formare programmatori professionisti
    - è un corso DI BASE
    - **ma NON SI STUDIA SOLO PROGRAMMAZIONE**

# Qualche altro consiglio...

- ❑ **Non** fate l'errore di considerare questo corso come **estraneo** al vostro indirizzo di studio, quello che studierete qui vi servirà (quasi) sicuramente per
  - Altri esami dei prossimi anni
  - La tesi di laurea
  - Il mondo del lavoro
- ❑ E poi... superare questo esame è **obbligatorio** per laurearsi! 😊
  - **Non fate l'errore di "lasciarlo per ultimo"...**



# Valutazione studentesca della didattica

- ❑ Come al solito, a fine corso ci sarà la "valutazione studentesca della didattica", così come previsto dal regolamento di ateneo
- ❑ Un paio di commenti ricevuti lo scorso anno (veri!)
  - Insegnante molto preparato e in grado di far comprendere molto bene ciò che spiega
  - Il professore non è in grado di trasmettere nessuna informazione e conoscenza



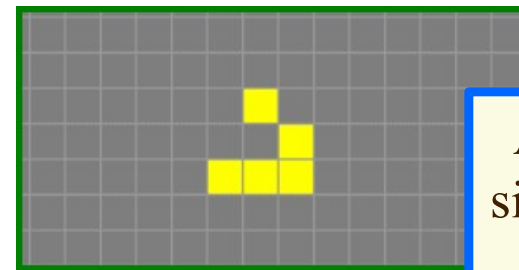
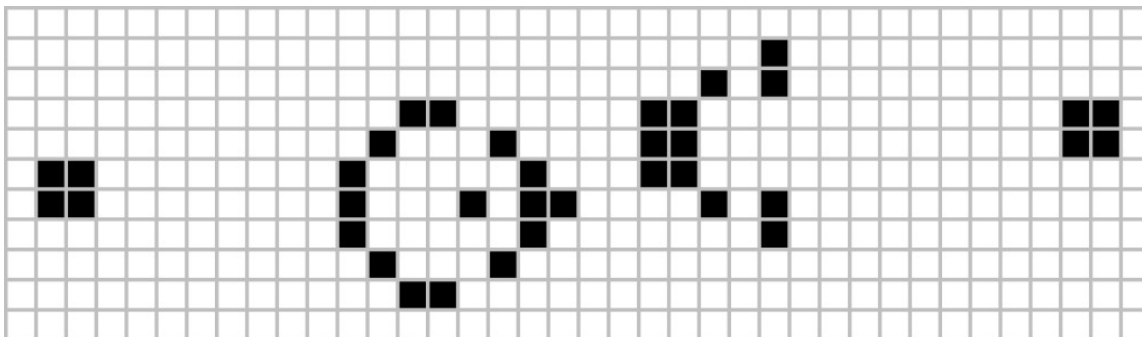
# Ma cosa saprete fare alla fine?

□ Esempio: Simulazione cellulare

## ■ Game of Life (di John Conway)



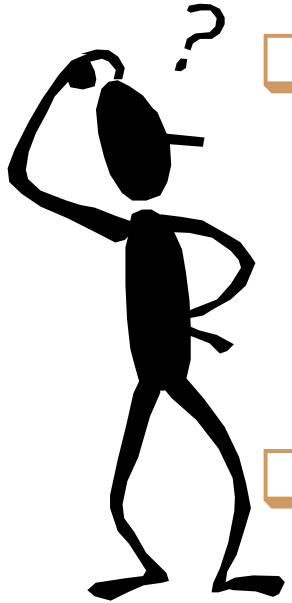
- Qualsiasi cellula viva con meno di due cellule vive adiacenti **muore, per isolamento**
- Qualsiasi cellula viva con due o tre cellule vive adiacenti **sopravvive** alla generazione successiva
- Qualsiasi cellula viva con più di tre cellule vive adiacenti **muore, per affollamento**
- In qualunque posizione vuota avente esattamente tre cellule vive adiacenti **nasce una nuova cellula, per riproduzione**



Aliante che  
si sposta ogni  
4 step

**Lezione 02**  
**02/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Consapevolmente incompleto...



- ❑ Quando vedete questa figura "perplessa", voglio indicare un'informazione **consapevolmente incompleta** e forse poco chiara... che verrà chiarita in seguito
- ❑ Se nel punto in cui vedete la figura vi rimane qualche dubbio... è normale!
- ❑ Se, invece, avete dubbi ma non c'è la figura... approfondite e insistete finché non avete capito 😊

# Le prime domande

☐ Cos'è un computer?

☐ Cos'è un programma?

☐ Cos'è la programmazione?

☐ Cos'è un algoritmo?

Cos'è un computer?



# Hardware e Software

- ❑ Un sistema di elaborazione delle informazioni (*computer* o calcolatore) è composto da elementi fisici (materiali) e logici (informazioni)
  - La parte fisica (elettronica, magnetica, ottica, ecc.) è detta **hardware** (letteralmente, *ferramenta*)
  - La parte logica (*dati* e *programmi*) è detta **software**
- ❑ I *dati* rappresentano qualunque tipo di informazione
  - **Numeri, testi, immagini**, suoni, filmati, ecc.
- ❑ I *programmi* sono formati da insiemi di istruzioni (elementari) che vengono eseguite in un ordine stabilito
- ❑ Ci occuperemo (quasi) esclusivamente di software

# Cos'è un computer?

- ❑ Oggi molte persone usano computer per lavoro o per svago
  - Sul lavoro, i computer sono ottimi per svolgere operazioni ripetitive o noiose, come effettuare calcoli, impaginare testi o simulare l'evoluzione di un sistema biologico
  - Nel gioco, i computer sono ottimi per coinvolgere al massimo l'utente-giocatore, perché possono riprodurre con estremo realismo suoni e sequenze di immagini
  - Entrambi gli ambiti sono enormemente arricchiti dalle interazioni in rete
- ❑ In realtà, tutto questo non è merito propriamente del computer, ma dei *programmi* che su di esso vengono eseguiti



# L'architettura di un computer

# L'architettura di un computer

- ❑ Per capire i meccanismi di base della programmazione è necessario conoscere gli **elementi hardware** che costituiscono un computer
- ❑ Prendiamo in esame il *Personal Computer* (PC), ma anche i computer più potenti hanno un'**architettura** molto simile



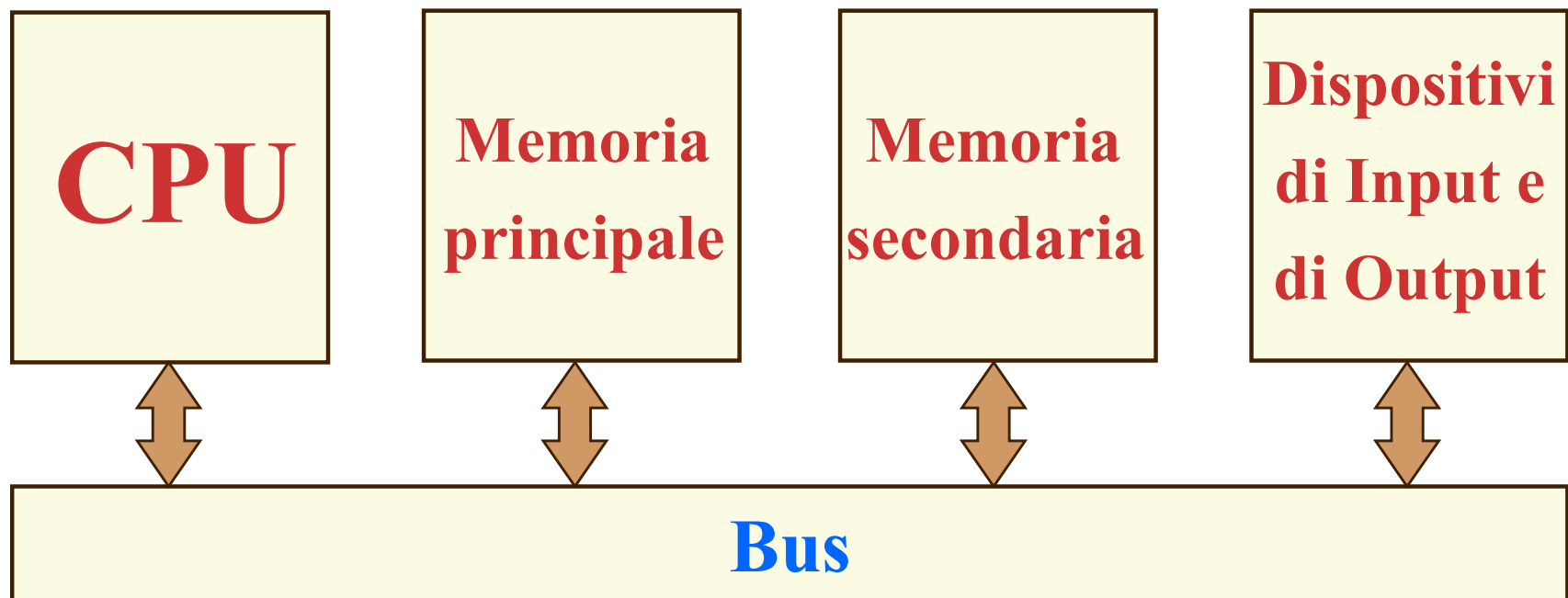
# Il modello di von Neumann

# Il modello di John von Neumann

- ❑ Nel 1946, John **von Neumann** elaborò un modello teorico dell'architettura di un elaboratore che è tuttora valido e molto utilizzato
- ❑ La grande maggioranza degli elaboratori odierni ha un'architettura che può essere (più o meno facilmente) ricondotta al modello di von Neumann
  - le eccezioni più importanti sono alcune macchine ad elaborazione parallela
- ❑ Il modello è importante in quanto schematizza in modo *omogeneo* situazioni diverse
  - lo presentiamo in una versione un po' modificata rispetto alla formulazione originale, ma più adatta alla didattica

# Il modello di John von Neumann

- L'architettura di von Neumann è composta da **quattro blocchi** comunicanti tra loro per mezzo di un **bus**, un canale di scambio di informazioni

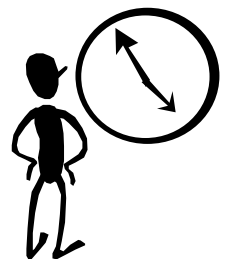


# La memoria del computer

- ❑ La memoria serve a *immagazzinare* (cioè a **preservare nel tempo**) **dati** e **programmi** all'interno del computer
- ❑ È suddivisa in *celle* o locazioni di memoria, ognuna delle quali è identificata univocamente da un **indirizzo**
- ❑ Ogni cella contiene un numero predefinito di *bit*, solitamente uguale a 32 o 64
  - Un bit è un **dato elementare**, l'unità elementare di informazione, che può assumere due valori, **convenzionalmente** chiamati **zero** e **uno**
    - In relazione alla tecnologia di memorizzazione, "zero" e "uno" sono due diversi valori di una specifica grandezza fisica (elettrica, magnetica, ottica, ecc.)
  - Un **insieme di otto bit** si chiama **byte** ed è l'unità di misura della memoria (ovviamente se ne usano anche multipli, ad esempio 512 MByte)
  - Vedremo più avanti le differenze tra memoria **principale** e **secondaria**

# L'unità centrale di elaborazione

- ❑ Dal punto di vista logico, la **CPU** (*Central Processing Unit*) è costituita da tre parti principali
  - l'**unità logico-aritmetica** (ALU, *arithmetic logic unit*), che "fa i calcoli"
  - l'**unità di controllo** (CU, *control unit*) che governa il funzionamento della CPU stessa
  - un insieme di **registri**, che sono celle ad accesso molto veloce per la memorizzazione temporanea dei dati (in pratica, un'altra piccola memoria interna alla CPU)
- ❑ Il funzionamento della CPU è **ciclico** e il periodo di tale ciclo viene scandito dall'orologio di sistema (**clock**), la cui **frequenza** costituisce una delle caratteristiche tecniche più importanti della CPU (es. 1 GHz, un miliardo di cicli al secondo, cioè periodo di 1 ns)



# Ciclo di funzionamento della CPU

- ❑ Ogni **ciclo di funzionamento** è composto da tre fasi
  - **accesso**: lettura dell'istruzione da eseguire e sua memorizzazione nel *registro istruzione*
  - **decodifica**: decodifica dell'istruzione da eseguire
  - **esecuzione**: esecuzione dell'istruzione

Si parla di ciclo *fetch-decode-execute*

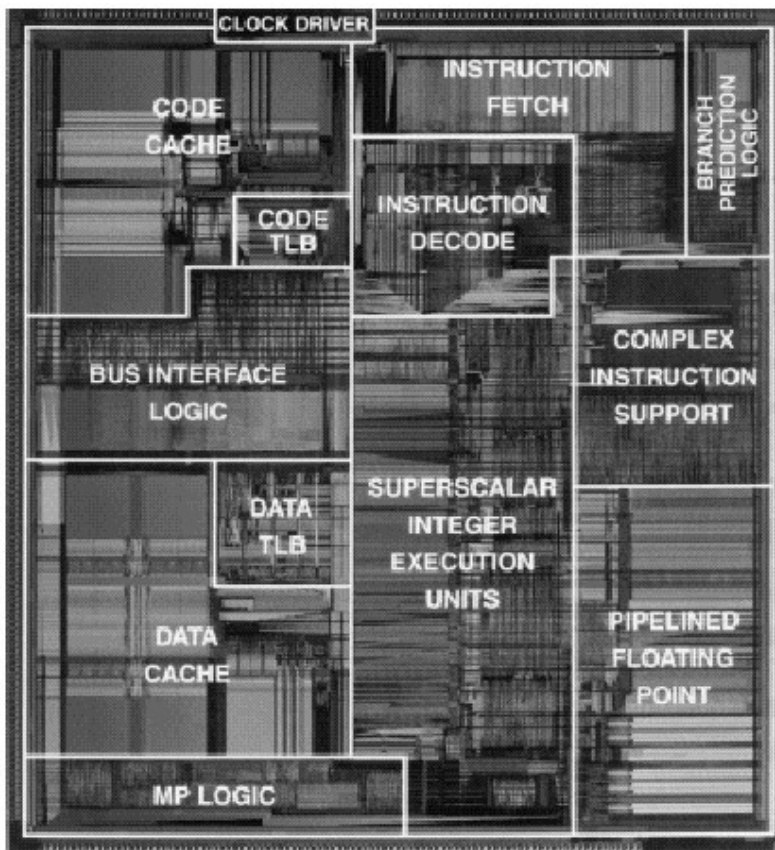
- ❑ La posizione (o indirizzo) dell'istruzione a cui si accede durante la fase di *fetch* è contenuta nel *contatore di programma* (*program counter*, PC)
  - viene automaticamente incrementato di un'unità ad ogni ciclo, in modo da *eseguire istruzioni in sequenza*, cioè istruzioni memorizzate in celle di memoria aventi indirizzi *consecutivi*



# L'unità centrale di elaborazione

□ L'unità centrale di elaborazione:

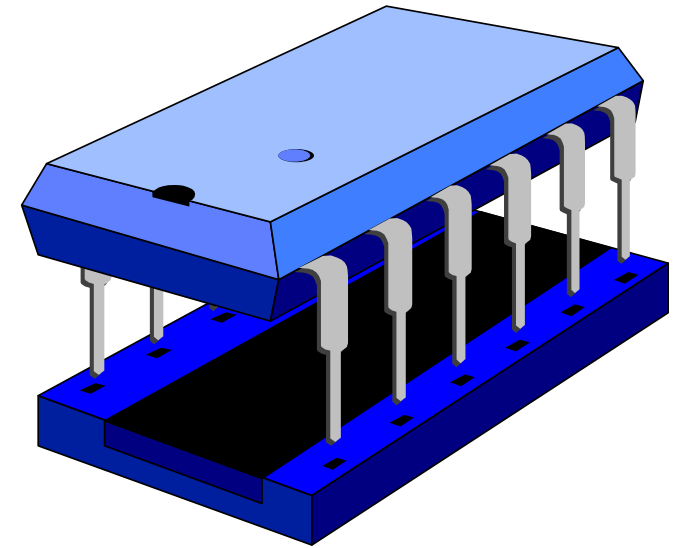
- individua ed esegue le istruzioni del programma



- effettua elaborazioni aritmetiche e logiche con la sua unità logico-aritmetica (**ALU**)
- reperisce i dati dalla memoria esterna e da altri dispositivi periferici e ve li rispedisce dopo averli elaborati
- è costituita da uno o più *chip* (microprocessori)

# Il chip della CPU

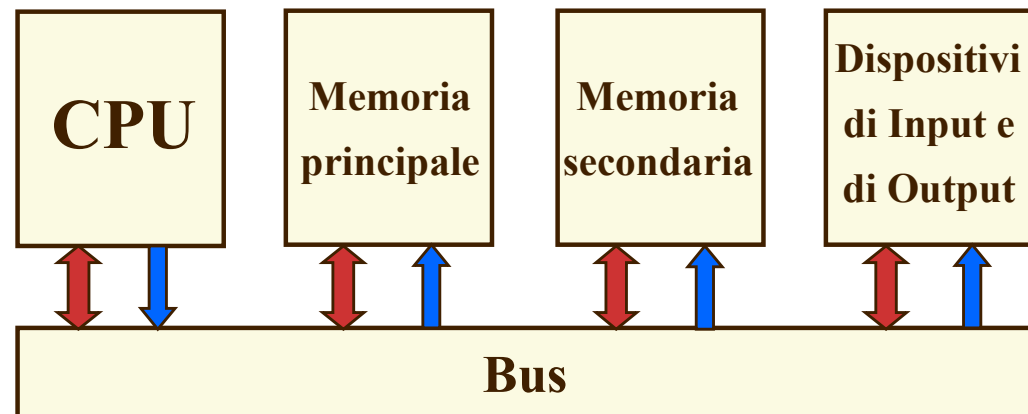
- ❑ Un chip, o *circuito integrato*, è un componente elettronico con connettori metallici esterni (*pin*) e collegamenti interni (*wire*), costituito principalmente di silicio e alloggiato in un contenitore plastico o ceramico (*package*)
- ❑ I collegamenti interni di un chip sono molto complicati e tipicamente in un chip sono presenti *alcuni miliardi di transistori* tra loro interconnessi



# Il bus nel modello di von Neumann

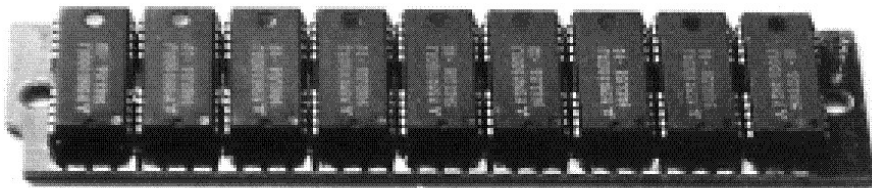
- ❑ Il **bus** è in realtà costituito da tre bus distinti
  - bus dei ***dati***
  - bus degli ***indirizzi***
  - bus dei ***segnali di controllo***
- ❑ Sul bus (bidirezionale) dei dati viaggiano dati **da e verso la CPU**
- ❑ Sugli altri bus viaggiano indirizzi e segnali di controllo che **provengono soltanto dalla CPU**

Attenzione alla  
direzione delle  
freccie...



# La memoria primaria

- ❑ La memoria *primaria* è **veloce** ma **costosa (5€/GByte)**
  - Cioè è **più** veloce e **più** costosa della memoria secondaria
- ❑ È costituita da **chip di memoria** realizzati con la stessa tecnologia (al silicio) utilizzata per la CPU ed è suddivisa in due parti, con funzionalità distinte
  - memoria *di sola lettura* (**ROM**, *Read-Only Memory*)
  - memoria ad accesso casuale (**RAM**, *Random Access Memory*)
    - **accesso casuale** significa che *il tempo per accedere ad un dato non dipende dalla sua posizione nella memoria*
    - dovrebbe chiamarsi memoria *di lettura e scrittura*, perché in realtà anche la ROM è ad accesso casuale, mentre ciò che le distingue è la possibilità di scrivervi



# La memoria RAM

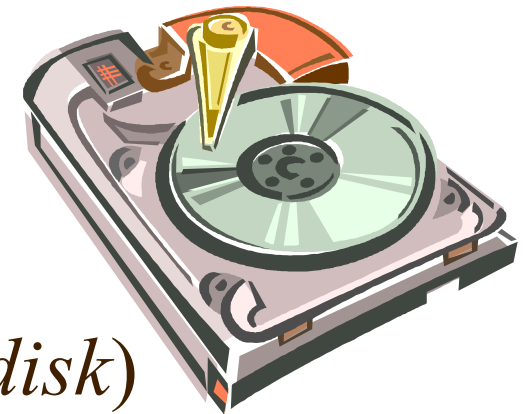
## □ La memoria ad accesso casuale, **RAM**

- è una memoria che consente la *lettura* e la *scrittura* dei dati e dei programmi in essa contenuti
- contiene dati in fase di modifica e programmi in fase di esecuzione
- **perde il proprio contenuto quando si spegne il computer** (è un supporto di memorizzazione *volatile*)

# La memoria ROM

- ❑ La memoria di sola lettura, **ROM**
  - conserva i dati e i programmi in essa memorizzati anche quando il computer viene spento
    - è una memoria *non volatile* o permanente
  - contiene i programmi necessari all'avvio del computer, programmi che devono essere *sempre disponibili*
    - nei PC, tali programmi prendono il nome di **BIOS** (**B**asic **I**nput/**O**utput **S**ystem)
- ❑ Una memoria ROM può essere scritta una sola volta, al momento della sua fabbricazione
  - In realtà, il BIOS è *aggiornabile* perché risiede in una **EPROM** (o tecnologia simile), memoria riscrivibile (lentamente e “poche” volte, decine di migliaia), comunque molto diversa da una RAM

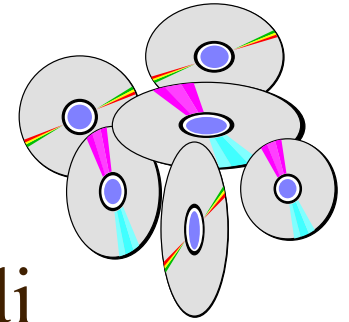
# La memoria secondaria



- ❑ La memoria *secondaria* (o *di massa*) è di solito un **disco rigido** (o disco fisso, *hard disk*) ed è un supporto *non volatile* e *meno costoso* della memoria primaria (circa **100** volte, come prezzo per byte) ma decisamente più lento (tempo di accesso dell'ordine di qualche ms anziché qualche ns)
  - Circa 50€/TByte = 0.05€/GByte il disco, 5€/GByte la RAM
  - Programmi e dati risiedono sul disco rigido e vengono caricati nella RAM quando necessario, per poi tornarvi aggiornati se e quando necessario
- ❑ Un disco rigido è formato da piatti rotanti rivestiti di materiale magnetico, con testine di lettura/scrittura
  - Processo simile a quello dei (vecchi) nastri audio o video



# La memoria secondaria



□ Negli anni si sono diffusi anche altri tipi di memoria secondaria a tecnologia **ottica**

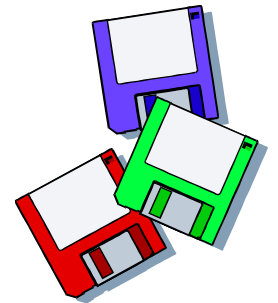
- **CD-ROM** (*Compact Disc Read-Only Memory*), viene letto da un dispositivo laser, esattamente come un CD audio; ha una elevata capacità ed è molto economico e affidabile; è un supporto di sola lettura, utilizzato per distribuire programmi e informazioni (1979)
- **CD-R** (*Compact Disc Recordable*), utilizza una tecnologia simile al CD-ROM ma può essere scritto dall'utente (una sola volta; più volte se CD-RW) (1990)
- **DVD**, ha rappresentato la nuova frontiera per questa tecnologia, con elevatissima capacità (1997)
  - Blu-Ray (2002), capacità ancora più elevata, ma troppo costoso





# La memoria secondaria

- Sono (meno) usati anche altri tipi di memoria secondaria a tecnologia magnetica
  - *floppy disk* (**dischetto** flessibile), di capacità limitata ma con il vantaggio di poter essere agevolmente rimosso dal sistema e trasferito ad un altro sistema (dispositivo di memoria *esterno*)
    - In pratica non sono più utilizzati, soppiantati dalle «chiavette USB»
  - *tape* (**nastri** per dati), di capacità elevatissima, molto economici, ma molto lenti, perché *l'accesso ai dati è sequenziale* anziché casuale (bisogna avvolgere o svolgere un nastro invece che spostare la testina di lettura sulla superficie di un disco)
    - Perfetti per attività di *backup* di grandi quantità di dati



# La memoria secondaria

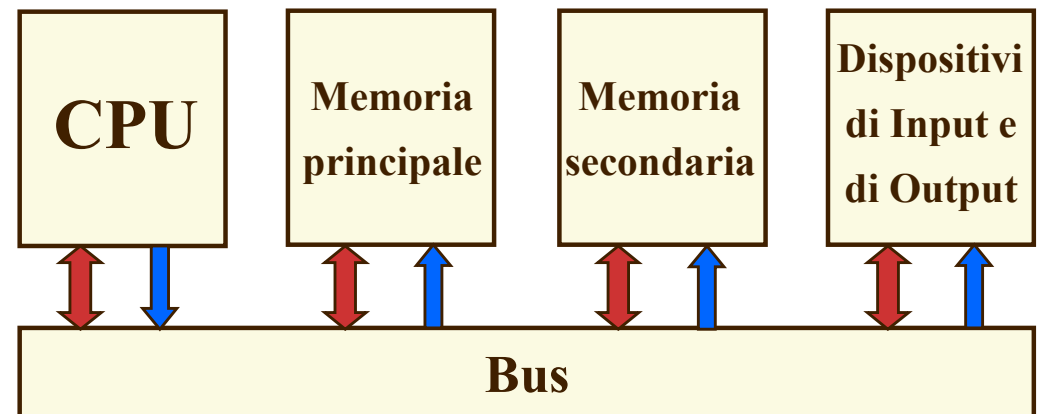
□ Sono molto diffusi anche altri tipi di memoria secondaria a tecnologia *microelettronica*: le “*Chiavette*” *USB*

- Usano una tecnologia molto simile a quella (EPROM) usata per il BIOS dei PC
  - Hanno sostituito i floppy disk per la portabilità
  - Più lente dei dischi rigidi, più veloci dei CD
  - Costo 0.15€/GByte, intermedio tra RAM e disco rigido, ma **non volatile** e soprattutto **portatile**!
    - "molto" portatile perché di piccole dimensioni
  - Ormai significativamente sostituiti dalla memorizzazione "nel cloud"

**Lezione 03**  
**04/10/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Dispositivi periferici di interazione

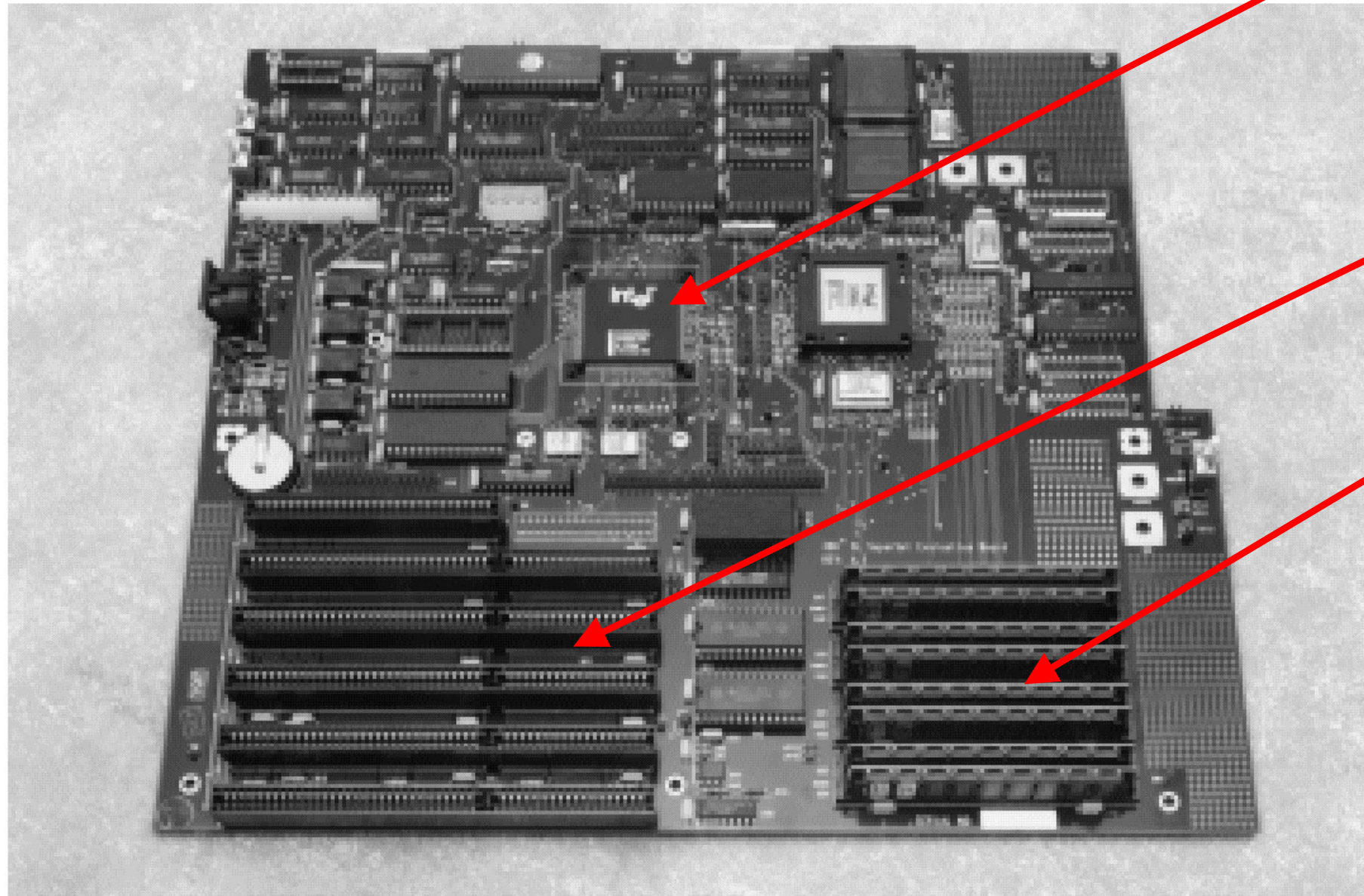
- ❑ L'interazione fra l'utente umano ed il computer avviene mediante i cosiddetti *dispositivi periferici di Input/Output* (dispositivi di I/O)
- ❑ Tipici dispositivi di *input* sono la **tastiera**, il **mouse** (dispositivo di puntamento), il **microfono** (per impartire comandi vocali), il **joystick** (per i giochi), lo **scanner** (per la scansione digitale di documenti e immagini)
- ❑ Tipici dispositivi di *output* sono lo **schermo** (*monitor*), le **stampanti**, gli **altoparlanti**
- ❑ Importanti dispositivi di *input/output* (bidirezionali) sono la **connessione di rete** e il **touchscreen**



# La scheda madre di un PC

- ❑ La CPU, la memoria primaria (RAM e ROM) e i circuiti elettronici che controllano il disco rigido e altri dispositivi periferici sono interconnessi mediante un insieme di linee elettriche che formano un *bus*
- ❑ I dati transitano lungo il bus, dalla memoria e dai dispositivi periferici verso la CPU, e viceversa
- ❑ All'interno del PC si trova la *scheda madre* (*mother-board*), che contiene la CPU, la memoria primaria, il bus e gli alloggiamenti (*slot*) di espansione per il controllo delle periferiche

# La scheda madre di un PC



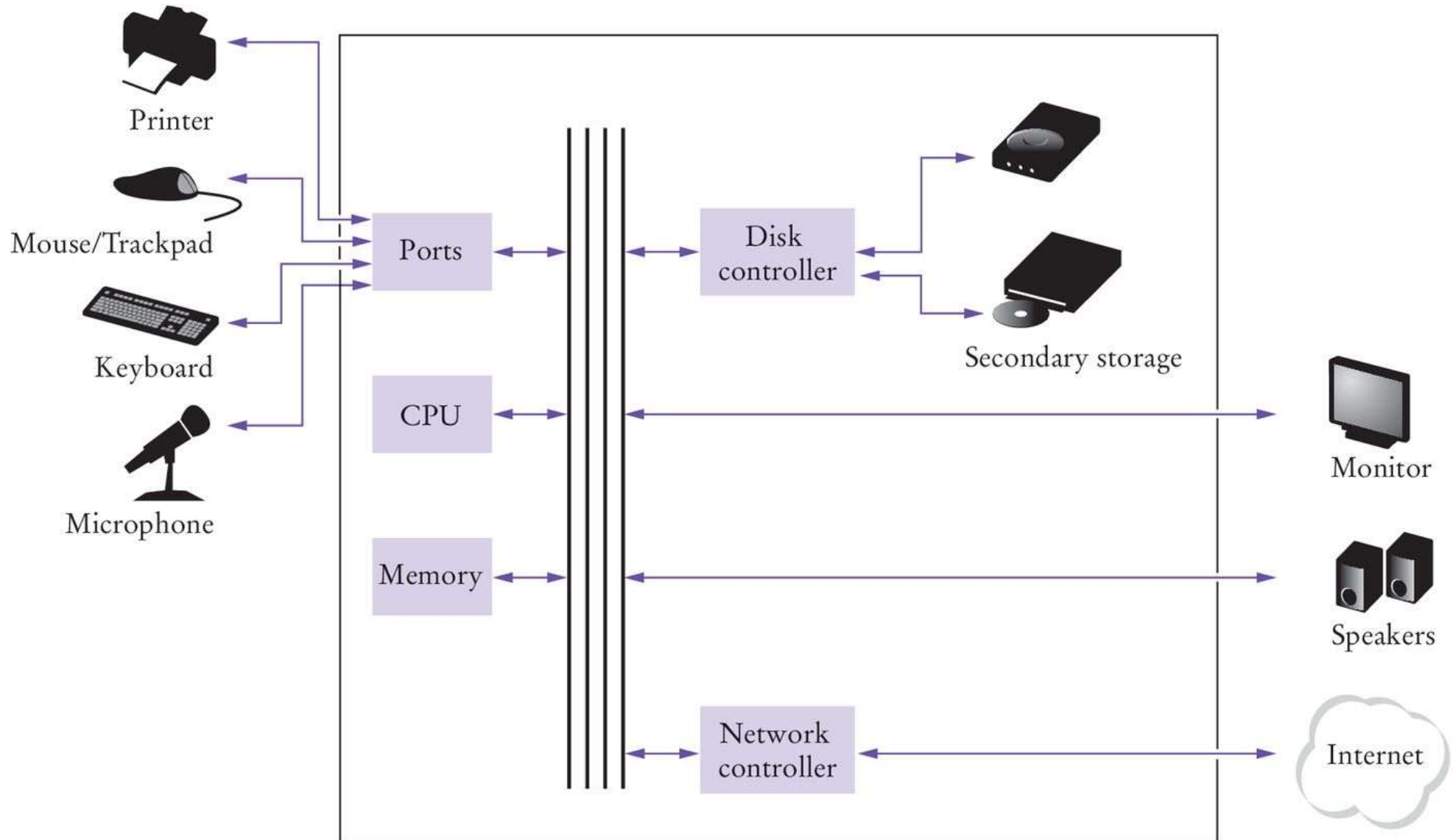
**CPU**

**slot**

**RAM**



# Schema degli elementi di un PC



**Cos'è un programma?**



# Cos'è un programma?

- ❑ Ogni programma svolge una diversa funzione, anche complessa

- Es. impaginare testi o giocare a scacchi

- ❑ Un computer è quindi una macchina che

- *memorizza dati* (numeri, parole, immagini, suoni...)
  - *interagisce con dispositivi* (schermo, tastiera, mouse...)
  - *esegue programmi*

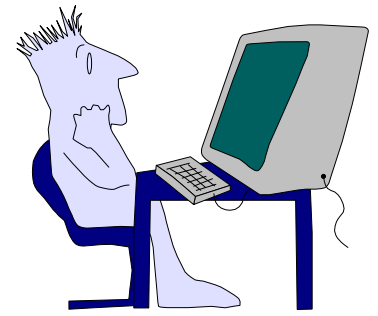
- ❑ I programmi sono *sequenze di istruzioni che il computer esegue e di decisioni che il computer prende* (sulla base dei dati) per svolgere una certa attività



# Quali istruzioni?

- Nonostante i programmi siano molto sofisticati e svolgano funzioni molto complesse, le istruzioni di cui sono composti sono *molto elementari*, ad esempio
  - Istruzioni imperative
    - leggere un numero da una posizione della memoria
    - sommare due numeri
    - accendere un punto rosso in una data posizione dello schermo (sul quale esistono, ad esempio,  $1024 \times 768 = 786432$  diverse posizioni...)
  - Istruzioni condizionali (che prendono *decisioni*)
    - **se** un dato è negativo, proseguire il programma con l'istruzione presente a un determinato indirizzo anziché con la successiva (si dice che l'esecuzione "fa un salto")

# Cos'è un computer?



- ❑ L'*elevatissimo numero* di tali istruzioni presenti in un programma e la loro esecuzione ad *altissima velocità* garantisce l'illusione di una interazione fluida che viene percepita dall'utente
- ❑ Il computer, in conclusione, è una macchina estremamente *versatile*, caratteristica che le è conferita dai molteplici programmi che vi possono essere eseguiti, ciascuno dei quali consente di svolgere una determinata attività

# Cos'è la programmazione?

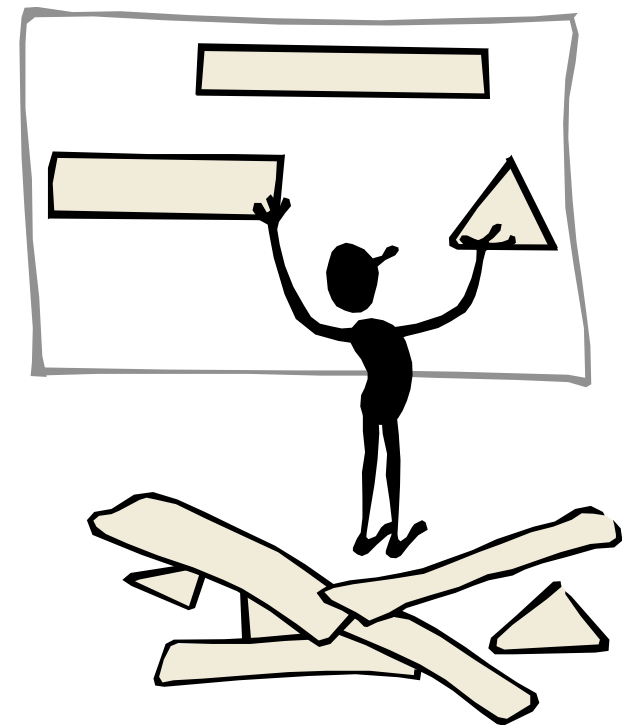


# Cos'è la programmazione?

Un programma descrive al computer, in estremo dettaglio, la sequenza di passi necessari per svolgere un particolare compito

L'attività di *progettare e realizzare un programma* è detta *programmazione*

**In questo corso imparerete a programmare un computer!**



# Cos'è la programmazione?

- ❑ *Usare* un computer *non* richiede **alcuna** attività di programmazione
  - così come per guidare un automobile non è necessario essere un ingegnere meccanico
- ❑ Al contrario, un *informatico professionista* solitamente svolge una intensa attività di programmazione, anche se la programmazione non è l'unica competenza che deve avere
- ❑ Chiunque lavori nel **settore dell'Ingegneria dell'Informazione** deve conoscere i fondamenti della programmazione, anche se poi si avvarrà della collaborazione di *programmatori*

# Cos'è la programmazione?

- ❑ La programmazione è una parte importante dell'informatica ed è un'attività che *in genere* affascina gli studenti e li motiva allo studio



*Cos'è un algoritmo?*



# Problemi e soluzioni

- ❑ Quale tipo di problemi è possibile risolvere con un computer?
  - Dato un insieme di fotografie di paesaggi, qual è il paesaggio *più rilassante*?
  - Avendo depositato ventimila euro in un conto bancario che produce il 5% di interessi all'anno, capitalizzati (cioè versati nel conto stesso) annualmente, quanti anni occorrono affinché il saldo del conto arrivi al doppio della cifra iniziale?
- ❑ Il primo problema NON può essere risolto dal computer. **Perché?**

# Problemi e soluzioni

- ❑ Il primo problema non può essere risolto dal computer perché non esiste una *definizione* di **paesaggio rilassante** che possa essere usata per confrontare *in modo univoco* due paesaggi diversi
- ❑ *Un computer può risolvere soltanto problemi che potrebbero essere risolti anche manualmente in modo univoco*
  - **è solo molto più veloce, non si annoia, non fa errori**
- ❑ Il secondo problema è certamente risolvibile manualmente, facendo un po' di calcoli...
- ❑ Esiste una definizione formale di *problema computazionale*, categoria a cui appartengono i problemi risolvibili da un calcolatore, ma non la vediamo
  - Complessa e poco utile nella pratica

# Cos'è un algoritmo?

- ❑ Si dice *algoritmo* la *descrizione* di un metodo di soluzione di un problema che
  - **sia eseguibile**
  - **sia priva di ambiguità**
  - **arrivi a conclusione in un tempo finito**
- ❑ *Un computer può risolvere soltanto quei problemi per i quali sia noto un algoritmo*
  - Sono un sottoinsieme dei "problemi computazionali", definiti con maggiore precisione dall'informatica teorica
- ❑ Nota: esistono molte diverse definizioni di “algoritmo”, ma tutte si equivalgono nella sostanza

# Come si descrive un algoritmo?

- ❑ Si dice *algoritmo* la *descrizione* di un metodo di soluzione di un problema che sia eseguibile, che sia non ambigua e che arrivi a conclusione in un tempo finito
- ❑ Come si descrive un algoritmo?
  - In un linguaggio naturale (a volte si dice "a parole"...)
  - In linguaggio matematico
  - **In un linguaggio di programmazione (vedremo!)**
  - In un linguaggio specifico del contesto applicativo
    - Es. una costruzione grafica
  - In “pseudocodice”, un linguaggio artificiale simile a molti linguaggi di programmazione
  - In una forma mista di tutte le precedenti

# Un esempio di algoritmo

- ❑ **Problema:** Avendo depositato ventimila euro in un conto bancario che produce il 5% di interessi all'anno, capitalizzati annualmente, quanti anni occorrono affinché il saldo del conto arrivi al doppio della cifra iniziale?
  - È un problema di "simulazione", che si risolve **accelerando il trascorrere del tempo...** quindi **dobbiamo tenere traccia esplicitamente del tempo** che passa (o, meglio, che "facciamo passare"), perché non possiamo guardare "l'orologio"
- ❑ **Algoritmo:**
  - 1 Situazione iniziale:  
il numero di **anni trascorsi** è 0 e il **saldo** è 20000€
  - 2 **Ripetere** i successivi passi 3 e 4 **finché** il **saldo** è minore di 40000€, **poi** passare al punto 5
    - 3 Aggiungere 1 al valore degli **anni trascorsi**
    - 4 Il **saldo** diventa uguale al valore **attuale** del **saldo** moltiplicato per 1.05 (cioè aggiungiamo il 5%)
  - 5 Il risultato è il valore attuale degli **anni trascorsi**

# Un esempio di algoritmo

## □ Il metodo di soluzione proposto

- **è non ambiguo**, perché fornisce precise istruzioni su cosa bisogna fare ad ogni passaggio e su quale deve essere il passaggio successivo
- **è eseguibile**, perché ciascun passaggio può essere eseguito concretamente (ad esempio, scrivendo su un foglio di carta...)
  - se, invece, il metodo di soluzione dicesse che il tasso di interesse da usare al punto 4 è variabile in dipendenza da fattori economici futuri, il metodo non sarebbe eseguibile!
- **arriva a conclusione in un tempo finito**, perché ad ogni passo il saldo aumenta di almeno mille euro, quindi al massimo in 20 passi arriva al termine

# A cosa servono gli algoritmi?

- ❑ Dopo aver individuato un algoritmo, per renderlo eseguibile mediante un calcolatore bisogna
  - scriverlo in un **programma**, cioè tradurlo in un **linguaggio di programmazione** che sia comprensibile al calcolatore
- ❑ **Prima di scrivere un programma, è necessario individuare un algoritmo!**
  - Iniziare a scrivere un programma senza aver individuato un algoritmo è come **partire senza sapere che direzione prendere...**  
**sarà molto probabilmente una perdita di tempo!**
  - Rileggere questa slide tra 2 settimane, poi ogni settimana 😊

**Lezione 04**  
**08/10/2024**  
**ore 10.30-12.30**  
**aula Ve**



# Linguaggi di programmazione

- ❑ Come abbiamo visto, ci serve **un linguaggio di programmazione** per descrivere ai computer la metodologia di soluzione di un problema, cioè **per scrivere un algoritmo** in un linguaggio comprensibile al calcolatore
  - Ovviamente il linguaggio di programmazione deve essere comprensibile anche al programmatore!
  - Acquisire questa competenza è l'obiettivo fondamentale di questo corso
- ❑ Negli anni, sono stati progettati (o "inventati") molti diversi linguaggi di programmazione, **tutti sostanzialmente equivalenti tra loro**
  - Alcuni hanno avuto (molto) più successo di altri
  - Il successo di un linguaggio di programmazione è dovuto a sue caratteristiche oggettive e (forse soprattutto) a questioni di "marketing" e a fenomeni sociologici
  - Tra i linguaggi oggi più diffusi, troviamo **Python**, Java e C++
    - Sono significativamente simili tra loro, soprattutto agli occhi di un programmatore professionista
  - Storicamente hanno avuto molta importanza FORTRAN e C
    - C è ancora piuttosto usato
    - Spesso c'è bisogno di scrivere codice anche in linguaggi ormai obsoleti per attività di manutenzione di programmi esistenti (es. COBOL, soprattutto per programmi bancari/finanziari)

# **Il linguaggio di programmazione Python**

# Il linguaggio di programmazione Python

- ❑ Nei primi Anni Novanta, Guido van Rossum progettò quello che sarebbe poi diventato il linguaggio Python
- ❑ Non era soddisfatto dei linguaggi disponibili
  - Erano stati progettati per scrivere programmi di grandi dimensioni che venissero **eseguiti** velocemente
- ❑ Aveva bisogno di un linguaggio che consentisse, invece, di **creare** programmi rapidamente
  - Python fu, quindi, progettato per avere una **sintassi** (molto?) **più semplice e chiara** di quella dei linguaggi di programmazione più diffusi, come Java, C e C++
  - In questo modo l'apprendimento del linguaggio è più rapido



# Installare Python

- ❑ Per seguire con profitto il corso è necessario installare l'ambiente di programmazione di Python sul proprio computer
  - **Nel sito del corso istruzioni per farlo**
  - **Se non l'avete ancora fatto, fatelo il più presto possibile**
  - Usate il programma Game of Life per verificare la corretta installazione, scaricandolo dal sito del corso e provando a eseguirlo
- ❑ In attesa di andare in laboratorio, provate a copiare ed eseguire i semplici esempi che vedremo nelle prossime lezioni
- ❑ In laboratorio (ovviamente) Python è già installato

# Scrittura di un programma Python

- ❑ Per scrivere un programma ci sono fondamentalmente due alternative
  - Usare un ambiente di sviluppo integrato (IDE, *integrated development environment*)
    - Es. `idle` (specifico per Python),  
`eclipse` (configurabile per molti linguaggi) ecc.
  - Usare un editor di testo (*text editor*)
    - Es. `notepad` o `notepad++` in Windows,  
`nedit` o `gedit` (o altri...) in Linux
    - Attenzione: **NON un impaginatore di testi** (*word processor*),  
come Microsoft Word
  - Consiglio di provare entrambe le strategie anche se probabilmente la prima, con `idle`, è più comoda
    - **Trovate un'apposita pagina nel sito del corso:**  
**"Come usare l'ambiente di programmazione Python"**  
**nella sezione "Esercitazioni"**

Dovete **PROVARE** sul PC  
mentre leggete quella pagina

# Il nostro primo programma Python

- ❑ Nella realizzazione del nostro primo programma scritto in Python NON partiremo dal problema, per passare all'algoritmo e, poi, alla scrittura del codice (come si dovrebbe fare...)
  - In parte perché il primo "problema" è veramente banale...
- ❑ Piuttosto, **ANALIZZEREMO** un primo programma scritto in Python, immaginando che l'abbia scritto qualcun altro
  - **Dedurremo** le sue funzionalità (cioè il "problema" che risolve) eseguendolo e osservando le informazioni visualizzate
  - Scopriremo che è **un programma che visualizza un messaggio di saluto, Hello, World!**
    - Iniziare in questo modo è una tradizione nella didattica dei linguaggi di programmazione, in omaggio a quanto fatto da Brian Kernighan e Dennis Ritchie nelle pagine iniziali del libro in cui presentavano il Linguaggio C

# Il nostro primo programma Python

```
# il mio primo programma scritto in Python  
print("Hello, World!")
```



## ❑ Occorre fare attenzione

- il testo va inserito esattamente come è presentato qui
  - per il momento... poi vedremo
  - anche gli spazi! non aggiungere/togliere spazi
- *maiuscole e minuscole sono considerate distinte*
- il file, creato con un editor di testo o un IDE, **deve** avere un nome che termina con **.py**
  - la parte iniziale del nome è a scelta libera
    - è bene che faccia riferimento al compito svolto dal programma
  - in questo caso, ad esempio, **hello.py** oppure **displayhello.py**

*[il docente presenta alcuni esempi  
interattivi di utilizzo dell'ambiente  
di sviluppo di Python, usando idle  
oppure editor+shell]*



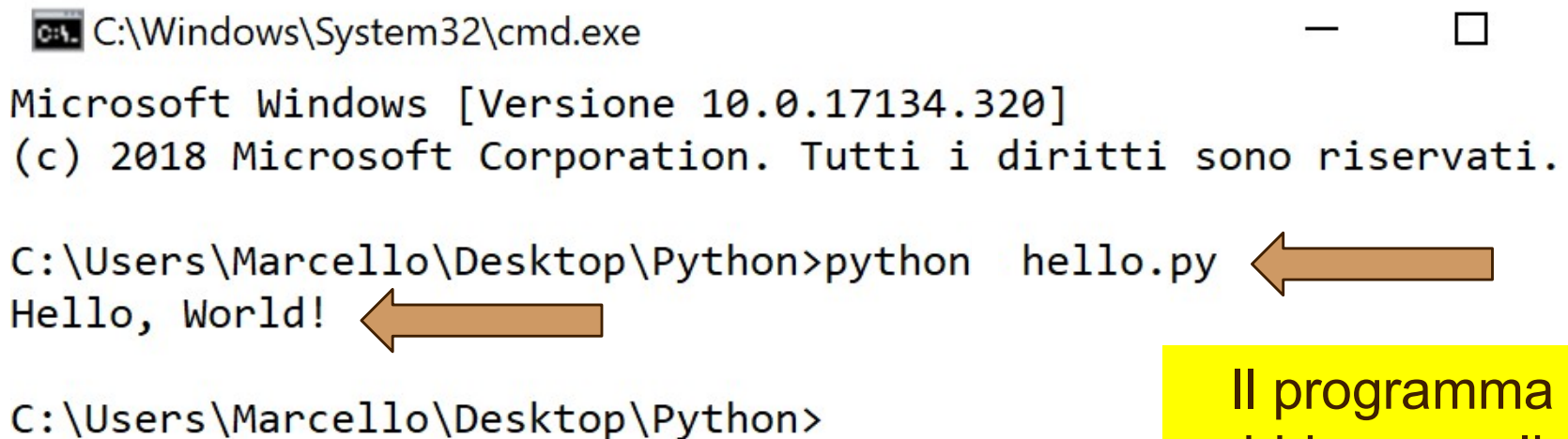
In alcune versioni il programma si chiama **python3**

Nel sito del corso: "Come usare l'ambiente di programmazione Python"

## Il nostro primo programma Python

❑ A questo punto, in una "finestra di comandi" (*shell*) ***eseguiamo*** il programma **`python hello.py`**

ottenendo la visualizzazione del messaggio di saluto sullo schermo, **all'interno della stessa finestra**



The screenshot shows a Windows Command Prompt window titled "C:\Windows\System32\cmd.exe". The text inside the window reads: "Microsoft Windows [Versione 10.0.17134.320] (c) 2018 Microsoft Corporation. Tutti i diritti sono riservati." followed by the command prompt "C:\Users\Marcello\Desktop\Python>". The command "python hello.py" has been entered and executed, resulting in the output "Hello, World!". Two brown arrows point to the command and the output. A third brown arrow points to the command prompt line "C:\Users\Marcello\Desktop\Python>".

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versione 10.0.17134.320]
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.

C:\Users\Marcello\Desktop\Python>python hello.py
Hello, World!

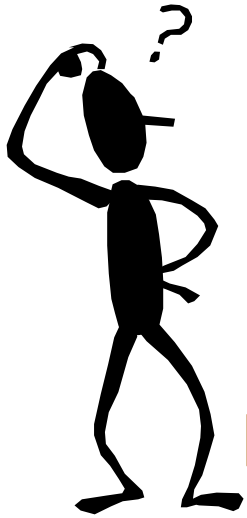
C:\Users\Marcello\Desktop\Python>
```

Quando la "shell" attende un comando, mostra il nome della cartella in cui ci si trova: si chiama *prompt* ("sollecito") dei comandi

Il programma che abbiamo realizzato NON è dotato di un'interfaccia grafica

# Analisi del primo programma

```
# il mio primo programma scritto in Python
print("Hello, World!")
```



- Un programma Python è costituito da *istruzioni* o *enunciati*, che verranno tradotti in linguaggio macchina dall'interprete Python (il programma **python** o **python3**), per poi essere eseguiti dalla CPU (dallo stesso interprete)
  - Immaginateli come *comandi* dati al computer
- (Per il momento) **ogni riga è un'istruzione**

Vedremo  
meglio i  
commenti  
più avanti

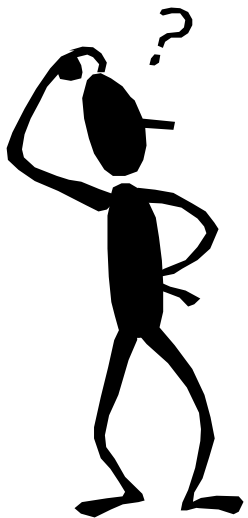
Tranne le righe che iniziano con il carattere **#**, che sono **commenti**, cioè **informazioni per un lettore umano**, **ignore** dall'interprete Python

Non essendo destinate all'interprete, possono essere in italiano (mentre le istruzioni Python sono in inglese)

# Analisi del primo programma

```
# il mio primo programma scritto in Python  
print("Hello, World!")
```

- ❑ La seconda riga contiene un "vero" enunciato, l'unico di questo nostro primo programma
- ❑ Ha il compito di visualizzare (o *stampare*, in inglese *print*) una riga di testo, in questo caso contenente **Hello, World!** (senza le virgolette, quelle servono a "delimitare" il testo da stampare)




- Dove la "stampa" ? Sullo schermo! Un po' strano, ma è così
- È un tipo di enunciato che si chiama **invocazione di funzione** (o chiamata di funzione), perché **print** è una *funzione* di Python a cui trasferiamo (o "passiamo") informazioni perché le elabori in qualche modo
  - L'elaborazione svolta dalla funzione **print** è la visualizzazione delle informazioni ricevute

**Lezione 05**  
**09/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Funzioni in Python

```
print("Hello, World!")
```

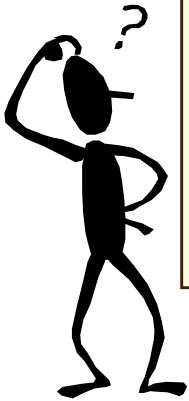
- ❑ Una **funzione**, come `print`, è un insieme di istruzioni che servono a portare a termine un compito specifico (di solito tale compito è descritto dal "nome" della funzione, come `print`)
- ❑ In questo caso la funzione è stata progettata da altri
  - È una **funzione predefinita** (*built-in function*) del linguaggio Python, il quale mette a disposizione una collezione di funzioni utili che impareremo a usarealtre volte la progetteremo noi: impareremo a progettare funzioni 
- ❑ Di una funzione già progettata interessa sapere soltanto
  - Il nome (per invocarla)
  - Quali informazioni si aspetta di ricevere (a volte niente)
    - Sono i cosiddetti **argomenti** o **parametri** della funzione, scritti **all'interno delle parentesi tonde** che seguono il nome della funzione
  - **Cosa fa**
- ❑ Non dobbiamo preoccuparci di **"come"** lo fa, cioè di come sia stato realizzato il suo progetto: è il classico e fondamentale **modello ingegneristico "a scatola nera"** (*black box*)

In Python (senza il commento,  
che non è necessario)

`print("Hello, World!")`

## Il nostro primo programma Java

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```



- ❑ Questo programma, scritto in linguaggio Java, svolge esattamente la stessa elaborazione del nostro
- ❑ **La semplicità di Python è evidente...**
- ❑ Diversamente dai programmi semplici, programmi che svolgono **elaborazioni complesse** avranno **complessità paragonabile** nei due linguaggi: **il vantaggio di Python si ha soprattutto nei programmi semplici**

# Stringhe in Python

```
# il mio primo programma scritto in Python  
print("Hello, World!")
```

- ❑ In informatica, una sequenza di caratteri racchiusa tra virgolette, come **"Hello, World!"**, viene chiamata **stringa** (*string*)
  - Per il momento useremo stringhe soltanto come argomenti di funzioni, ma presto ne vedremo molti altri utilizzi
- ❑ In Python, una stringa può, in alternativa, essere delimitata da "virgolette singole" o apici/apostrofo: **'Hello, World!'**
  - È indifferente, ma è **meglio essere coerenti all'interno di un singolo programma** per evitare dubbi da parte di un lettore umano
- ❑ Nota: seguendo le lezioni, sorgono spontanee molte domande relative alla **sintassi** del linguaggio
  - La cosa migliore è **PROVARE per vedere cosa succede**
  - Es. cosa succede se inizio una stringa con apice singolo e la termino con virgolette doppie? Facciamoci del male... 😊

# Stampare molte righe

```
print("Hello, World!")  
print('Ciao, Mondo!')  
print()  
print("Di nuovo ciao")
```

```
Hello, World!  
Ciao, Mondo!  
  
Di nuovo ciao
```

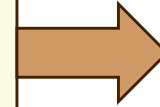
- ❑ Un programma può essere costituito da **più enunciati**, che vengono **eseguiti in sequenza**, dall'alto in basso, seguendo l'ordine con cui sono stati scritti dal programmatore nel file
  - Il file in cui scriviamo il programma è detto **file sorgente**, in inglese *source file*, perché è la *sorgente di informazioni* per il processo di esecuzione del programma
- ❑ Per il momento conosciamo un unico enunciato, l'invocazione della funzione **print**
  - Ogni invocazione di **print** stampa una riga
  - L'invocazione **print()**, senza stringa da stampare, stampa una riga vuota [non è un errore! può servire per impaginare...]

Brutto...



# Spazi nelle stringhe

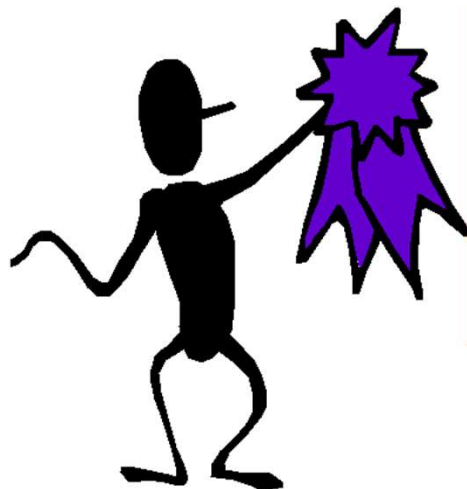
```
print("Hello, World!")  
print(" Ciao, Mondo!")  
print("Ciao,  Mondo!")
```



```
Hello, World!  
  Ciao, Mondo!  
Ciao,  Mondo!
```

- ❑ **Gli spazi contenuti all'interno delle stringhe sono significativi**, fanno parte del messaggio visualizzato e vengono riportati fedelmente sullo schermo dalla funzione **print**
- ❑ L'impaginazione delle informazioni visualizzate da un programma è un aspetto **molto** rilevante
  - Si pensi a una tabella di numeri... gli spazi possono essere decisivi per visualizzare una "bella" tabella, dove le informazioni siano chiare

# Impaginazione dei dati visualizzati



|   |    |    |    |    |    |
|---|----|----|----|----|----|
| 2 | 4  | 6  | 8  | 10 | 12 |
| 3 | 6  | 9  | 12 | 15 | 18 |
| 4 | 8  | 12 | 16 | 20 | 24 |
| 5 | 10 | 15 | 20 | 25 | 30 |

|   |   |   |    |    |    |    |    |    |    |
|---|---|---|----|----|----|----|----|----|----|
| 2 | 4 |   | 6  |    | 8  | 10 | 12 |    |    |
| 3 |   | 6 |    | 9  | 12 | 15 |    | 18 |    |
| 4 |   | 8 |    | 12 | 16 | 20 | 24 |    |    |
| 5 |   |   | 10 | 15 | 20 |    |    | 25 | 30 |



# Enunciati incolonnati!!

```
print("Hello, World!")  
print("Ciao, Mondo!")  
    print()  
print("Di nuovo ciao")
```

- ❑ In un programma Python, tutti gli enunciati **devono** essere "incolonnati a sinistra", cioè **devono tutti iniziare nella prima colonna**, all'inizio della riga del testo nel file sorgente
- ❑ Se ci sbagliamo, cosa succede? **IndentationError** !

```
C:\Windows\System32\cmd.exe  
Microsoft Windows [Versione 10.0.17134.320]  
(c) 2018 Microsoft Corporation. Tutti i diritti sono riservati.
```

```
C:\Users\Marcello\Desktop\Python>python hello.py
```

```
File "hello.py", line 3  
    print()  
    ^  
IndentationError: unexpected indent
```

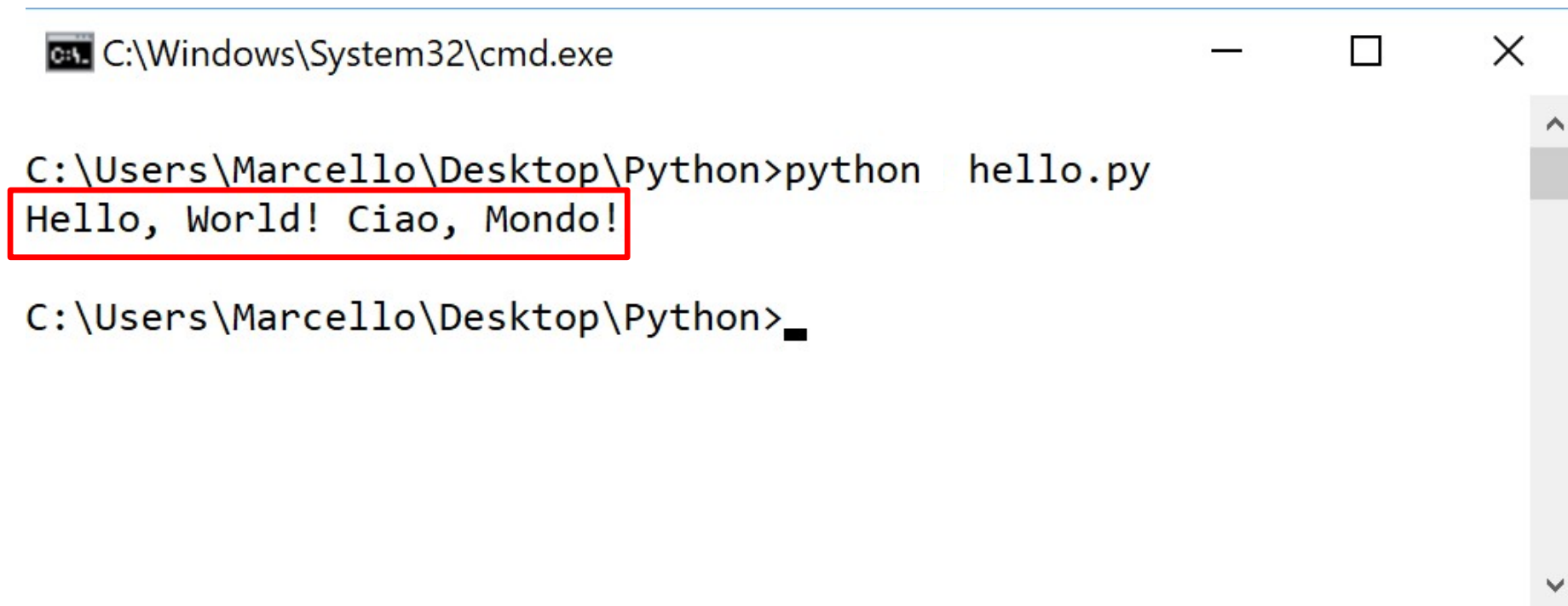
```
C:\Users\Marcello\Desktop\Python>
```

Leggiamo sempre con attenzione le segnalazioni fornite dall'interprete Python: sono un valido aiuto per individuare gli errori

# Stampare più stringhe in una riga

```
print("Hello, World!", "Ciao, Mondo!")
```

- Se scriviamo **più stringhe** come argomento di **print**, una dopo l'altra, **separate da una virgola** (posta FUORI dalle virgolette), viene stampata **un'unica riga** che contiene le stringhe, nello stesso ordine, **separate tra loro da uno spazio** aggiunto automaticamente dalla funzione **print** durante l'esecuzione
  - bisogna **tenerne conto** per ottenere l'impaginazione desiderata



The screenshot shows a Windows command prompt window titled "C:\Windows\System32\cmd.exe". The command prompt shows the directory "C:\Users\Marcello\Desktop\Python" and the command "python hello.py" being executed. The output "Hello, World! Ciao, Mondo!" is displayed on the next line and is highlighted with a red rectangular box. The prompt then shows "C:\Users\Marcello\Desktop\Python>" with a cursor.

```
C:\Windows\System32\cmd.exe
C:\Users\Marcello\Desktop\Python>python hello.py
Hello, World! Ciao, Mondo!
C:\Users\Marcello\Desktop\Python>
```

# Stampare più stringhe in una riga

```
print("Hello, World!", "Ciao, Mondo!")
```

```
print("Hello, World! Ciao, Mondo!")
```

spazio

- ❑ Questi due programmi sono **equivalenti**: risolvono lo stesso problema (visualizzano lo stesso messaggio)
  - Scrivere un'unica stringa al posto di più stringhe è sempre possibile, quindi il fatto che **print** possa ricevere più argomenti (separati da virgole) **sembra** una cosa inutile
  - In realtà è utile, soprattutto quando (tra poco) impareremo che gli argomenti di **print** possono essere informazioni diverse dalle stringhe!

# Stampare risultati di operazioni!

- ❑ La funzione **print** non è limitata alla visualizzazione di stringhe, può anche visualizzare **il risultato di operazioni aritmetiche!**

```
print("Il risultato di 4 + 5 è", 4 + 5)
```



- Gli spazi prima e dopo il + sono necessari? Provate...

- ❑ Ovviamente avrei potuto scrivere un unico argomento

```
print("Il risultato di 4 + 5 è 9")
```

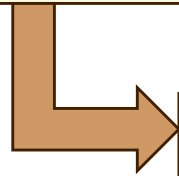
ma nel primo programma non devo fare i calcoli! 😊

- ❑ **Gli argomenti di print possono essere espressioni aritmetiche, delle quali viene visualizzato il risultato**
- ❑ Dobbiamo imparare a scrivere espressioni aritmetiche!

# Operazione aritmetica o stringa?

- ❑ Attenzione, cosa stampa questo programma?

```
print("Il risultato di 4 + 5 è", "4 + 5")
```



```
Il risultato di 4 + 5 è 4 + 5
```

- ❑ **Il contenuto di una stringa** (cioè i caratteri racchiusi tra virgolette, semplici o doppie) **non viene analizzato** dall'interprete Python, viene stampato e basta!
  - Il fatto che all'interno di una stringa siano presenti caratteri che compongono un'espressione aritmetica, non fa differenza: è una stringa, non viene interpretata
- ❑ **Se voglio visualizzare il risultato di un'operazione, la devo mettere FUORI dalle virgolette**

# Espressioni aritmetiche



# Espressioni aritmetiche in Python

- ❑ Le espressioni aritmetiche possono contenere numeri interi e numeri reali
  - Nella scrittura dei numeri, come separatore decimale si usa **il punto**, secondo la convenzione anglosassone, e **non la virgola**
    - Non si usano i separatori delle migliaia
- ❑ Gli operatori aritmetici sono
  - + per l'addizione
  - - per la sottrazione
  - \* per la moltiplicazione [ATTENZIONE]
  - / per la divisione
  - \*\* per l'elevamento a potenza ( $2^{**}3$  è  $2^3$ )

# Regole di precedenza

- ❑ Le regole di precedenza tra gli operatori aritmetici sono uguali a quelle dell'algebra ordinaria [ripassare...]
- ❑ Si possono alterare introducendo parentesi
  - Nelle espressioni si usano **SOLO le parentesi tonde**, anche se sono presenti più livelli di parentesi annidate

```
print( (2.3 + 5* (3+4) ) / 7.22 )
```

- ❑ Attenzione: diversamente dall'algebra ordinaria, non è possibile lasciare sottinteso il simbolo di moltiplicazione davanti a una parentesi
  - Cosa succede se lo dimentico? **Sperimentare...**

**Lezione 06**  
**11/10/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Elevamento a potenza... e radice!

- ❑ Usando l'operatore di elevamento a potenza possiamo ovviamente anche calcolare, ad esempio, la radice quadrata o cubica di un numero
- ❑ Ricordiamo che estrarre la radice quadrata di un numero equivale a elevarlo all'esponente  $\frac{1}{2} = 0.5$

```
print(16**0.5)  
print(27**(1/3))  
print(28**(1/3))
```

```
4.0  
3.0  
3.0365889718756622
```

- ❑ Osserviamo che il risultato generato dell'operatore `**` viene sempre scritto come "numero con la virgola" (in realtà, con il punto) anche quando è un numero intero



- Vedremo meglio più avanti...

- Per gli altri operatori aritmetici non è così

```
print(2*3)  
print(2*3.1)
```

```
6  
6.2
```

# Divisione intera e resto

- ❑ Il risultato di  $7 / 4$  è (ovviamente) 1.75
- ❑ A volte, quando il dividendo e il divisore sono valori numerici **interi**, può interessare il **quoziente intero** e/o il **resto**
- ❑ Nella "divisione intera"  $7 \div 4$ , il quoziente è 1 e il resto è 3; in Python si ottengono questi due valori usando due operatori aritmetici specifici:

```
print(7 // 4)  
print(7 % 4)
```



```
1  
3
```

## Attenzione:

(purtroppo) il simbolo % in Python non ha NIENTE a che fare con la percentuale...

- ❑ Vedrete che può essere molto utile negli esercizi!!
  - Es. il resto della divisione per 10 è l'ultima cifra di un numero
  - Es. il resto della divisione per 2 è 0 se e solo se il numero è pari

# Variabili e acquisizione di dati

# Programmi poco utili...

- ❑ Tutti i (semplicissimi) programmi Python visti finora eseguono **sempre la stessa elaborazione** ogni volta che vengono eseguiti
  - Sembrano veramente poco utili...
- ❑ Per ottenere un risultato diverso, **l'utilizzatore del programma dovrebbe modificare il file sorgente!**
  - Normalmente l'esperienza degli UTILIZZATORI di programmi è ben diversa: **nessun utilizzatore modifica il codice sorgente di un programma** (di solito non sa nemmeno che esiste il codice sorgente...)
- ❑ Dobbiamo imparare a scrivere programmi che **acquisiscono dati dall'utente e si comportano di conseguenza** (cioè svolgono un'elaborazione che dipende dai dati acquisiti)

# Un saluto personalizzato

- ❑ Vogliamo progettare un programma che **chiede all'utente** di scrivere il proprio nome, per poi visualizzare un messaggio di saluto personalizzato
- ❑ Algoritmo (in linguaggio naturale)
  - **Chiedi all'utente** di scrivere il suo nome e **archivia tale nome "da qualche parte" nella memoria**
  - Stampa un messaggio di saluto costituito da una prima parte prefissata ("Ciao, "), seguita dal nome dell'utente (in generale, variabile ad ogni esecuzione), **recuperandolo dalla memoria**
- ❑ Esempio di esecuzione

```
Come ti chiami? Marcello  
Ciao, Marcello
```

Qui in **rosso** i caratteri digitati dall'utente, in nero quelli scritti dal programma. In realtà il colore sarà identico (e dipende dalla configurazione della finestra)



# Un saluto personalizzato



```
name = input("Come ti chiami? ")  
print("Ciao, ", name)
```

Uno spazio è utile

Spazio  
aggiunto  
da print

❑ Per acquisire dati dall'utente utilizziamo un'altra funzione predefinita di Python: **input**

- Per prima cosa, visualizza la stringa fornita come argomento, esattamente come **print** (ma accetta **un solo argomento**, che può essere assente)
- Poi, **sospende** l'esecuzione del programma, in attesa che l'utente scriva una stringa, terminandola con il tasto Invio/Enter
  - Tale stringa può anche essere composta da più "parole", numeri, simboli: **qualsiasi cosa fino al tasto Invio/Enter**

```
Come ti chiami? C-3PO R2-D2  
Ciao, C-3PO R2-D2
```

- La stringa acquisita tramite la tastiera deve essere memorizzata **"da qualche parte"** nella RAM, per essere utilizzata dai successivi enunciati del programma (in questo caso, da **print**): memorizzata **DOVE ?**

# Il concetto di "variabile"

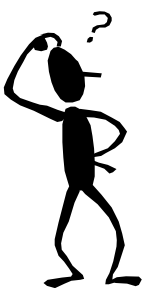
```
name = input("Come ti chiami? ")  
print("Ciao, ", name)
```

- ❑ Nella programmazione, spesso (praticamente sempre...) avremo bisogno di **"zone di memoria" in cui archiviare informazioni temporaneamente**, durante l'esecuzione di un programma
- ❑ Le singole celle di memoria sono individuate univocamente da **indirizzi**: questo sistema è scomodo per chi programma (ma efficiente per la CPU...)
  - Le persone umane preferiscono associare le informazioni a **NOMI** piuttosto che a numeri... pensate alla rubrica telefonica!
  - Quando, in un programma, dobbiamo memorizzare un'informazione temporanea (ad esempio, una stringa o un numero), **ci inventiamo un NOME per la zona di memoria** che verrà occupata da tale informazione, senza preoccuparci del suo indirizzo e della sua dimensione (che verranno gestiti automaticamente dall'interprete, in collaborazione con il sistema operativo)
  - **Scegliamo un nome che abbia per noi un significato!**

# Il concetto di "variabile"

```
name = input("Come ti chiami? ")  
print("Ciao, ", name)
```

- ❑ Una zona di memoria utilizzata durante l'esecuzione di un programma per conservare informazioni temporanee viene chiamata **"variabile"** (perché, come vedremo, il suo contenuto può anche cambiare nel tempo) e viene identificata da un **nome** scelto dal programmatore (**solitamente in inglese**)
  - Un **nome di variabile** può essere composto da lettere (maiuscole e/o minuscole), cifre numeriche e carattere di sottolineatura (*underscore*, `_`), ma deve iniziare con una lettera (es. `name01`, `nameAndSurname`, `LOGICAL_AND_PHYSICAL`)
- ❑ Lo stesso nome verrà utilizzato per **"scrivere"** informazioni nella variabile e per **"leggere"** tali informazioni in seguito
  - In informatica, la **"scrittura"** di informazioni è sempre **"distruttiva"**, cioè l'informazione che viene effettivamente memorizzata NON dipende da ciò che la zona di memoria eventualmente conteneva in precedenza, come se, prima di scrivere, si facesse "pulizia", mentre la **"lettura"** è sempre **"non distruttiva"**, cioè non modifica in alcun modo l'informazione contenuta nella memoria letta (che può, quindi, essere letta più volte)



# Scrivere in una variabile

```
name = input("Come ti chiami? ")  
print("Ciao,", name)
```

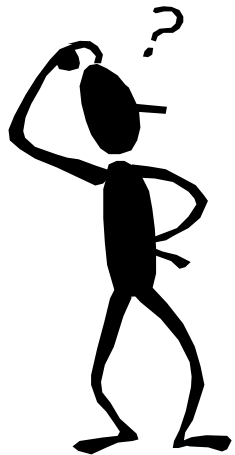
□ Come facciamo a "scrivere" un valore in una variabile?

■ Usiamo l'**istruzione (o enunciato) di assegnazione**

• Sintassi: nome della variabile, **seguito dal segno di uguale** e dal "valore che vogliamo scrivere nella variabile"



– Se il nome non era mai stato utilizzato prima nel programma, stiamo **utilizzando/definendo una nuova variabile**: l'interprete lo sa, perché durante l'esecuzione di un programma **conserva un elenco di tutte le variabili utilizzate fino a quel momento, insieme all'indirizzo in memoria assegnato a ciascuna**, quindi inserisce tale nome nel proprio elenco di variabili note, assegnando uno spazio in memoria a tale nuova variabile



– Altrimenti, stiamo **modificando il valore memorizzato in una variabile definita in precedenza**: l'interprete recupera nel proprio elenco l'indirizzo della zona di memoria corretta

– La sintassi dell'istruzione è identica, nei due casi

# Scrivere in una variabile

```
name = input("Come ti chiami? ")  
print("Ciao,", name)
```

- ❑ In un'istruzione di assegnazione, **a destra** del segno di uguale ci deve essere il **valore** che vogliamo assegnare alla variabile nominata a sinistra
  - In alcuni linguaggi di programmazione, come simbolo per l'assegnazione si usa una freccia orientata da destra a sinistra (o qualcosa di simile), che rende meglio l'idea

```
name ← qualcosa...
```

```
name <- qualcosa...
```

- ❑ **Nel nostro esempio, tale valore è quello "restituito" dall'invocazione della funzione `input`**



- Quando si invoca una funzione, questa può semplicemente "fare qualcosa" (come **print**, che visualizza quanto richiesto) oppure può anche (o soltanto) **restituire un valore** (un numero, una stringa, ecc.)



- Nell'istruzione che contiene l'invocazione della funzione, il valore restituito dalla funzione "prende il posto" dell'invocazione, dopo che questa è stata eseguita

# Scrivere in una variabile

```
name = input("Come ti chiami? ")  
print("Ciao,", name)
```

- ❑ Di solito le istruzioni vengono eseguite "da sinistra a destra"... ad esempio, nell'invocazione della funzione **print** con due argomenti, viene prima visualizzato il primo argomento, poi il secondo...
- ❑ L'esecuzione di un enunciato di assegnazione inizia sempre dalla parte che si trova **a destra del segno di uguale**, per calcolare il valore che verrà poi memorizzato nella variabile che si trova a sinistra del segno di uguale!
- ❑ A volte assegneremo direttamente un valore a una variabile (non è necessario acquisirlo da tastiera): 

```
name = "Marcello"
```


 ovviamente è un utilizzo diverso, manca l'interazione con l'utente!



```
name = input("Come ti chiami? ")  
startOfMessage = "Ciao,"  
print(startOfMessage, name)
```

# Leggere una variabile

```
name = input("Come ti chiami? ")  
print("Ciao, ", name)
```

- ❑ Dopo aver assegnato un valore a una variabile, solitamente il programma avrà bisogno di utilizzarlo, altrimenti non serve a niente 😊
- ❑ "Utilizzare" il valore di una variabile significa "leggere" l'informazione che vi è stata memorizzata e inserire tale valore all'interno di un'istruzione (che può essere un'invocazione della funzione **print**, ma si tratta soltanto di un esempio)
- ❑  Più precisamente, nel caso in cui la variabile sia stata modificata più volte, si legge il valore che vi è stato memorizzato più recentemente (perché, come detto, ogni scrittura cancella qualsiasi traccia delle informazioni precedenti)
- ❑ Per fare questo, è sufficiente **scrivere il nome della variabile** nella stessa posizione in cui vorremmo scrivere il valore in essa memorizzato! Tutte le volte che vogliamo...

```
name = input("Chi sei? ")  
print("Ciao, ", name)  
print(name, "sei super!")
```



```
Chi sei? zia  
Ciao, zia  
zia sei super!
```



**Lezione 07**  
**15/10/2024**  
**ore 10.30-12.30**  
**aula Ve**



Le espressioni aritmetiche possono contenere variabili, oltre ai "numeri" (che si chiamano tecnicamente "valori numerici letterali")

# Chiedere un numero all'utente

```
age = input("Quanti anni hai? ")  
print("Fra due anni avrai", age + 2, "anni")
```

Sempre attenzione agli spazi...

Quanti anni hai? 21  
Fra due anni avrai 23 anni

❑ **Vorremmo** che succedesse questo

❑ Invece, se proviamo a eseguire il programma, l'interprete Python scrive un messaggio d'errore



- Sul quale per il momento non investighiamo...
- Il problema è che la funzione **input** restituisce **SEMPRE** una stringa, cioè una sequenza di caratteri, e **fare operazioni aritmetiche** (come l'addizione **age + 2** nella seconda riga) **con una sequenza di caratteri** è (ovviamente?) un'azione priva di senso



❑ **Intuizione:** ci sarà una diversa funzione che acquisisce dati numerici dalla tastiera! Qualcosa come **inputNumber(...)**

- Bella idea, ma non è giusta ☺ se fosse così, probabilmente la funzione **input** si sarebbe chiamata **inputString...**

Usando questo nome di variabile, mi ricordo meglio che contiene una stringa...

# Chiedere un numero all'utente

```
ageString = input("Quanti anni hai? ")  
age = int(ageString)  
print("Fra due anni avrai", age + 2, "anni")
```

- ❑ **Ora funziona!** Abbiamo **"trasformato"** la stringa

(contenuta nella variabile **ageString**) in un numero intero (poi memorizzato nella variabile **age**) invocando la funzione **int**

- ❑ La funzione **int**, predefinita in Python, vuole come argomento una stringa (**i cui caratteri descrivano un numero intero**) e restituisce il numero intero corrispondente

```
Quanti anni hai? 21  
Fra due anni avrai 23 anni
```

Altrimenti? Esperimento...

- In "gergo" si dice che la stringa viene "convertita" in un numero intero, ma... attenzione: la stringa rimane così com'è! Il contenuto della variabile **ageString** NON cambia, tale variabile viene soltanto letta e, analizzando i caratteri che contiene, viene generato un numero intero. Attenzione al significato delle parole... a volte usate "con leggerezza" dai programmatori. Bisogna sempre ragionare e pensare a ciò che accade nell'interprete.

Da qui in avanti, non sempre gli esempi saranno programmi completi...

# Valore restituito da una funzione

```
ageString = input("Quanti anni hai? ")  
x = 3 + int(ageString)
```

- ❑ Se l'invocazione di una funzione è presente all'interno di un'espressione (in questo esempio, la funzione **int** lo è), il **valore restituito** dalla funzione **sostituisce** semanticamente l'invocazione stessa durante la valutazione dell'espressione
  - **Prima di valutare un'espressione, vengono eseguite tutte le invocazioni di funzioni che eventualmente contiene**
  - **Non sempre l'invocazione di una funzione restituisce un valore** (dipende dal progetto della funzione stessa e dal suo obiettivo): in tal caso, l'invocazione della funzione **non** deve comparire all'interno di una espressione, bensì solamente in un enunciato a sé stante
    - Es. la funzione **print** non restituisce alcun valore

# Valore restituito da una funzione

```
ageString = input("Quanti anni hai? ")  
x = 3 + int(ageString)
```

```
# equivalente...  
x = 3 + int(input("Quanti anni hai? "))
```

- ❑ In questo esempio vediamo "invocazioni annidate", **una** dentro **l'altra**
  - Per eseguire l'invocazione di **int**, occorre conoscere il valore restituito dall'invocazione di **input**, che, quindi, verrà eseguita per prima
  - Tutto questo (ovviamente?) avviene prima di eseguire l'addizione con 3
  - Rivedremo queste azioni di "eliminazione di una variabile"
    - In questo esempio è stata eliminata la variabile **ageString**



Vediamo tre programmi equivalenti: questa è (probabilmente) la versione migliore

# Eliminazione di una variabile

```
ageString = input("Quanti anni hai? ")
age = int(ageString)
print("Fra due anni avrai", age + 2, "anni")
```

□ Osservo che la variabile **ageString** viene **"scritta" e "letta" una sola volta**

- In questa situazione, **la variabile può essere "eliminata"** scrivendo, nella posizione in cui la si legge, al posto del suo nome, **la stessa espressione** (in questo caso, invocazione di funzione) usata nel lato destro dell'unico enunciato di assegnazione in cui essa compare a sinistra, con il quale è stata definita

e scritta

```
age = int(input("Quanti anni hai? "))
print("Fra due anni avrai", age + 2, "anni")
```

- Anche **age** viene usata una sola volta: proviamo a eliminarla

```
print("Fra due anni avrai", int(input("Quanti anni hai? ")) + 2, "anni")
```

- Ora, però, l'unico enunciato che costituisce il programma è diventato un po' troppo articolato e complesso, si capisce a fatica!
- **A volte si usano variabili al solo scopo di rendere il codice più leggibile!**
- Ma senza esagerare... infatti programmare è (anche) un'arte! ☺

**Ancora aritmetica...**

# Espressioni "in linea"

- ❑ In algebra siamo abituati a scrivere espressioni "bidimensionali", nel senso che, ad esempio, usiamo il segno orizzontale di frazione per poter scrivere il numeratore al di sopra e il denominatore al di sotto  $\frac{a+b}{2}$
- ❑ Questo in Python non si può fare, perché possiamo scrivere soltanto "righe di codice", quindi le frazioni vanno ridotte a espressioni unidimensionali (come anche le potenze, dove gli esponenti non possono essere scritti "in alto a destra" rispetto alla base)
  - Bisogna fare attenzione perché questo a volte richiede parentesi necessarie che NON erano presenti nella forma originaria

$$\frac{a+b}{2} \longrightarrow (a+b)/2 \neq a+b/2$$
$$2^{a+b} \longrightarrow 2^{**}(a+b) \neq 2^{**}a+b$$

Bisogna conoscere le regole di precedenza dell'algebra...

**Risultati intermedi  
dell'elaborazione svolta da un  
programma**



# Memorizzare risultati intermedi

- ❑ La memorizzazione di risultati intermedi in una variabile può essere utile quando
  - L'intera espressione senza risultati intermedi sarebbe di difficile comprensione
  - **Un risultato intermedio verrà utilizzato più volte nel seguito:**  
**senza la variabile intermedia, bisognerebbe rifare i calcoli più volte**
- ❑ Esempio: soluzione di un'equazione di secondo grado (con coefficienti interi, vedremo più avanti come leggere numeri reali...)

```
a = int(input("Coefficiente a: "))
b = int(input("Coefficiente b: "))
c = int(input("Coefficiente c: "))
# calcolo la radice del discriminante
# che userò due volte
d = (b*b - 4*a*c)**(1/2)
print("Soluzione 1:", (-b - d)/(2*a))
print("Soluzione 2:", (-b + d)/(2*a))
# attenzione alle parentesi dopo la divisione...
```

# Visualizzare risultati intermedi

- Durante il collaudo, dopo aver scoperto che un programma non si comporta come previsto, inizia la difficile fase della **diagnosi** dell'errore
  - Bisogna capire DOVE si trova l'errore e qual è
- **A questo scopo può essere molto utile aggiungere enunciati che **visualizzino** alcuni risultati intermedi**
  - Se, ad esempio, i primi due risultati intermedi visualizzati sono corretti, l'errore si troverà nella sezione di programma successiva

# Visualizzare risultati intermedi

```
a = int(input("Coefficiente a: "))
b = int(input("Coefficiente b: "))
c = int(input("Coefficiente c: "))
# calcolo la radice del discriminante
d = (b*b - 4*a*c)**(1/2)
print("DEBUG (discriminante):", d)
sol1 = (-b - d) / (2*a)
sol2 = (-b + d) / (2*a)
print("Soluzione 1:", sol1)
print("Soluzione 2:", sol2)
```

Rileggere  
questa slide  
OGNI DUE  
GIORNI 😊

- ❑ Se le soluzioni sono sbagliate ma il discriminante è corretto, l'errore sarà nel calcolo delle soluzioni
- ❑ È bene identificare in modo speciale (es. DEBUG) tali enunciati aggiunti, in modo da poterli eliminare facilmente quando il collaudo sarà terminato
  - **In generale, un programma NON deve visualizzare informazioni diverse da quelle previste nelle sue specifiche di progetto, per non "confondere" l'utente**

# Visualizzare risultati intermedi

```
a = int(input("Coefficiente a: "))
b = int(input("Coefficiente b: "))
c = int(input("Coefficiente c: "))
# calcolo la radice del discriminante
d = (b*b - 4*a*c)**(1/2)
# print("DEBUG (discriminante):", d)
sol1 = (-b - d) / (2*a)
sol2 = (-b + d) / (2*a)
print("Soluzione 1:", sol1)
print("Soluzione 2:", sol2)
```

Rileggere  
questa slide  
OGNI  
SETTIMANA 😊

- ❑ Invece di eliminare gli enunciati di debugging dopo aver collaudato il programma, è comodo **COMMENTARLI** ma lasciarli nel codice... in questo modo sarà semplice ripristinarli nel caso in cui si debba procedere a un nuovo collaudo (e nuovo debugging) dopo un aggiornamento del codice

Il materiale necessario per il  
Laboratorio 01  
termina qui

**Alcuni approfondimenti**

# Ancora sulle stringhe...

- ❑ Ricordiamoci sempre che ciò che scriviamo ALL'INTERNO di una stringa non viene analizzato dall'interprete Python
  - Finché non scopriremo qualche eccezione a questa regola...

- ❑ Quindi attenzione a questo esempio

```
name = input("Come ti chiami? ")  
print("Ciao, name")
```

Esempio di errore con diagnosi difficile...

Come ti chiami? **Ugo**  
Ciao, name

Due esecuzioni del programma

Come ti chiami? **Anna**  
Ciao, name

- ❑ Questo NON è un utilizzo della variabile **name**! Perché il nome della variabile è "nascosto" all'interno della stringa, quindi "non viene visto" dall'interprete
  - Meno male! Altrimenti, se ne avessi bisogno, non potrei più visualizzare la parola **name** 😊, visualizzerei sempre il contenuto della variabile **name**

# Confusione potenziale...

## ❑ Fare attenzione a concetti diversi...

- **ALL'INTERNO di un programma**, le stringhe sono sequenze di caratteri racchiuse da virgolette (o singoli apici...)
- **Nelle comunicazioni tra programma e utente** (nelle due direzioni, output e input di dati), vengono usate sequenze di caratteri, ma non ci sono le virgolette (a meno che non facciano parte della comunicazione stessa)
  - Quando **print** visualizza una stringa, "toglie" le virgolette
  - Quando **input** acquisisce una sequenza di caratteri, "inserisce" le virgolette, per farla diventare una stringa (l'utente NON digita le virgolette rispondendo alla richiesta di **input**)

## ❑ **Il concetto di stringa è relativo AI PROGRAMMI**, non alle comunicazioni tra programma e utente

- **L'utente non conosce il concetto di stringa...**



# Invocazione di una funzione

□ Approfondiamo un po'... durante l'esecuzione di un programma Python, **quando viene “invocata una funzione”, cosa succede?**

- Viene *sospesa* l'esecuzione del programma “invocante”, in attesa che termini l'esecuzione del codice corrispondente alla funzione “invocata”
  - E così via! Se la funzione invocata risulta essere, a sua volta, invocante di un'altra funzione... si crea una lista (tecnicamente, come vedremo meglio, una “pila”, *stack*, chiamata *runtime stack*) di funzioni sospese, che via via si “rianimeranno”: quando l'unica funzione in esecuzione termina, il programma invocante “si rianima”
    - **C'è sempre una sola funzione in esecuzione!**
    - Possiamo pensare al nostro programma in esecuzione come a una specie di “funzione iniziale”: se è sospesa, è sempre in fondo alla pila; quando termina, pone fine all'intero programma




**Lezione 08**  
**16/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Numeri non interi...

```
age = int(input("Quanti anni hai? "))  
money = float(input("Quanti dollari hai? "))
```

- ❑ La funzione **float**, predefinita in Python, vuole come argomento una stringa

 i cui caratteri descrivano un numero intero oppure un numero "con la virgola" (cioè contenente il punto separatore decimale, un formato chiamato *floating point* in inglese) e restituisce il numero corrispondente, in ogni caso sotto forma di numero "con la virgola" ("virgola zero" se è intero, es. 3.0)

- ❑ **Perché ci servono due diversi tipi di numeri?** Non potremmo usare soltanto i "numeri con la virgola" (che, come vedremo, sono il modo con cui l'informatica rappresenta un sottoinsieme finito dei numeri reali)? Dopo tutto, un numero intero non è altro che un numero reale con parte frazionaria uguale a zero...

- ❑ Per il momento, accontentiamoci di sapere che **elaborare in modo specifico i numeri interi fa risparmiare tempo durante l'esecuzione** del programma (e spesso fa risparmiare anche memoria)



- Cioè il calcolo  $2 + 3$  è più veloce del calcolo  $2.0 + 3.0$

**Ancora sull'enunciato di  
assegnazione**

# Enunciato di assegnazione

## □ Ripensiamo alla semantica dell'enunciato di assegnazione

- **Alla variabile nominata a sinistra dell'uguale viene assegnato il valore ottenuto valutando l'espressione che si trova a destra dell'uguale**
- Finora a destra abbiamo spesso avuto soltanto l'invocazione di una funzione (es. `input` oppure `int`), ma cosa ci può essere?

- Ad esempio, come sappiamo, un'espressione aritmetica

```
age = int(input("Quanti anni hai? "))  
newAge = age + 2  
print("Fra due anni avrai", newAge, "anni")
```

- A volte il vecchio valore di una variabile, dopo averlo letto, non serve più... anzi, ci è utile che tale valore venga **modificato**:  
si può fare!



```
age = int(input("Quanti anni hai? "))  
age = age + 2  
print("Fra due anni avrai", age, "anni")
```

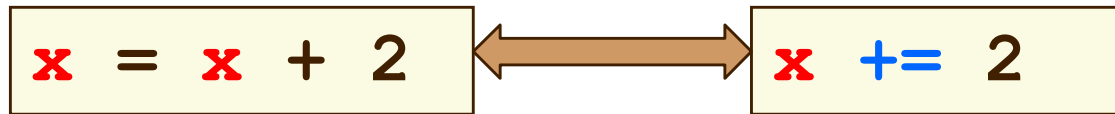
- Ho anche risparmiato spazio in memoria: uso una variabile in meno...  
perché **riutilizzo** una variabile, il cui vecchio valore non mi serviva più

# Enunciato di assegnazione

- ❑ Nella programmazione, cosa significa **age** = **age** + 2 ?
- ❑ Se facciamo la domanda a un matematico, risponderà:
  - questa equazione è impossibile, nessun valore di **age** la soddisfa
  - perché è abituato a usare il segno = per indicare un'uguaglianza tra parte sinistra e parte destra di un'equazione
- ❑ **Ma in informatica il segno = si usa per indicare un'operazione di assegnazione, che ha una semantica completamente diversa**
  - Significa: **valuta** l'espressione che si trova a destra (eventualmente leggendo le variabili utilizzate ed eseguendo le funzioni invocate e/o le operazioni aritmetiche indicate), **poi scrivi** nella variabile di sinistra il risultato della valutazione (detto "valore dell'espressione")
  - Quindi tale assegnazione è sensata: leggi il valore di **age**, aggiungi 2 e scrivi il risultato in **age** (così come lo scriveresti in una nuova variabile, tanto l'hai già letta e utilizzata). Cioè, in breve, incrementa di 2 unità il valore di **age**.
  - Per questo **sarebbe (stato) meglio usare un simbolo diverso per l'assegnazione**... il simbolo = è fuorviante, abbiamo studiato matematica!

# Assegnazione con "scorciatoia"

- Capita spesso che il nome della variabile a cui si sta assegnando un nuovo valore compaia anche nell'espressione che si trova alla destra del carattere =



- Python mette a disposizione parecchie "scorciatoie" (*shortcut*), in verità molto utilizzate dai programmatori
- La forma a destra (totalmente equivalente a quella di sinistra) solitamente si legge "incrementa **x** di 2", forse più intuitivo...
  - Le due forme sono equivalenti perché **l'interprete**, prima di tradurre il codice in formato eseguibile, **traduce la forma di destra in quella di sinistra!**
  - Funziona con tutti gli operatori aritmetici, ma attenzione a quelli non commutativi...

`x = x - 2` equivale a `x -= 2`  
`x = 2 - x` **NON** equivale a `x -= 2`

Stringhe



# Il tipo di dati “stringa”

- ❑ I dati più importanti nella maggior parte dei nostri programmi saranno i *numeri* e le *stringhe*
- ❑ Una *stringa* è una **sequenza di caratteri**, che, in Python (come in molti altri linguaggi), vanno *racchiusi tra virgolette* o singoli apici

- *I delimitatori NON fanno parte della stringa*

- ❑ Possiamo *definire variabili che contengono una stringa*

```
name = "John"
```

- Possiamo ovviamente *assegnare un nuovo valore* a una variabile che contiene una stringa

```
name = "Michael"
```

"Hello"

Sintatticamente è un *letterale* (o, ancora meglio, un *valore letterale*) di tipo stringa, così come 15000 è un letterale ("literal") di tipo intero

# Lunghezza di una stringa

- ❑ Vedremo che in alcuni casi ci interesserà sapere quanto è lunga una stringa, cioè quanti sono i caratteri che la compongono
  - ricordando che i delimitatori, virgolette o apici che siano, **non** ne fanno parte
- ❑ Questa informazione viene calcolata e restituita dalla funzione predefinita **len** (parte iniziale della parola *length*), che riceve una stringa come argomento

Novità: commento  
“a fine riga”

```
name = "John" # più interessante: name=input(...)
n = len(name)
print("La stringa", name, "ha", n, "caratteri")
# attenzione al corretto inserimento di spazi
# nelle frasi che si visualizzano...
```

# La stringa vuota



Sembra una stringa inutile, ma a volte ci servirà

- Una *stringa di lunghezza zero*, che non contiene caratteri, si chiama *stringa vuota* e si indica con due caratteri delimitatori *consecutivi*, senza spazi interposti

```
emptyString = ""  
print(len(emptyString))
```

0

Dato che **print** accetta come parametro un'espressione avente valore numerico, (ovviamente?) accetta anche, come parametro, l'invocazione di un metodo che restituisce un valore numerico (perché tale invocazione è proprio un esempio di espressione numerica)

# Elaborazione di stringhe

- ❑ Le informazioni numeriche vengono solitamente elaborate mediante strumenti matematici
  - Addizioni, sottrazioni, ecc.
- ❑ Come elaboriamo le stringhe?
  - **Concatenazione di stringhe, cioè creazione di una stringa a partire dal contenuto di altre stringhe**
    - Per il momento ci occupiamo soltanto della concatenazione
  - Ispezione di un carattere in una specifica posizione
  - Estrazione di sottostringhe, cioè creazione di una stringa a partire da una porzione del contenuto di un'altra stringa
  - Conversioni varie (maiuscole/minuscole, ecc.)
  - Ecc.

# Concatenazione di stringhe

- ❑ Per concatenare due stringhe si usa l'*operatore* **+**

```
s1 = "eu" # pessimi nomi di variabili, per brevità  
s2 = "ro"  
s3 = s1 + s2 # s3 contiene "euro"
```

- ❑ L'operatore di concatenazione è identico all'operatore di addizione, ma la sua **semantica** è completamente diversa (si parla di *grammatica dipendente dal contesto*), ad esempio l'operatore di concatenazione **NON** è **commutativo!**

```
s3 = s2 + s1 # s3 contiene "roeu"
```

- ❑ I due operandi dell'operatore **+** devono essere entrambi stringhe o entrambi numeri (e in questo secondo caso viene effettuata un'addizione), altrimenti si verifica un errore

# Concatenazione di stringhe

- ❑ Se vogliamo **concatenare una stringa e un numero** (cioè un valore numerico), dobbiamo prima "convertire il numero in stringa", cioè generare una stringa che abbia come caratteri la descrizione del valore del numero
  - Per la conversione inversa abbiamo visto **int** e **float**
  - Si usa la funzione **str**, che accetta sia numeri interi sia numeri frazionari (e ovviamente anche espressioni numeriche, di cui converte il valore risultante dalla valutazione)

```
eu = "euro"  
s = str(12.45) + eu  
print(s) # eu eliminabile
```

12.45euro

```
eu = "euro"  
# aggiungiamo uno spazio...  
s = str(12.45) + " " + eu  
print(s) # s eliminabile...
```

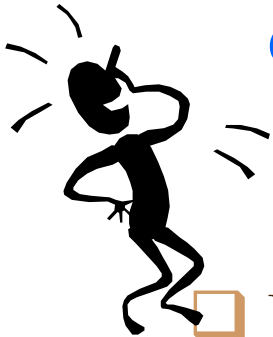
12.45 euro

# Decisioni

# Gestione di un conto corrente

```
balance = 10000 # saldo iniziale
amount = float(input("Quanto vuoi prelevare? "))
balance = balance - amount
print("Saldo:", balance)
```

- ❑ Questo "programma" consente di prelevare tutto il denaro che si vuole



- il saldo **balance** può diventare negativo!!

```
balance = balance - amount
```

- ❑ È una situazione assai poco realistica!
- ❑ Il programma *deve controllare il saldo e agire di conseguenza*, consentendo il prelievo oppure no
  - Cioè vogliamo che l'enunciato **balance = balance - amount** venga eseguito **a condizione che amount** non sia maggiore di **balance**



# L'enunciato if

Attenzione ai due punti!

```
if amount <= balance :  
    balance = balance - amount
```

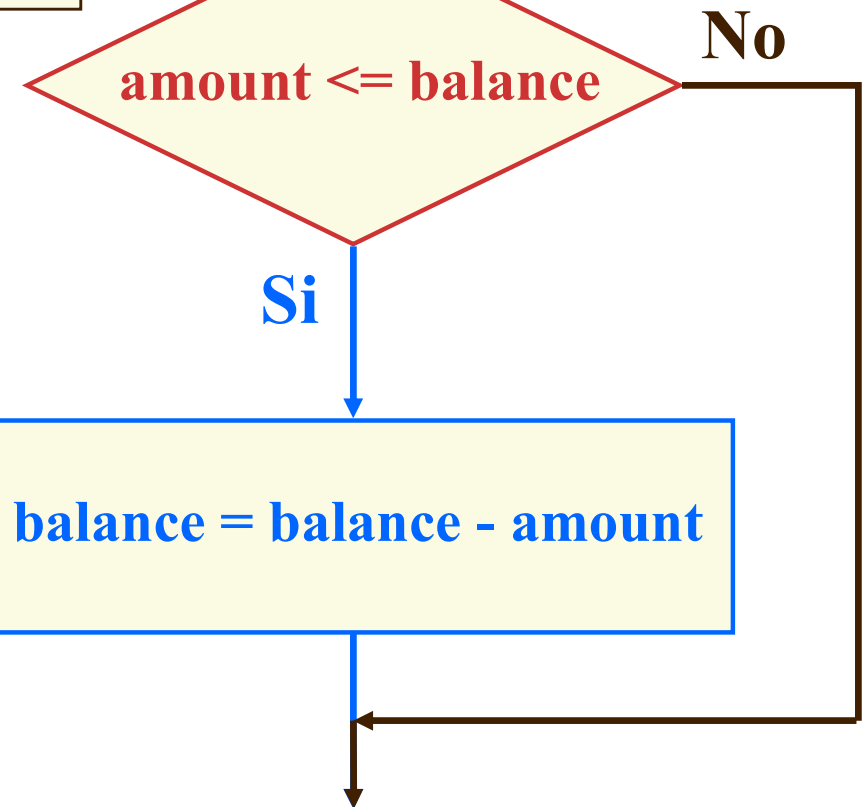
Questo è un  
“diagramma di  
flusso” o  
flow-chart



- ❑ L'enunciato **if** si usa per prendere una decisione ed è diviso in due parti

- una *verifica di condizione*
- un *corpo*

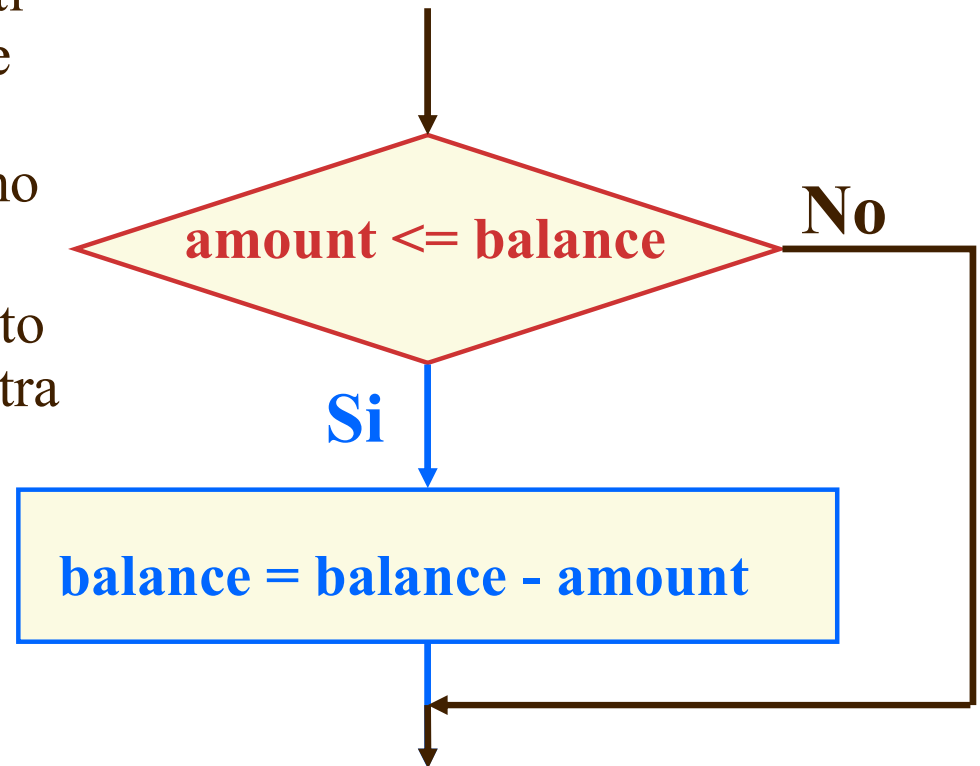
- ❑ Il corpo viene eseguito se e solo se la verifica ha successo



Attenzione all'indentazione obbligatoria!

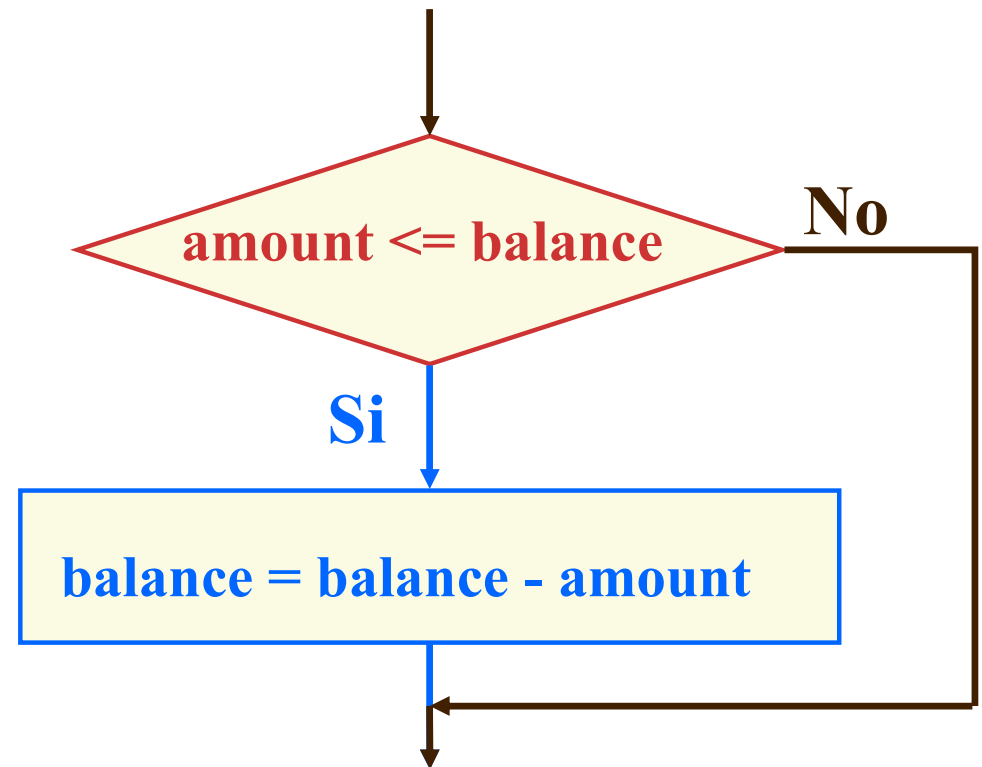
# Diagrammi di flusso

- ❑ I diagrammi di flusso rappresentano **graficamente** la successione di enunciati che vengono eseguiti in un programma e sono utili per rappresentare algoritmi semplici o parti “critiche” di un algoritmo
  - Basta seguire le frecce...  
Che sono DI SOLITO dirette dall'alto verso il basso e da sinistra verso destra
- ❑ Un **blocco rettangolare** contiene istruzioni che vengono eseguite attraversandolo
  - Può avere più ingressi e deve avere **un'unica uscita** (perché, se rappresenta un algoritmo, non ci possono essere ambiguità...)



# Diagrammi di flusso

- Un **blocco romboidale** contiene una domanda
  - Può avere più ingressi e deve avere **almeno due uscite**
  - A **ogni possibile risposta** alla domanda, deve corrispondere **una e una sola uscita** dal blocco, identificata mediante etichette
- Istruzioni e domande possono essere espresse in qualsiasi linguaggio, anche naturale (o misto...)



**Lezione 09**  
**18/10/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Un nuovo problema

- Proviamo ora a visualizzare un messaggio d'errore se (e solo se!) il prelievo non è consentito

```
if amount <= balance :  
    balance = balance - amount  
if amount > balance :  
    print("Conto scoperto")
```

**Sembra che  
funzioni!**

- Primo problema di questo codice: se si modifica la prima condizione, bisogna ricordarsi di modificare anche la seconda
  - es. viene concesso un **FIDO** sul conto, che può così “andare in rosso”

```
if amount <= balance + FIDO :  
    balance = balance - amount  
if amount > balance + FIDO :  
    print("Conto scoperto")
```

- In generale, è DECISAMENTE preferibile **evitare**, per quanto possibile, **casi in cui una modifica richiede necessariamente un'altra modifica** conseguente in un diverso punto del codice

# Un nuovo problema

```
if amount <= balance :  
    balance = balance - amount  
if amount > balance :  
    print("Conto scoperto")
```

Sembra che  
funzioni!  
**Invece no...**

- ❑ **Secondo problema** di questo codice: se il corpo del primo **if** viene eseguito, la verifica del secondo **if** usa il *nuovo* valore di **balance**, dando luogo a un errore logico
  - Questa situazione accade *ogni volta che si preleva più della metà del saldo disponibile*: il prelievo viene correttamente eseguito (diminuendo il saldo), ma viene anche (erroneamente) visualizzato il messaggio di “Conto scoperto”
  - I due corpi, secondo l’algoritmo, dovrebbero essere sempre eseguiti **IN ALTERNATIVA**, ma così non è
  - Come possiamo codificare correttamente questo algoritmo?

Osservare bene le  
indentazioni

# La clausola else

- ❑ Per realizzare un'*alternativa*, si utilizza **else**, la clausola facoltativa dell'enunciato **if**

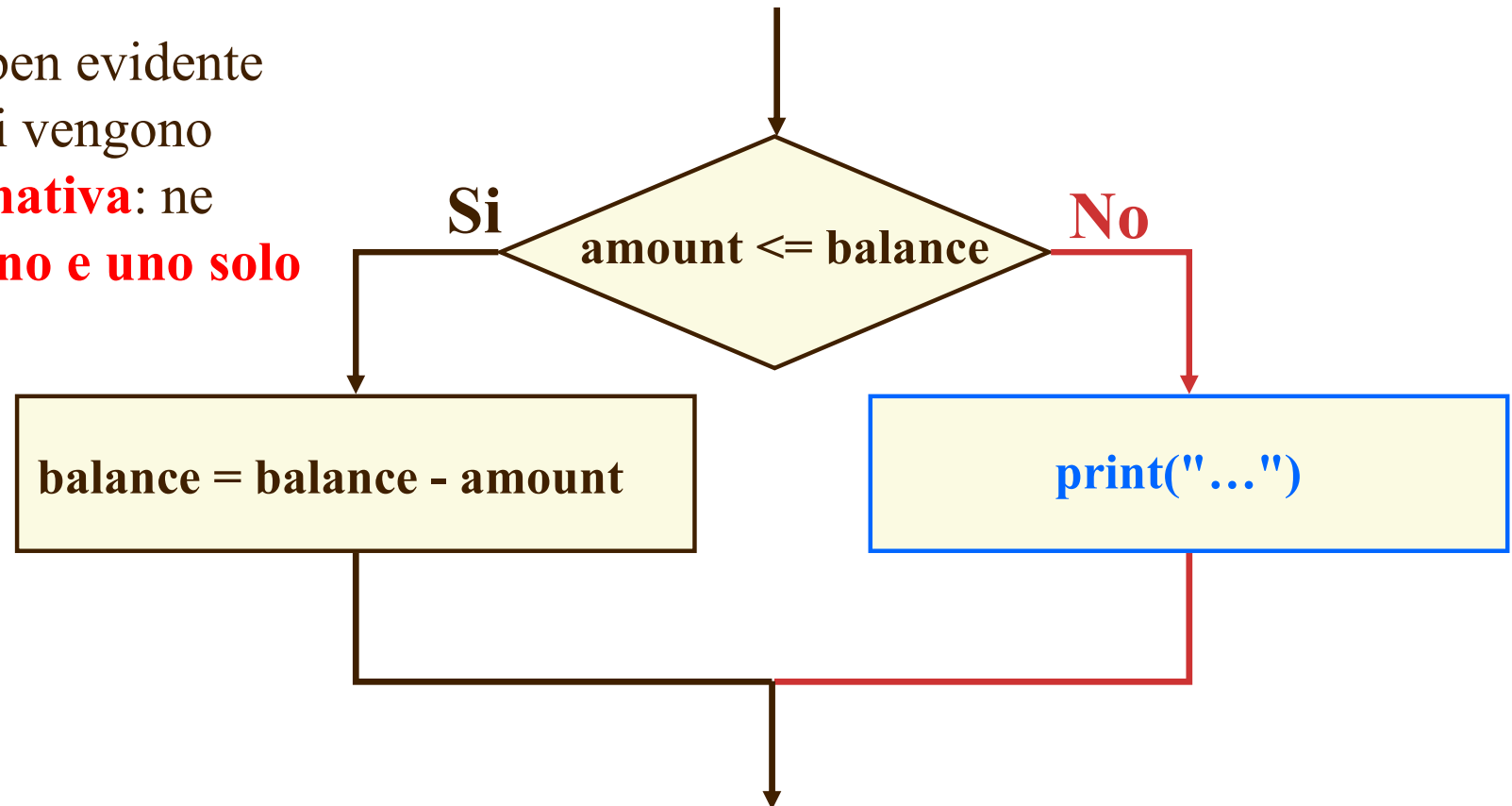
```
if amount <= balance :  
    balance = balance - amount # primo corpo  
else : # notare: "due punti" anche qui  
    print("Conto scoperto")    # secondo corpo
```

- ❑ **Vantaggio:** ora c'è *una sola verifica*
  - se la verifica ha successo, viene eseguito il *primo* corpo dell'enunciato **if/else**
  - *altrimenti*, viene eseguito il *secondo* corpo
    - **NON VIENE RIPETUTA LA VERIFICA**
- ❑ Non è un costrutto sintattico *necessario*, ma è utile
  - Esercizio: realizzare correttamente il medesimo algoritmo senza utilizzare la clausola **else**

# La clausola else

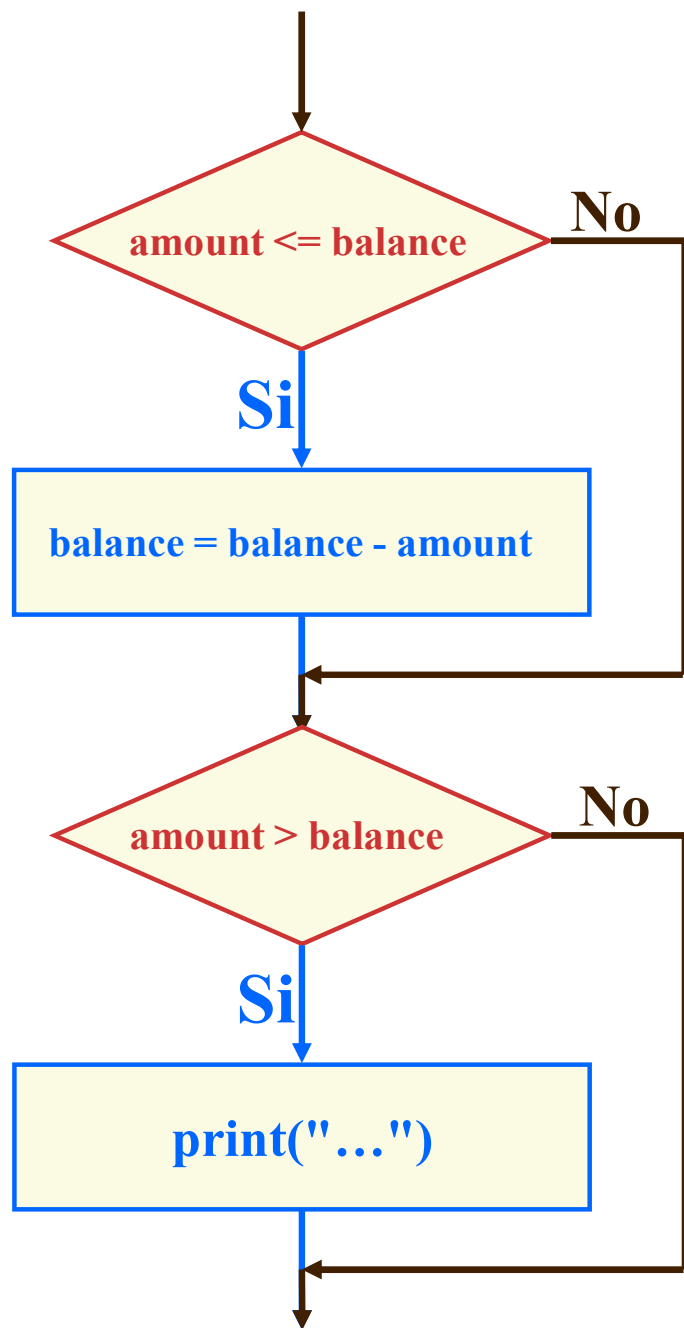
```
if amount <= balance :  
    balance = balance - amount # primo corpo  
else :  
    print("Conto scoperto")    # secondo corpo
```

È graficamente ben evidente  
che i due blocchi vengono  
eseguiti in **alternativa**: ne  
viene eseguito **uno e uno solo**



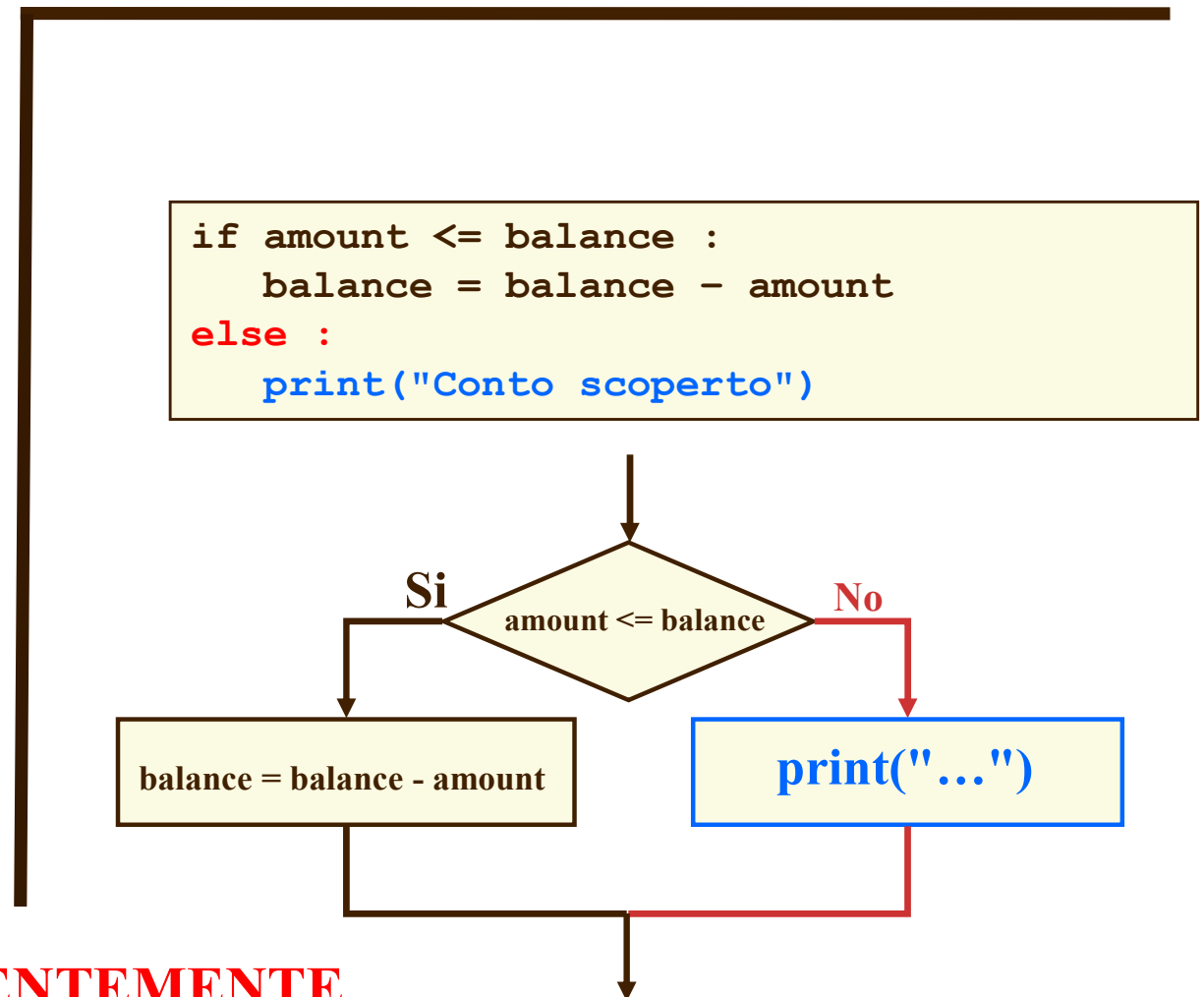
**Il programma prosegue dopo che le due "strade" si sono riunite**





```

if amount <= balance :
    balance = balance - amount
if amount > balance :
    print("Conto scoperto")
# NON funziona correttamente!
  
```



```

if amount <= balance :
    balance = balance - amount
else :
    print("Conto scoperto")
  
```

**Sono EVIDENTEMENTE  
due algoritmi diversi**

# Esercizio

I contenuti delle due variabili  
diventano uguali in questo  
momento ma non lo  
rimangono per sempre!

- ❑ Realizzare correttamente il medesimo algoritmo senza utilizzare la clausola **else**

```
# non è un programma completo... manca l'assegnazione di
# un valore iniziale alle variabili balance e amount
# faremo spesso così! per risparmiare spazio
testBalance = balance # eventualmente + FIDO...
if amount <= testBalance :
    balance = balance - amount # la modifica di balance
                                # non ha ovviamente
                                # effetto su testBalance
if amount > testBalance :
    print("Conto scoperto")
#
# nel seguito bisogna usare balance,
# che contiene il valore correttamente aggiornato
# del saldo; testBalance non serve più
```

# Soluzione alternativa

- Dato che il corpo del primo **if** modifica **balance** ma il corpo del secondo non lo fa, scambio tra loro i due **if**: in questo modo diventano mutuamente esclusivi

```
if amount > balance :  
    print("Conto scoperto")  
if amount <= balance :  
    balance = balance - amount
```

- Ora il comportamento del programma è sempre corretto
- Nessuna di queste due soluzioni, però, è applicabile in generale per evitare di usare la clausola **else**
  - L'informatica teorica ci dice che è sempre possibile fare a meno di **else**, ma bisogna analizzare con cura il problema specifico



# L'enunciato if

## ❑ Sintassi:

```
if condizione :  
    enunciato1
```

```
if condizione :  
    enunciato1  
else :  
    enunciato2
```

- ❑ Scopo: eseguire *enunciato1* se e solo se la *condizione* è vera; se è presente la clausola *else*, eseguire *enunciato2* se e solo se la *condizione* è falsa
- ❑ Le righe che contengono **if** e **else** devono iniziare nella stessa colonna di testo, mentre i loro enunciati devono iniziare **più a destra**
  - Tecnicamente almeno uno spazio più a destra, **di solito 3 o 4 spazi**: fatta una scelta, adottarla per l'intero file
- ❑ Le righe che contengono **if** e **else** devono terminare con un carattere "due punti"

# Blocco di enunciati

- ❑ Spesso avremo bisogno di inserire più enunciati nel corpo di un enunciato **if** (o della sua clausola **else**)
  - **Perché ci saranno più enunciati di cui vogliamo condizionare l'esecuzione alla verifica di una stessa condizione**
- ❑ Non c'è problema, basta scrivere gli enunciati condizionati usando **lo stesso livello di rientro verso destra**: creeranno un ***blocco di enunciati***, che può essere usato come corpo

```
if amount <= balance :  
    balance = balance - amount  
    print("Prelievo accordato")  
else :  
    print("Conto scoperto")  
    print("Prelievo non accordato")
```

# Blocco di enunciati

- Dal punto di vista sintattico, **un blocco di enunciati** (cioè una sequenza di enunciati che iniziano nella stessa colonna) **è un enunciato!**
- Quindi, **in un punto che sintatticamente prevede la presenza di un enunciato, possiamo sempre scrivere un blocco di enunciati**
  - Esempio: il corpo di un **if**, il corpo di un **else**
    - Vedremo altri esempi
  - **Non diremo più:** qui ci può stare un enunciato, quindi ci posso mettere un blocco di enunciati.  
Da questo momento in poi, è ovvio!

**Confrontare valori numerici**

# Confrontare valori numerici

- ❑ Le *condizioni* dell'enunciato **if** sono molto spesso dei *confronti tra i valori di due espressioni numeriche*

```
if x >= 0 :
```

- ❑ Gli *operatori di confronto* si chiamano

## *operatori relazionali*

- ❑ **Attenzione:** negli operatori costituiti da due caratteri *non* vanno inseriti spazi intermedi
- ❑ Ovviamente i valori confrontati possono essere il risultato della valutazione di espressioni di tipo numerico

```
if x + 2 >= 0 :
```

|    |                   |
|----|-------------------|
| >  | Maggiore          |
| >= | Maggiore o uguale |
| <  | Minore            |
| <= | Minore o uguale   |
| == | Uguale            |
| != | Diverso           |



# Operatori relazionali

- ❑ Fare molta attenzione alla differenza tra operatore relazionale **==** e operatore di assegnazione **=**

```
a = 5          # assegna 5 alla variabile a

if a == 5 :    # esegue enunciato
    enunciato  # se e solo se a è uguale a 5

if a = 5 :     # fortunatamente (e diversamente
    enunciato  # da altri linguaggi)
               # ERRORE DI SINTASSI
```

# Operatori relazionali

❑ (Come in matematica) Gli operatori relazionali hanno una precedenza inferiore rispetto agli operatori aritmetici

❑ Quindi `if a - 1 < 25 :`

equivale a `if (a - 1) < 25 :`

Le parentesi qui non servono!

(ma ovviamente si possono mettere...)

# Errori di programmazione

# Errori di programmazione

```
print("Hello, World!")
```

- ❑ L'attività di programmazione, come ogni altra *attività di progettazione umana*, è soggetta a errori di vario tipo

- errori di sintassi

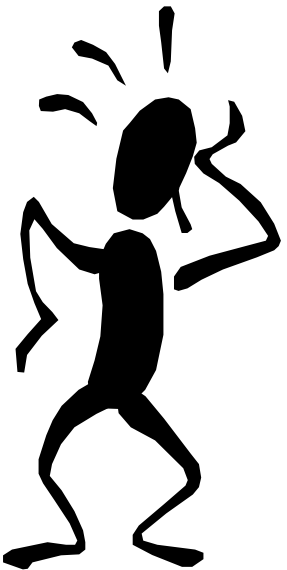
```
sprint("Hello, World!")
```

- errori in esecuzione

```
print(1/0)
```

- errori logici

```
print("Hel1, World!")
```



# Errori di sintassi

```
sprint("Hello, World!")
```

- ❑ In questo caso l'interprete riesce agevolmente ad individuare e segnalare l'errore di sintassi, perché identifica un nome (**sprint**) che **non conosce**: non riesce a trovarlo in libreria
- ❑ L'errore di sintassi viene segnalato indicando
  - il nome del file contenente l'errore
  - il numero della riga contenente l'errore
  - il tipo di errore (in questo caso, **NameError**)

```
NameError: name 'sprint' is not defined
```

# Errori di sintassi

```
print("Hello, 2+3)
```

- ❑ Non sempre la segnalazione d'errore è precisa... qui volevo che il programma visualizzasse **Hello 5** ma mi sono dimenticato di terminare la stringa "**Hello**"

```
print("Hello, 2+3)
```

^

```
SyntaxError: EOL while scanning string literal
```

- ❑ Innanzitutto, il messaggio è un po' criptico: "**EOL while...**"
  - Cosa significa **EOL** ? Significa **End of line**... quindi "raggiunta la fine della riga mentre si leggeva il contenuto di una stringa" o, per meglio dire, "ti sei dimenticato di terminare la stringa!"
- ❑ Poi, perché la segnalazione del punto in cui c'è l'errore è sulla parentesi chiusa e non vicino alla virgola?
  - Perché **l'interprete non può indovinare** dove avrei voluto terminare la stringa! Fa del suo meglio...

# Errori in esecuzione

```
print(1/0)
```

- ❑ In questo enunciato non ci sono errori di sintassi (l'espressione aritmetica è scritta correttamente), infatti l'interprete prova a tradurlo e a eseguirlo, ma **si interrompe durante l'esecuzione** con una segnalazione d'errore

```
ZeroDivisionError
```

- Come in aritmetica, anche in Python non è possibile dividere per zero
- ❑ Tecnicamente si dice che, durante l'esecuzione del programma, si è verificata **una eccezione** (cioè una "situazione eccezionale")

# Errori logici

manca un  
carattere

```
print("Hell, World!")
```

- ❑ Questo errore, invece, *non viene segnalato dall'interprete*, che non può sapere che cosa il programmatore avesse intenzione di far scrivere al programma
  - si ha un errore durante l'*esecuzione* del programma, perché viene visualizzata un'informazione **diversa da quella prevista**, ma **non si verifica un'eccezione**: l'esecuzione del programma prosegue e termina normalmente
  - verificare che un programma visualizzi esattamente ciò che è previsto è una responsabilità tipica del **collaudo**



# Errori logici

- ❑ Sono molto più insidiosi degli errori di sintassi
  - *il programma* viene eseguito senza segnalazioni d'errore, ma *non fa quello che dovrebbe fare*
- ❑ L'eliminazione degli errori logici richiede molta *pazienza*, eseguendo il programma e osservando con attenzione i risultati prodotti
  - *è necessario collaudare i programmi, come qualsiasi altro prodotto dell'ingegneria*
- ❑ Si usano programmi specifici (*debugger*) per trovare gli errori logici (*bug*) in un programma
  - **noi NON useremo un debugger** (è "troppo difficile")



**Lezione 10**  
**22/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Valori booleani, operatori booleani ed espressioni booleane

# Il tipo di dato *booleano*

- Ogni *espressione* ha un *valore*, che si ottiene *valutando* l'espressione, cioè sostituendo le variabili con i loro valori e le invocazioni di funzioni con i valori restituiti.  
Se la variabile **x** contiene un numero, allora
  - **x + 10** è un'espressione aritmetica e ha un valore numerico
  - **x < 10** è un'espressione *relazionale* e ha un valore *booleano*,  
dal nome del matematico George Boole (1815-1864), pioniere della logica
- Un'espressione relazionale (detta anche "logica") ha un valore *vero* o *falso* (**True** o **False**, in Python)
  - Attenzione: in altri linguaggi si usa **TRUE** e **FALSE**, in maiuscolo, oppure **true** e **false**, in minuscolo
  - Attenzione: **True** e **False** non sono stringhe!  
Sono (gli unici!) **valori letterali di tipo booleano**



# Rivediamo l'enunciato if

## □ Sintassi

```
if condizione :  
    enunciato1
```

```
if condizione :  
    enunciato1  
else :  
    enunciato2
```

*condizione* deve essere un'espressione booleana

- Il corpo dell'**if** (cioè *enunciato1*) viene eseguito se e solo se l'espressione che rappresenta la *condizione* assume il valore **True**
- Il corpo dell'**else** (cioè *enunciato2*), se presente, viene eseguito se e solo se l'espressione che rappresenta la *condizione* assume il valore **False**

# Gli operatori booleani o logici

- ❑ Gli **operatori** booleani o logici servono a svolgere *operazioni* su valori booleani

```
if x > 10 and x < 20 :  
    # esegue se x è maggiore di 10 e minore di 20  
    ... # corpo da eseguire
```

- ❑ L'operatore **and** (in italiano *e*) combina due espressioni booleane in una sola, che risulta *vera se e solo se sono tutte vere*
- ❑ L'operatore **or** (in italiano *oppure*) combina due espressioni booleane in una sola, che risulta *vera se e solo se almeno una è vera*
- ❑ L'operatore **not** (in italiano *non*) *inverte* il valore di una singola espressione booleana
- ❑ Sono tre nuove parole chiavi: **and**, **or**, **not**

# Gli operatori booleani o logici

- ❑ In un'espressione booleana con più operatori, la valutazione viene fatta da sinistra a destra, dando la **precedenza** all'operatore *not*, poi all'operatore *and*, infine all'operatore *or*
  - Tutti hanno precedenza **inferiore** a quella di qualsiasi operatore aritmetico o relazionale
- ❑ L'ordine di valutazione può comunque essere alterato dalle parentesi tonde

```
if not(x < 0 or x > 10) :  
    # Esegue se x è compreso tra 0 e 10, estremi  
    # inclusi, cioè esattamente come if 0 <= x <= 10:
```

```
if not(x < 0) or x > 10 :  
    # Esegue se x è maggiore o uguale a 0.  
    # Questa condizione ha poco senso, perché  
    # il secondo operando di or ha un insieme  
    # di verità che è un sottoinsieme di quello  
    # del primo operando: basta quello!  
    # Totalmente equivalente a if x >= 0:
```

# Commenti su più righe

- Quando vogliamo scrivere commenti "lunghi", che si estendano su più righe, è comodo usare una sintassi alternativa, rispetto a un `#` all'inizio di ogni riga

```
if not(x < 0) or x > 10 :
```

```
    """
```

```
        esegue se x è maggiore o uguale a 0  
        questa condizione ha poco senso, perché  
        il secondo operando di or ha un insieme  
        di verità che è un sottoinsieme di quello  
        del primo operando: basta quello!
```

```
        totalmente equivalente a if x >= 0:
```

```
        Dentro un commento come questo posso  
        scrivere qualsiasi cosa TRANNE tre  
        virgolette consecutive... ovviamente!
```

```
    """
```



# Valutazione in cortocircuito

```
if (x > 10 and x < 20) or x > 30 :
```

- ❑ La valutazione di un'espressione con operatori booleani viene effettuata con una strategia detta "**cortocircuito**"

*la valutazione dell'espressione termina appena è possibile decidere il risultato*

nel caso precedente, se **x** vale **15**, l'ultima condizione **non viene valutata**, perché sicuramente l'intera espressione vale **True**

Questo ha alcuni "effetti collaterali" interessanti...  
che vedremo più avanti!

Non c'è soltanto il vantaggio di "risparmiare tempo"



# Tavole della verità degli operatori booleani

| <i>A</i> | <i>B</i>         | <i>A and B</i> |
|----------|------------------|----------------|
| True     | True             | True           |
| True     | False            | False          |
| False    | <i>qualsiasi</i> | False          |

| <i>A</i> | <i>B</i>         | <i>A or B</i> |
|----------|------------------|---------------|
| True     | <i>qualsiasi</i> | True          |
| False    | True             | True          |
| False    | False            | False         |

**cortocircuito**

| <i>A</i> | <i>not A</i> |
|----------|--------------|
| True     | False        |
| False    | True         |

Ciascuna riga con "qualsiasi" sostituisce due righe della tabella

# Catene di operatori relazionali

- ❑ Come visto, in Python (diversamente da altri linguaggi di programmazione) si possono usare catene di operatori relazionali come siamo abituati a fare in matematica... **Cosa significa questo?**

```
if 10 <= x < 20 :  
    ...
```

- ❑ È equivalente a questo, ma qui l'espressione **x** viene **valutata due volte**, non una volta sola... la differenza può avere effetti se l'espressione contiene invocazioni di funzioni (pensate a **input**...)

```
if 10 <= x and x < 20 :  
    ...
```

- ❑ Ovviamente si possono scrivere condizioni anche più complesse e catene più lunghe, ma non esageriamo...

```
if a + b <= c / int(input()) < 20 < f :  
    ... # corpo eseguito se e solo se  
# a + b <= c/int(input()) and c/int(input()) < 20 and 20 < f  
# (ma con ciascuna espressione valutata una sola volta)
```

- ❑ E anche cose matematicamente "strane"...  
che EVITIAMO !

```
if a < b > c:  
    ... # corpo eseguito se e solo se  
# a < b and b > c
```

# Visualizzazione di valori booleani

- ❑ Abbiamo visto che la funzione **print** è in grado di visualizzare informazioni di tipo diverso
  - Stringhe
  - Valori numerici (interi o frazionari)
- ❑ Sarà in grado di visualizzare valori booleani? Se sì, come?

```
a = 3  
print(a > 2)
```

True

- ❑ Il codice appena visto è, dal punto di vista logico, equivalente a questo

- ❑ Il primo è decisamente preferibile
  - È più "elegante"...

```
a = 3  
if a > 2 :  
    print("True")  
else :  
    print("False")
```

- ❑ Quindi, la funzione **print** è in grado di visualizzare
  - Stringhe
  - Valori numerici (interi o frazionari)
  - Valori booleani
  - ...

Enunciati if annidati e  
alternative multiple

# Enunciati **if** annidati

Attenzione  
all'indentazione!

❑ Un dubbio legittimo... si può mettere un enunciato **if** nel corpo di un altro enunciato **if** o di una clausola **else** ???

❑ La risposta è semplice, basta ricordare la sintassi

- Cosa può costituire il corpo di un **if** o di una clausola **else** ?
  - **Un enunciato! (senza nessuno specifico "divieto"...)**

❑ Quando un **if** si trova all'interno del corpo di un altro **if** o di una clausola **else** si parla di "**if annidato**" (*nested*, in inglese)

- **Non è una costruzione particolare:** semplicemente, è una delle possibilità previste dalle regole sintattiche e, anzi, come vedremo, è molto utilizzata

```
if c > 10 :  
    if a + b < 20 :  
        print("YYY")  
    else :  
        print("ZZZ")  
else : # cioè c <= 10  
    print("XXX")
```

È un esempio incompleto!

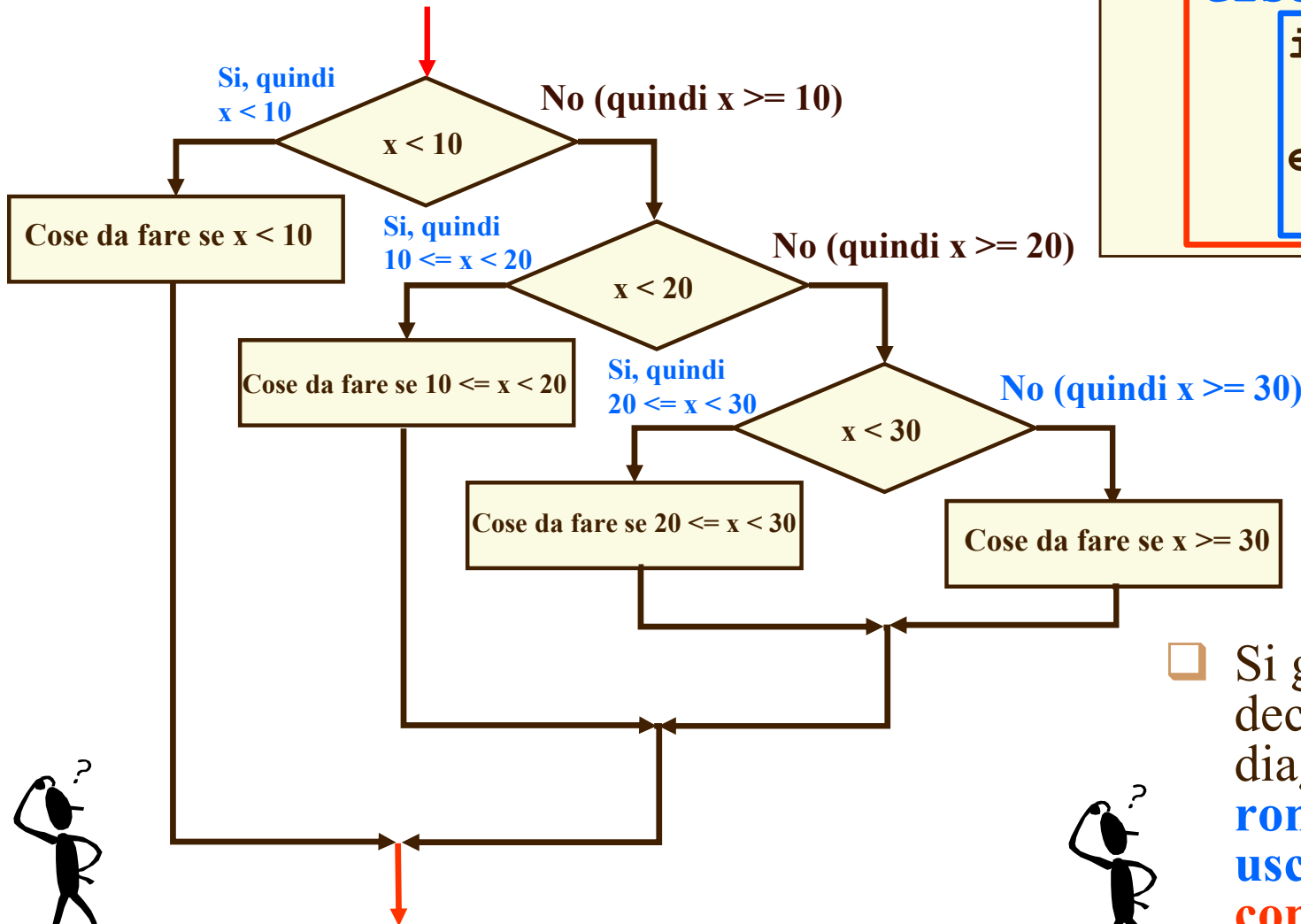
Devono, ovviamente, esserci righe precedenti che assegnino un valore alle variabili

# Alternative multiple

- Usando in modo opportuno gli **if** annidati, si possono realizzare **alternative multiple**

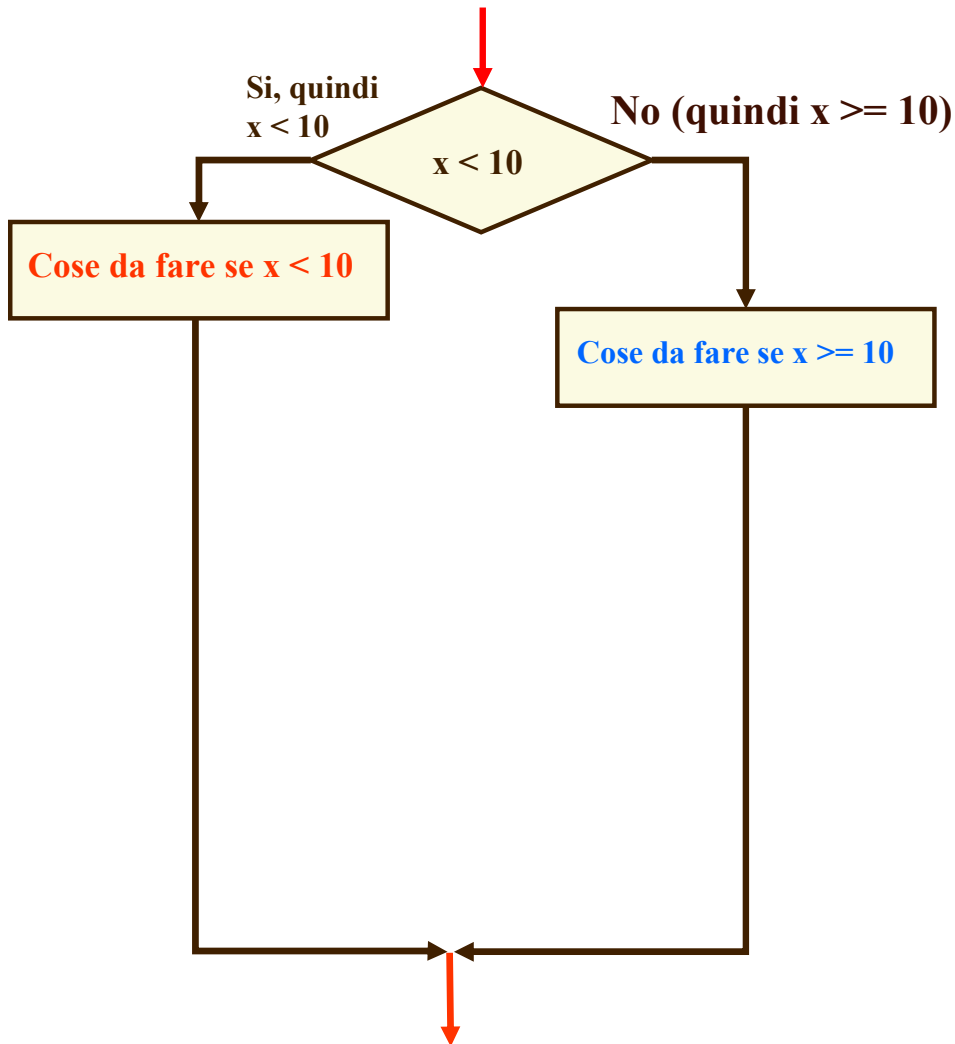
```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ...# 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ...# 20 <= x < 30  
        else : # x >= 30  
            ...
```

Qui sopra c'è un **unico** enunciato (composito...)



- Si genera una struttura decisionale equivalente, nei diagrammi di flusso, a **un rombo con più di due uscite** (e relative etichette), **come vedremo meglio nelle slide successive**

# Alternative multiple

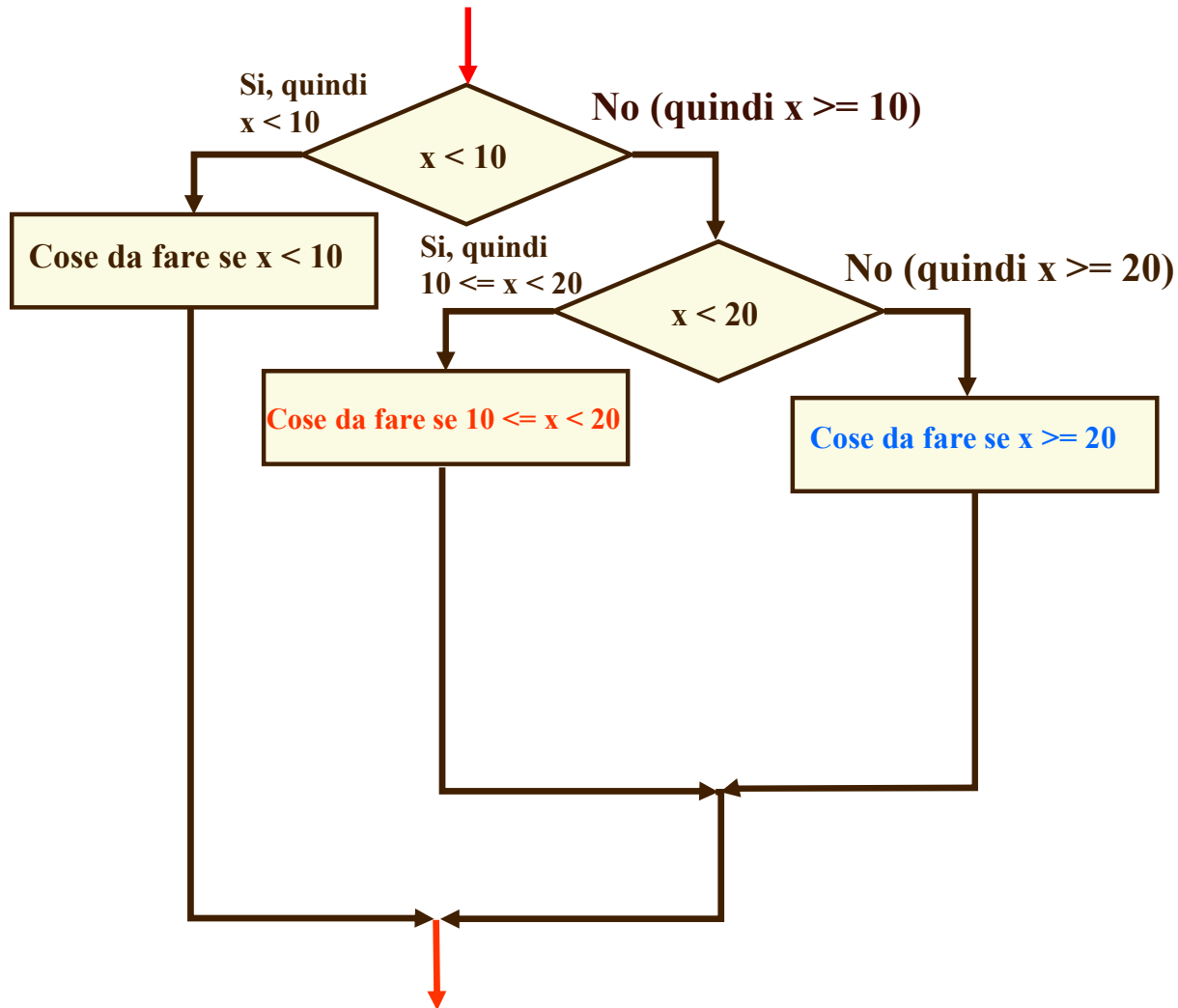


```
if x < 10 :  
    ... # x < 10  
else : # x >= 10  
    if x < 20 :  
        ...# 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20<=x<30  
        else : # x >= 30  
            ...
```

**Continua...**



# Alternative multiple

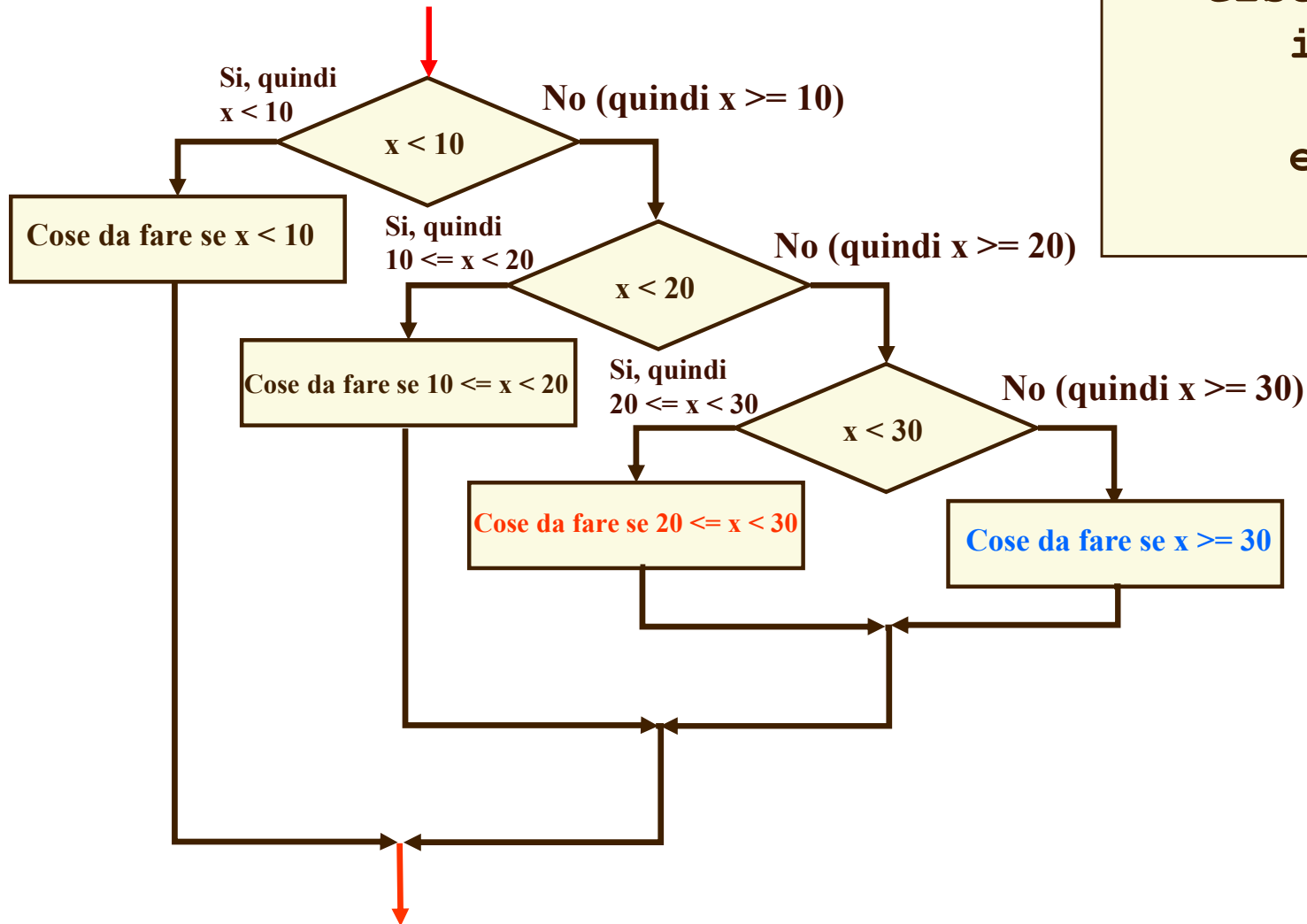


```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ...
```

Continua...

# Alternative multiple

```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ... # x >= 30
```

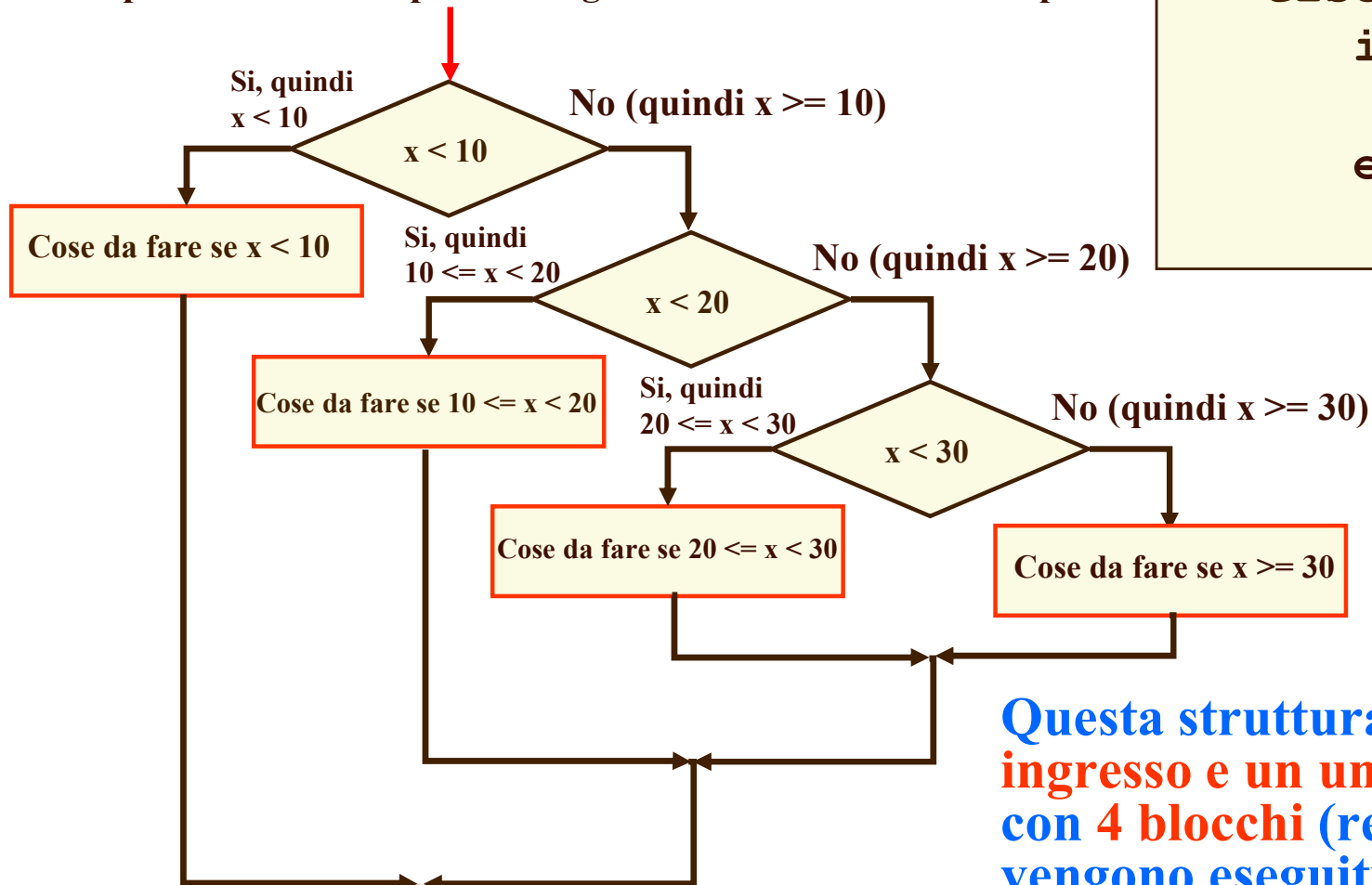


**Finito**

# Alternative multiple

```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ... # x >= 30
```

Dal passato... **unico** punto d'ingresso nell'alternativa multipla



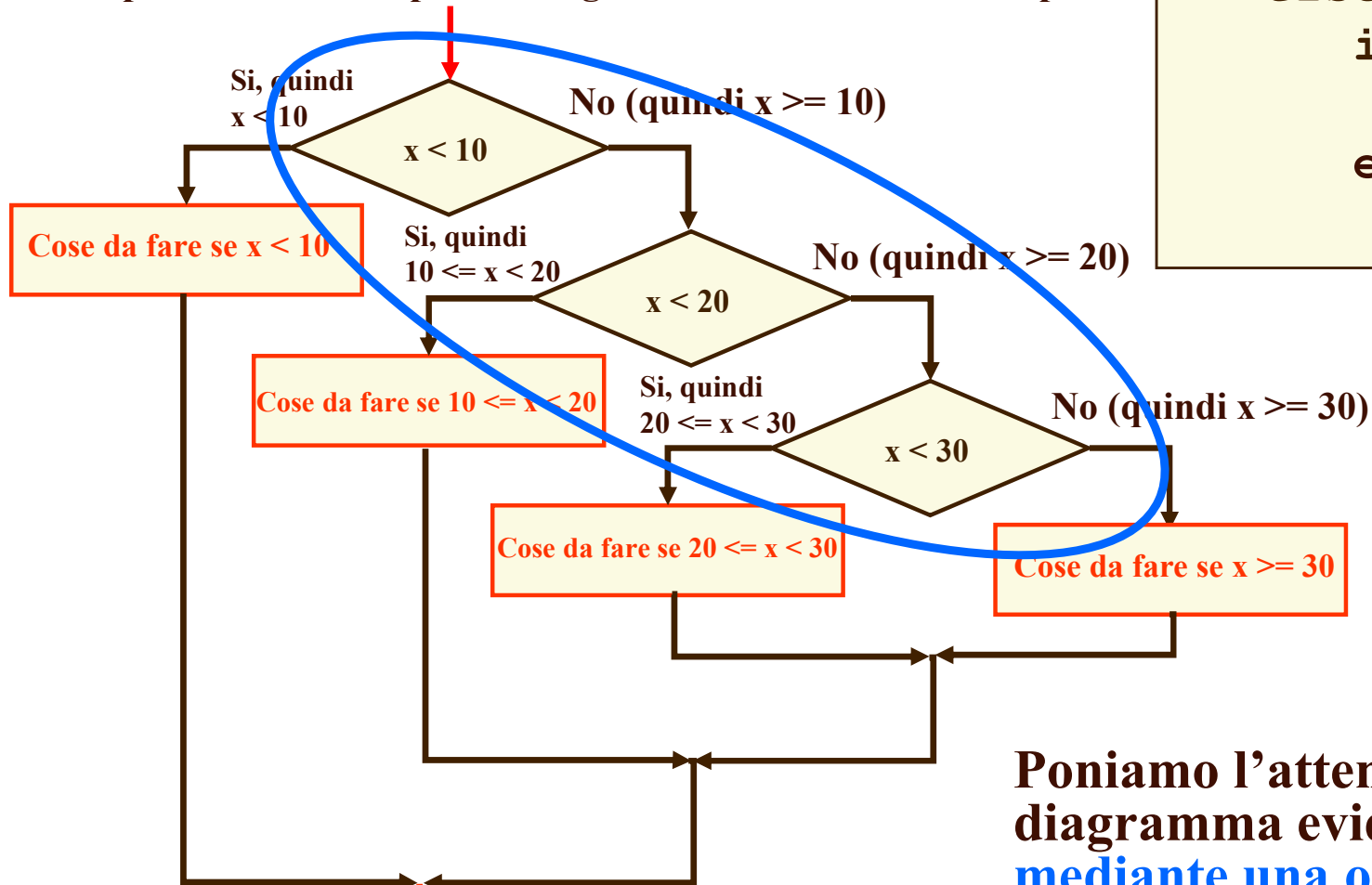
Verso il futuro...  
**Unico** punto d'uscita dall'alternativa multipla!

**Questa struttura ha un unico punto di ingresso e un unico punto di uscita, con 4 blocchi (rettangolari) che vengono eseguiti IN ALTERNATIVA: ne viene sempre eseguito uno e soltanto uno, per poter passare "dal passato al futuro" dell'esecuzione**

# Alternative multiple

```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ... # x >= 30
```

Dal passato... **unico** punto d'ingresso nell'alternativa multipla

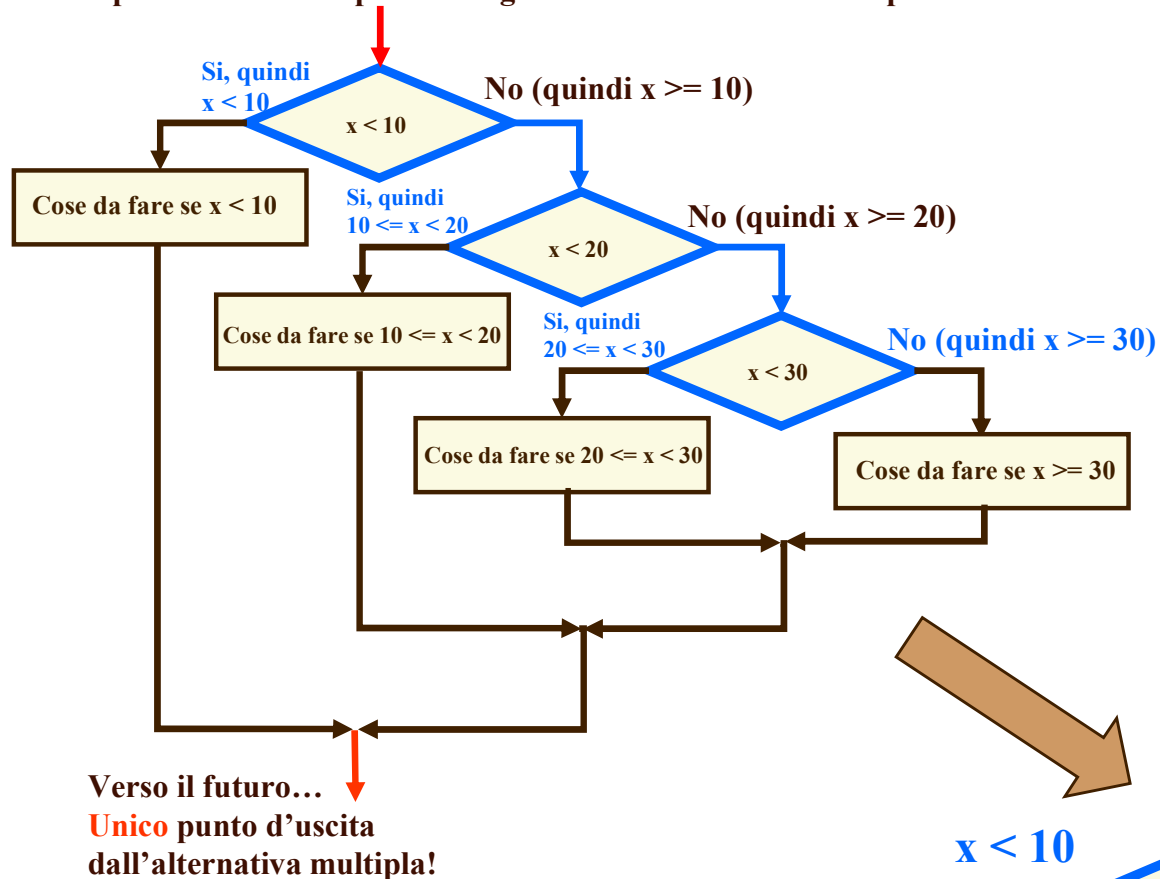


Verso il futuro...

**Unico** punto d'uscita dall'alternativa multipla!

Poniamo l'attenzione sulla zona di diagramma evidenziata, che, **mediante una o più scelte**, guida il percorso dall'alto in basso attraverso **uno (e uno solo) dei blocchi** eseguibili

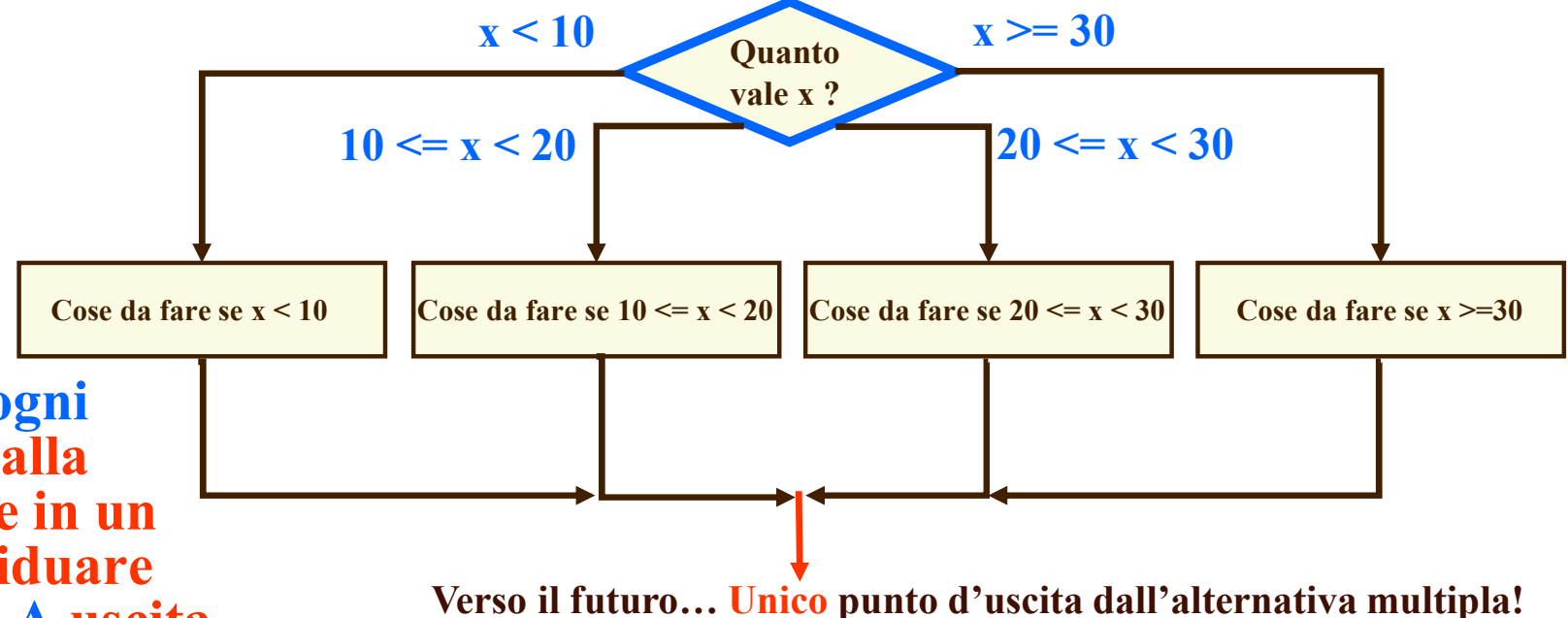
Dal passato... **unico** punto d'ingresso nell'alternativa multipla



```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ...# 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20<=x<30  
        else : # x >= 30  
            ...
```

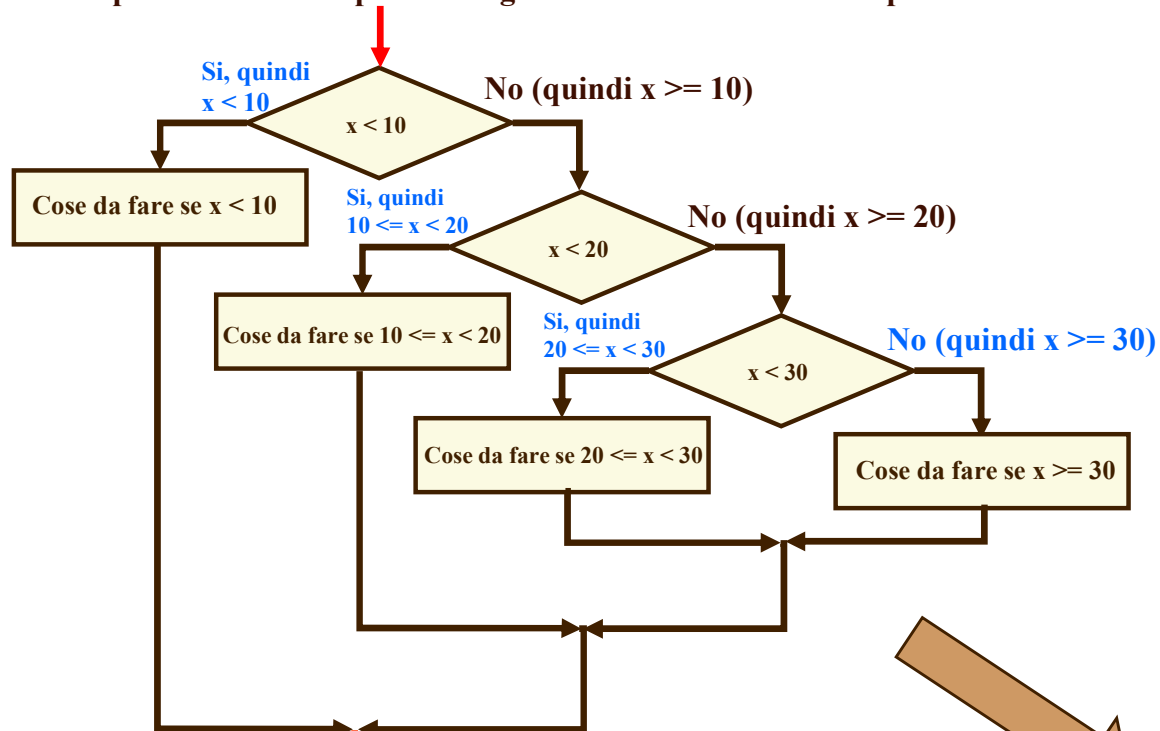
Questa struttura decisionale equivale, nei diagrammi di flusso, a un rombo con più di due uscite (e **relative etichette**)

Dal passato... **unico** punto d'ingresso nell'alternativa multipla



**ATTENZIONE:** ogni possibile risposta alla domanda presente in un rombo deve individuare **UNA E UNA SOLA uscita**

Dal passato... **unico** punto d'ingresso nell'alternativa multipla



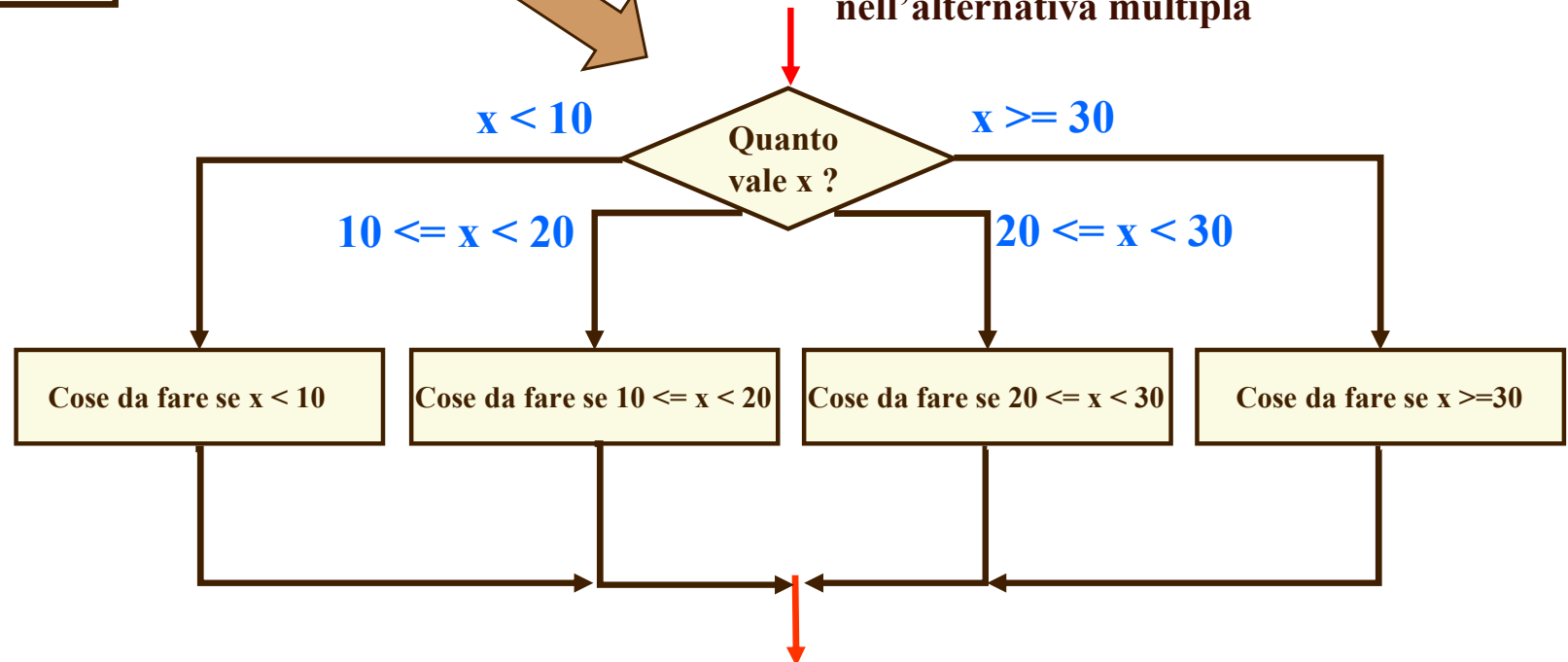
Verso il futuro...  
**Unico** punto d'uscita  
dall'alternativa multipla!

**Purtroppo non esiste un elemento sintattico per rappresentare un rombo con più di due uscite, bisogna usare *if* annidati**

```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ...
```

**Struttura equivalente decisamente più semplice da disegnare, da leggere e da capire**

Dal passato... **unico** punto d'ingresso  
nell'alternativa multipla

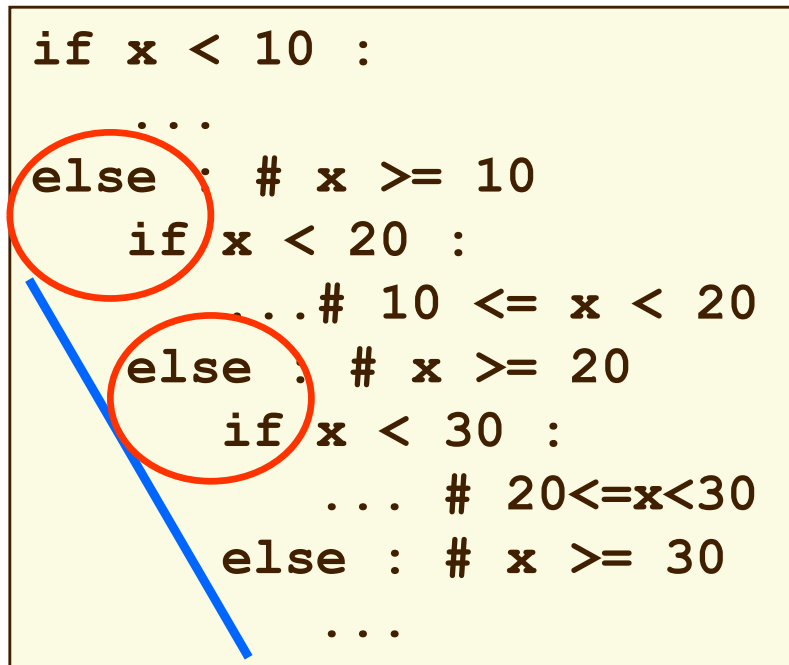


Verso il futuro... **Unico** punto d'uscita dall'alternativa multipla!

# Clausola `elif`

- ❑ Come abbiamo visto, usando in modo opportuno gli `if` annidati, si possono realizzare alternative multiple
  - Nel diagramma di flusso, si genera una struttura decisionale equivalente a un rombo con più di due uscite (e relative etichette)

```
if x < 10 :  
    ...  
else : # x >= 10  
    if x < 20 :  
        ... # 10 <= x < 20  
    else : # x >= 20  
        if x < 30 :  
            ... # 20 <= x < 30  
        else : # x >= 30  
            ...
```



- ❑ Se le alternative sono molte, il codice "scivola" rapidamente verso destra
- ❑ Per ovviare a questo e rendere il codice più chiaro, Python mette a disposizione la clausola `elif` (che sta per `else if`), che va indentata come `if` a cui appartiene

```
if x < 10 :      # x < 10  
    ...  
elif x < 20 :   # 10 <= x < 20  
    ...  
elif x < 30 :   # 20 <= x < 30  
    ...  
else :          # 30 <= x  
    ...
```

- ❑ La clausola conclusiva, `else`, è anche in questo caso facoltativa (dipende dalla logica che si vuole realizzare)

# Alternative multiple

- ❑ Nel progettare alternative multiple (con **elif** o con **if** annidati), bisogna fare attenzione all'ordine in cui vengono specificate le condizioni

```
if x < 10 :      # x < 10
    ...
elif x < 30 :    # 10 <= x < 30
    ...
elif x < 20 :
    ... # non succederà MAI !
else :          # 30 <= x
    ...
```

```
if x < 10 :
    ...
elif 10 <= x < 20 :
    ...
elif 20 <= x < 30 :
    ...
elif x >= 30 :
    ...
```

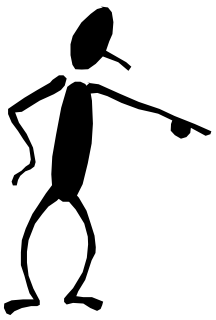
- ❑ Ricordiamo che in un enunciato **if/elif/elif/.../else**, uno e uno solo dei corpi viene eseguito!
- ❑ Se **x** è minore di 20 (ma non di 10), la condizione del **primo elif** sarà vera e **nessun'altra condizione seguente verrà valutata**
- ❑ Se la condizione del primo **elif** è falsa, sarà sempre falsa anche la condizione del secondo **elif**, quindi **il corpo del secondo elif non viene MAI eseguito**, c'è un errore logico oppure ho scritto del codice inutile (che, comunque, è sempre un errore)

Scrivere **così** è **lecito** (e apparentemente più chiaro) **ma sconsigliato**: si appesantisce il codice e si rallenta l'esecuzione, inoltre chi legge si chiederà perché siano state scritte **condizioni totalmente inutili** (perché **sempre vere** nel momento in cui vengono valutate)



Il materiale necessario per il  
Laboratorio 02  
termina qui

**Lezione 11**  
**23/10/2024**  
**ore 10.30-12.30**  
**aula Ve**



# Enunciato if definitivo

## ❑ Sintassi

```
if condizione :  
    enunciato  
clausole elif facoltative  
clausola else facoltativa
```

- ❑ Il corpo dell'**if** (cioè **enunciato**) viene eseguito se e solo se l'espressione booleana che rappresenta la **condizione** assume il valore **True**
- ❑ Se, invece, la **condizione** assume il valore **False**, vengono valutate, nell'ordine, le eventuali clausole **elif**
  - Se la **condizione** assume il valore **True**, viene eseguito l'**enunciato** e non vengono valutate ulteriori eventuali clausole, né **elif** né **else**
- ❑ Se non è stato eseguito alcun enunciato, viene eseguito il corpo dell'eventuale clausola **else**, se presente

```
elif condizione :  
    enunciato
```

```
else :  
    enunciato
```

Costanti

# L'uso delle costanti

- Un programma per la conversione di valuta

```
dollars = float(input("Quanti dollari? "))  
euros = dollars * 0.87  
print(dollars, "dollari =", euros, "euro")
```

- Chi legge il programma potrebbe legittimamente chiedersi quale sia il significato del “*numero magico*” **0.87** usato nel programma per convertire i dollari in euro...

# L'uso delle costanti

- ❑ Così come si usano nomi simbolici descrittivi per le variabili, è opportuno assegnare *nomi descrittivi* anche ai valori *costanti* utilizzati nei programmi

- Si parla di "variabili costanti" (una specie di ossimoro) o semplicemente **costanti**

```
EURO_PER_DOLLAR = 0.87
dollars = float(input("Quanti dollari? "))
euros = dollars * EURO_PER_DOLLAR
print(dollars, "dollari =", euros, "euro")
```

- ❑ Un primo *vantaggio* molto importante

*migliora la leggibilità*

- ❑ Di solito in Python si usa la seguente **convenzione**

- *i nomi di costanti sono formati da lettere maiuscole*
  - i nomi composti si ottengono collegando le parole successive alla prima mediante un *carattere di sottolineatura*

# L'uso delle costanti

- ❑ Un *altro grande vantaggio*: se il valore della costante deve cambiare, in una nuova versione del programma, la modifica va fatta *in un solo punto* del codice!

```
EURO_PER_DOLLAR = 0.93
dollars = float(input("Quanti dollari? "))
euros = dollars * EURO_PER_DOLLAR
print(dollars, "dollari =", euros, "euro")
dollars = float(input("Quanti altri dollari? "))
euros = dollars * EURO_PER_DOLLAR
print(dollars, "dollari =", euros, "euro")
```

# L'uso delle costanti

- ❑ Quando **la logica del programma** prevede che il valore di una variabile non venga MAI aggiornato durante l'esecuzione, meglio assegnarle un nome di costante (cioè tutto maiuscolo)
- ❑ In questo modo, se tale variabile sarà oggetto di un enunciato di assegnazione, l'errore può essere evidenziato proprio dal nome maiuscolo
  - **Ogni volta che assegniamo un nuovo valore a una variabile che ha il nome interamente maiuscolo, dobbiamo chiederci se stiamo facendo una cosa sensata...**
- ❑ **In altri linguaggi di programmazione la "protezione" del valore di una costante è molto più rigida e garantita dal compilatore/interprete, in Python (purtroppo) non è così**
  - In Python, assegnare un nuovo valore a una (variabile) costante **NON** è un errore di sintassi né di esecuzione, anche se molto probabilmente sarà un errore logico



# Operatori relazionali e stringhe


# Confronto di stringhe

- ❑ Gli operatori relazionali di **uguaglianza e diversità** funzionano correttamente anche per confrontare stringhe

```
s1 = "Marcello"
s2 = "Marce" + "l" + "lo"
# il valore di s2 è "Marcello"
s3 = "Dalpasso"
if s1 == s2 : # è vero
    ... # il corpo verrà eseguito
if s1 != s3 : # è vero
    ... # il corpo verrà eseguito
```

- ❑ **Due stringhe sono uguali se e solo se hanno la stessa lunghezza e contengono caratteri identici in posizioni corrispondenti (z e Z sono caratteri diversi...)**

# Confronto di stringhe

- ❑ E gli altri operatori relazionali? Che senso ha chiedersi se **una stringa "è minore" di un'altra stringa?**
  - ❑ È una domanda molto sensata, pensiamo ad archivi come i dizionari... le parole (che per noi sono stringhe) sono memorizzate in **ordine alfabetico** e ciascuna "è minore" di quella seguente
  - ❑ L'ordinamento alfabetico, però, non è sufficiente, perché le stringhe, in Python, possono contenere qualsiasi carattere, mentre l'ordinamento alfabetico riguarda stringhe che contengono soltanto lettere (cioè caratteri appartenenti a un alfabeto...)
-  Si definisce l'**ordinamento lessicografico** come estensione dell'ordinamento alfabetico

# Confronto alfabetico

- Partendo dall'inizio delle stringhe, si confrontano a due a due i caratteri in posizioni corrispondenti, finché **una delle stringhe termina** oppure **due caratteri sono diversi**
  - se **una stringa termina**, essa precede l'altra
    - se terminano contemporaneamente, sono uguali

|   |   |   |   |   |
|---|---|---|---|---|
| c | a | s | t | a |
|---|---|---|---|---|

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| c | a | s | t | a | n | o |
|---|---|---|---|---|---|---|



lettere  
uguali

**casta**  
*precede*  
**castano**

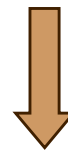
# Confronto alfabetico

- Partendo dall'inizio delle stringhe, si confrontano a due a due i caratteri in posizioni corrispondenti, finché **una delle stringhe termina** oppure **due caratteri sono diversi**
  - se una stringa termina, essa precede l'altra
    - se terminano contemporaneamente, sono uguali
  - **altrimenti**, l'ordinamento tra le due stringhe è uguale all'ordinamento **alfabetico** tra i primi **due caratteri diversi** (indipendentemente da quali siano gli eventuali caratteri successivi)

|   |   |   |   |   |
|---|---|---|---|---|
| c | a | r | t | e |
|---|---|---|---|---|

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| c | a | s | t | a | n | o |
|---|---|---|---|---|---|---|

lettere  
uguali



|             |
|-------------|
| r precede s |
|-------------|



**carte**  
*precede*  
**castano**

# Confronto lessicografico

- ❑ Il confronto lessicografico induce un ordinamento *simile* a quello alfabetico di un comune dizionario
- ❑ L'algoritmo è identico, l'unica cosa che serve è definire un "ordinamento lessicografico tra singoli caratteri", che sostituisca quello alfabetico



- L'ordinamento lessicografico tra singoli caratteri è definito dallo standard **Unicode**,  
<http://www.unicode.org>

- ❑ L'ordinamento tra singoli caratteri in Python è, quindi, *simile* all'ordinamento alfabetico, con qualche differenza... perché tra i caratteri non ci sono solo le lettere! Ad esempio
  - le cifre numeriche precedono le lettere
  - tutte le lettere maiuscole precedono tutte le lettere minuscole
  - il carattere di “spazio” (*blank*) precede tutti gli altri caratteri
  - ...

# Confronto lessicografico di stringhe

- ❑ Usando gli operatori relazionali  $<$ ,  $\leq$ ,  $>$  e  $\geq$  tra stringhe, si ottengono risultati che dipendono dall'ordinamento lessicografico
- ❑ La stringa X **"è minore"** della stringa Y se e solo se X **precede** Y nell'ordinamento lessicografico
- ❑ Ovviamente,  
X "è maggiore" di Y se e solo se Y "è minore" di X

Singoli caratteri  
di una stringa



# Un carattere di una stringa

- ❑ Se ci interessa elaborare **un solo carattere** appartenente a una stringa e ne conosciamo la posizione (ad esempio, il primo o il terzo carattere), possiamo **generare una nuova stringa** a partire da quella, **copiandone** una porzione di lunghezza unitaria
  - Si usa **una sintassi specifica**, con le **parentesi quadre**, indicando al loro interno la posizione del carattere che ci interessa
  - La posizione viene anche chiamata **indice** (un termine tecnico in informatica)

## ❑ Le posizioni sono numerate a partire da zero !!!

```
name = input("Come ti chiami? ")
s = name[0]
print("Il tuo nome inizia con la lettera " + s)
```

```
Come ti chiami? Marcello
Il tuo nome inizia con la lettera M
```

```
# alternativa: cosa c'è di diverso? output identico...
print("Il tuo nome inizia con la lettera", s)
```

# Un carattere di una stringa

- ❑ Le posizioni sono numerate **ANCHE** a partire da **-1**, procedendo dalla fine verso l'inizio

```
name = input("Come ti chiami? ")
s = name[-1] # equivale a s = name[len(name) - 1]
print("Il tuo nome termina con la lettera " + s)
print("e la sua penultima lettera è " + name[-2])
```

```
Come ti chiami? Marcello
Il tuo nome termina con la lettera o
e la sua penultima lettera è l
```

- ❑ Tecnicamente, quando si usa un **indice negativo**, questo viene interpretato da Python come se fosse uguale a "lunghezza della stringa – valore assoluto dell'indice"
  - Infatti l'ultimo carattere della stringa **s** ha (sempre) indice **len(s) - 1**, oppure soltanto **-1**

# Un carattere di una stringa

- ❑ Se si usa un indice che rappresenta una posizione che non è presente nella stringa, si verifica un **errore in esecuzione** (non di sintassi),

```
s = "01234"  
print(s[5])
```

perché è un'operazione non eseguibile  
(come una divisione per zero...)

```
s = "01234"  
print(s[-6])
```

```
IndexError: string index out of range
```

- ❑ (Ovviamente?)

**In una stringa vuota non esiste alcun indice valido**

- ❑ Dentro alle parentesi quadre deve esserci un valore numerico intero (appartenente all'intervallo opportuno), non necessariamente un valore letterale!

Quindi, può essere qualsiasi

**espressione numerica**

**intera** (con valore opportuno)

```
s = "43210"  
a = 2  
b = 1  
print(s[a + b]) # visualizza 1
```

# Impaginazione del codice

# Istruzioni su più righe

- ❑ È bene evitare di scrivere enunciati troppo lunghi, che possono costringere il lettore a far scorrere la finestra dell'editor verso destra
- ❑ Abbiamo detto fin dall'inizio che "una riga = un enunciato", in realtà **si può andare a capo**, inserendo un carattere **\** (*backslash*, o "barra rovesciata") alla fine della riga, prima di andare a capo
  - La parte di riga che si scrive a capo va un po' indentata "per chiarezza"

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c)) \  
      / (2 * a)
```

- ❑ **Dopo il *backslash* non ci deve essere NESSUN carattere, neanche uno spazio**

# Istruzioni su più righe

NON USATE  
QUESTO METODO

- ❑ È bene evitare di scrivere enunciati troppo lunghi, che possono costringere il lettore a far scorrere la finestra dell'editor verso destra
- ❑ In alternativa, si può andare a capo senza il *backslash* a fine riga **se si lascia una parentesi aperta e non chiusa**

```
x1 = ( (-b + sqrt(b ** 2 - 4 * a * c))  
      / (2 * a) )
```

- ❑ Così invece non va bene, perché la prima riga è un enunciato completo e come tale viene considerato dall'interprete Python, che poi segnala un errore sulla seconda riga, che non è un enunciato valido

```
x1 = (-b + sqrt(b ** 2 - 4 * a * c))  
      / (2 * a)
```

- ❑ Eventualmente si può aggiungere una coppia di parentesi che racchiuda l'intera espressione! Si può sempre fare...

# Righe vuote nel codice

- ❑ Ora che i programmi diventano più lunghi e articolati... possiamo **introdurre righe vuote nel codice** per suddividere in modo evidente le diverse sezioni del programma
  - Acquisizione e validazione dei dati in input
  - Inizializzazioni varie
  - Elaborazione principale
  - Visualizzazione dei risultati
- ❑ Spesso risulta utile iniziare ciascuna "sezione" con una riga di commento che ne illustri le finalità

**Lezione 12**  
**25/10/2024**  
**ore 16.30-18.30**  
**aula Ve**



Cicli  
(o iterazioni)

# Problema

- ❑ Riprendiamo un problema visto in precedenza, per il quale *abbiamo individuato un algoritmo* senza averlo ancora realizzato in un programma Python

**Problema:** Avendo depositato ventimila euro in un conto bancario che produce il 5% di interessi all'anno, capitalizzati annualmente, quanti anni occorrono affinché il saldo del conto arrivi al doppio della cifra iniziale?

# Algoritmo che risolve il problema

- 1 All'anno 0 il saldo è 20000
- 2 **Ripetere** i passi **3** e **4** **finché** il saldo è minore del doppio di 20000, poi passare al punto **5**
- 3 Aggiungere 1 al valore dell'anno corrente
- 4 Il nuovo saldo è il valore del saldo precedente moltiplicato per 1.05 (cioè aggiungiamo il 5%)
- 5 Il risultato è il valore dell'anno corrente

Dimostrando la correttezza dell'algoritmo, abbiamo visto che **la ripetizione richiesta al punto 2 può avvenire al massimo per 20 volte** (in realtà la ripetizione avviene 15 volte, ma questo è il risultato che deve essere calcolato dal programma, non lo sappiamo a priori)

# Programma che risolve il problema

```
year = 0          # Punto 1
balance = 20000   # Punto 1
if balance < (2 * 20000) : # Punto 2 per la prima volta
    year += 1      # Punto 3 per la prima volta
    balance *= 1.05 # Punto 4 per la prima volta
if balance < (2 * 20000) : # Punto 2 per la seconda volta
    year += 1      # Punto 3 per la seconda volta
    balance *= 1.05 # Punto 4 per la seconda volta
if balance < (2 * 20000) : # Punto 2 per la terza volta
    year += 1      # Punto 3 per la terza volta
    balance *= 1.05 # Punto 4 per la terza volta
... # le stesse tre righe ripetute...
if balance < (2 * 20000) : # Punto 2 per la ventesima volta
    year += 1              # Punto 3 per la ventesima volta
    balance *= 1.05        # Punto 4 per la ventesima volta
print(year, "anni") # Punto 5
```

- ❑ Nei primi 15 enunciati **if** la condizione sarà vera, nei successivi (fino al ventesimo) la condizione sarà falsa
- ❑ "Piccolo" problema di questa soluzione: se il tasso d'interesse fosse 1%, dovremmo scrivere 100 **if**, **tutti identici**, e solo i primi 70 sarebbero utili...
  - Speriamo che ci sia una soluzione **sintatticamente migliore!!**

# Algoritmo che risolve il problema

- 1 All'anno 0 il saldo è 20000
- 2 **Ripetere** i passi **3** e **4** **finché** il saldo è minore del doppio di 20000, poi passare al punto **5**
- 3 Aggiungere 1 al valore dell'anno corrente
- 4 Il nuovo saldo è il valore del saldo precedente moltiplicato per 1.05 (cioè aggiungiamo il 5%)
- 5 Il risultato è il valore dell'anno corrente

In Python, l'enunciato **while** consente la realizzazione di programmi che devono

- ***eseguire ripetutamente una serie di azioni sempre uguali** (in questo caso, i passi 3 e 4) **finché** è verificata una specifica condizione (espressa, in questo caso, al passo 2)*

# Soluzione con while

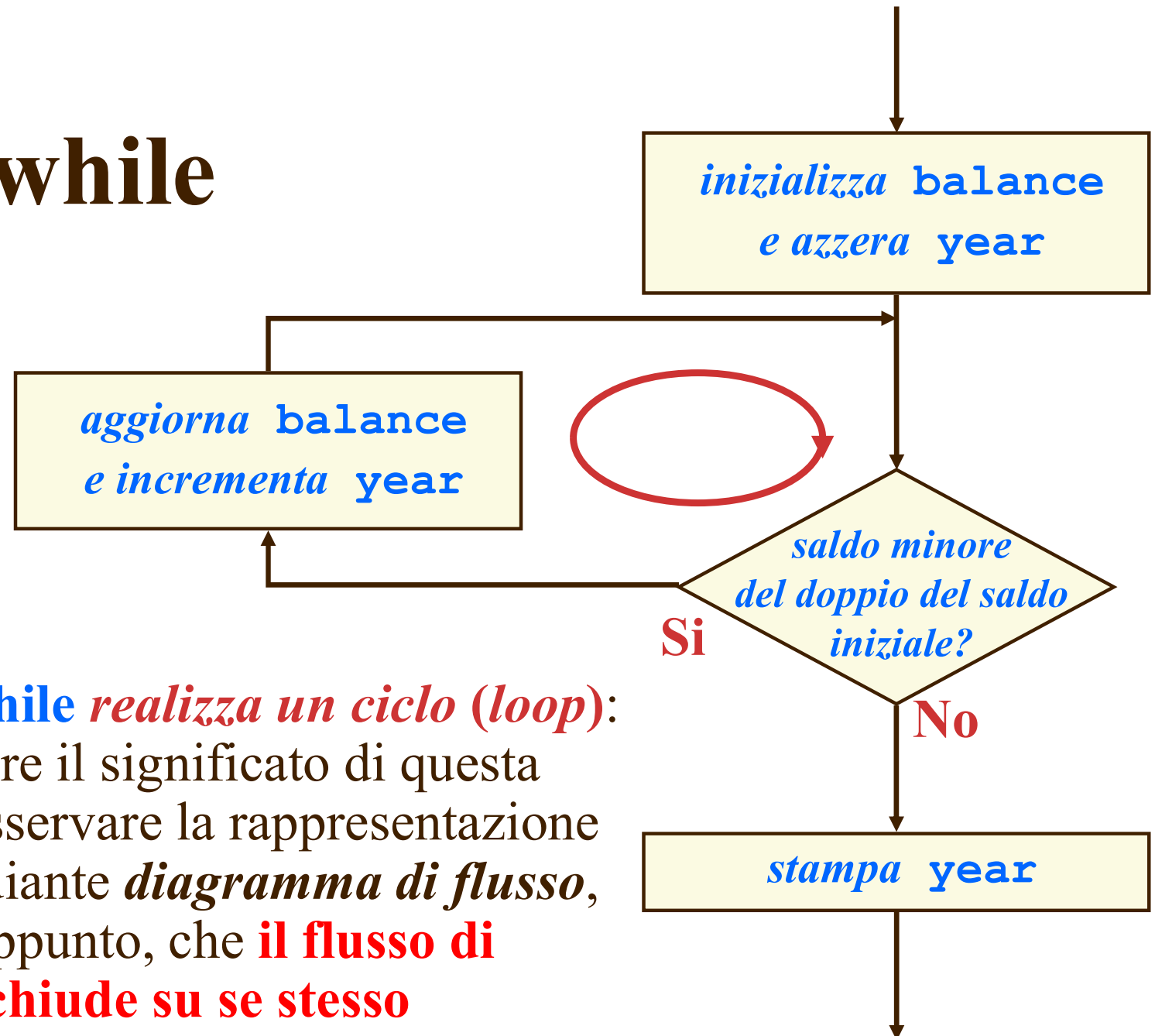
```
year = 0
balance = 20000
while balance < 2 * 20000 : # sintassi come if, ma while!
    year += 1
    balance *= 1.05
print(year, "anni")
```

```
year = 0
balance = 20000
if balance < (2 * 20000) :
    year += 1
    balance *= 1.05
... # 20 if identici!
print(year, "anni")
```

## Un po' più utile...

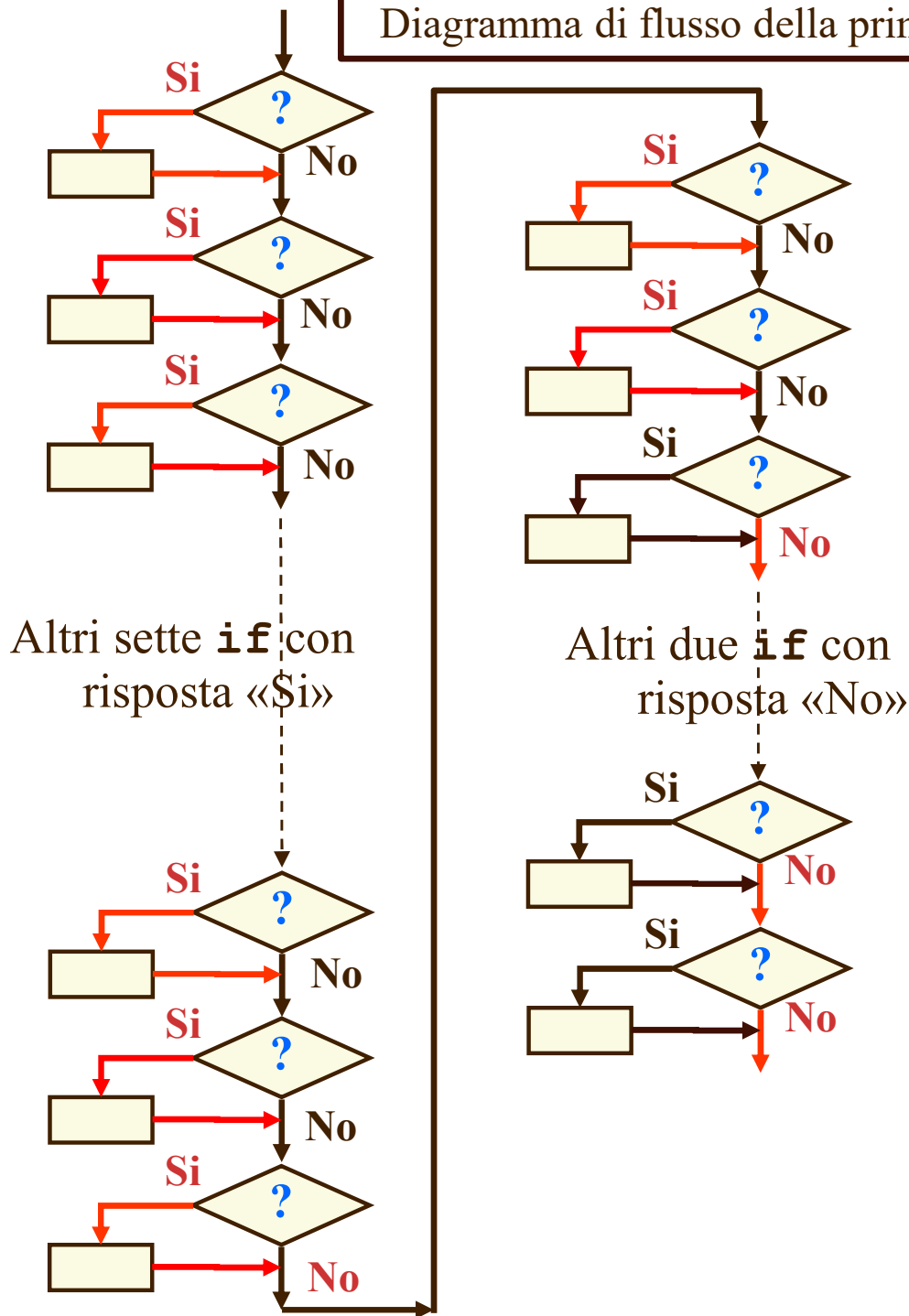
```
INITIAL_BALANCE = float(input("Saldo da raddoppiare? "))
RATE = float(input("Tasso di interesse percentuale? "))
balance = INITIAL_BALANCE
FINAL_BALANCE = 2 * INITIAL_BALANCE
year = 0
while balance < FINAL_BALANCE :
    interest = balance * RATE / 100
    balance += interest # interest eliminabile
    year += 1
print(year, "anni")
```

# Il ciclo while



L'enunciato **while** *realizza un ciclo (loop)*: per comprendere il significato di questa frase, è utile osservare la rappresentazione del codice mediante *diagramma di flusso*, dove si nota, appunto, che **il flusso di esecuzione si chiude su se stesso**

# Diagramma di flusso della prima versione, senza ciclo

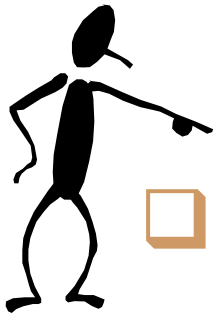


La domanda nel rombo è sempre  
"il saldo è minore del doppio  
del saldo iniziale?"

Il blocco da eseguire è sempre  
"aggiorna saldo e anno"

Il flusso d'esecuzione è  
*domanda, risposta Si, aggiornamento,*  
poi ancora  
*domanda, risposta Si, aggiornamento,*  
per 15 volte consecutive, quindi  
*domanda, risposta No,*  
*domanda, risposta No,*  
per 5 volte consecutive,  
esattamente come nel ciclo  
(dove si risparmiano le ultime 4  
domande, perché, dopo aver ottenuto la  
prima *risposta No*, non vengono più  
fatte domande)





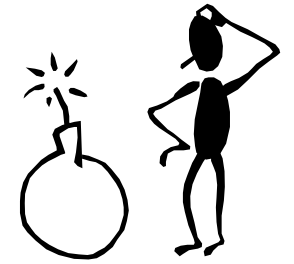
# L'enunciato while

□ Sintassi:

```
while condizione :  
    enunciato
```

- Scopo: eseguire un *enunciato* (detto “corpo”) finché la *condizione* è vera
  - La *condizione* è un'espressione booleana (esattamente come nell'enunciato **if**) e viene spesso chiamata "condizione di **continuazione**" (del ciclo)
    - La *condizione* viene (ri)valutata prima dell'esecuzione di ciascuna iterazione del ciclo: finché è vera, il ciclo continua
- Nota: il *corpo* del ciclo **while** può essere un enunciato **qualsiasi**, quindi può anche essere un blocco di enunciati (e sarebbe inutile dirlo...) e può anche contenere un altro enunciato **while** annidato (e sarebbe inutile dirlo...)
  - Esattamente come nell'enunciato **if**

# Ciclo infinito



❑ Esistono errori logici che *impediscono la terminazione di un ciclo*, dando luogo a un *ciclo infinito*

- **L'esecuzione del programma continua ininterrottamente**

- Bisogna arrestare il programma con un comando del sistema operativo (**Ctrl-C** in Windows e Linux), oppure addirittura riavviare il computer ☹ [MAI in Aula Taliercio!]

- **Problema con idle... a volte bisogna usare `xkill` (vedere il sito del corso)**

```
balance = 20000 # problema diverso dal precedente
RATE = 5
year = 0
while year < 20 : # quale saldo dopo 20 anni ?
    balance = balance + balance * RATE / 100
    # qui mi sono dimenticato year = year + 1 !!!
print(balance) # non viene mai eseguito!!
```

❑ **Non sono rilevati dall'interprete!!!**

# Ciclo a contatore o a evento

- Si parla di *ciclo a contatore* quando la condizione di continuazione è controllata da una variabile che assume il ruolo di un **contatore**, cioè è una variabile numerica intera che viene incrementata o decrementata di un'unità a ogni iterazione del ciclo

```
balance = 20000
RATE = 5
year = 0 # valore iniziale del contatore
while year < 20 : # verifico il contatore
    balance = balance + (balance * RATE / 100)
    year = year + 1 # aggiorno il contatore
```

- In generale, invece, si parla invece di *ciclo a evento*

```
INITIAL_BALANCE = 20000
RATE = 5
balance = INITIAL_BALANCE
year = 0
while balance < 2 * INITIAL_BALANCE :
    balance = balance + balance * RATE / 100
    year = year + 1 # year è un contatore ma
                    # NON controlla il ciclo
```

L'evento (che termina il ciclo) è il fatto che la **condizione** diventi falsa

Seguire passo dopo passo  
l'esecuzione di un ciclo  
(*tracing* o tracciamento)

# Seguire l'esecuzione di un ciclo

- ❑ Per comprendere il funzionamento di un ciclo e/o per verificare di averlo progettato correttamente, è spesso utile (se non indispensabile...) **seguire la sua esecuzione passo dopo passo, confrontandola con quanto previsto "a mano"**
- ❑ Per farlo, si possono **inserire degli enunciati `print`** all'interno del corpo del ciclo, per poi rimuoverli (o commentarli)

```
INITIAL_BALANCE = 20000
RATE = 5
balance = INITIAL_BALANCE
year = 0
while balance < 2 * INITIAL_BALANCE :
    balance = balance + balance * RATE / 100
    year = year + 1
    print("DEBUG: anno =", year, "saldo =", balance)
```

Rileggere questa  
slide OGNI DUE  
GIORNI 😊

- ❑ **Poi, OVVIAMENTE, bisogna verificare i calcoli in modo indipendente dal programma**

**Acquisire una  
sequenza di dati**

# Acquisire una sequenza di dati

- ❑ Molti problemi di elaborazione richiedono la *lettura di una sequenza di dati in ingresso*
- ❑ Esempio: calcolare la somma di **DIECI** numeri forniti dall'utente

```
print("Scrivi 10 numeri, uno per riga")
sum = 0
count = 0
while count < 10 : # ciclo a contatore
    sum = sum + float(input())
    count = count + 1
print("Somma:", sum)
```

# Acquisire una sequenza di dati

- ❑ Per rendere il programma più flessibile, si può **chiedere all'utente la lunghezza della sequenza**

```
NUM = int(input("Quanti numeri? "))
print("Scrivi i numeri, uno per riga")
sum = 0
count = 0
# è ancora un ciclo a contatore, ma il valore
# finale del conteggio non è più prefissato
while count < NUM :
    sum = sum + float(input())
    count = count + 1
print("Somma:", sum)
```



# Acquisire una sequenza di dati

- ❑ Può succedere, però, che l'utente non sappia *quanti saranno* i dati che fornirà in ingresso
  - ad esempio perché li sta acquisendo a sua volta da uno strumento di misura e la durata dell'esperimento non è prefissata
  - oppure, più in generale, perché è scomodo o impossibile contarli prima di iniziare a scriverli
- ❑ **Problema:** leggere una sequenza di dati in ingresso *finché i dati non sono finiti*
- ❑ **Soluzione:** usare un **approccio “a sentinella”**
  - L'utente introduce un dato “non valido” che segnala la fine dei veri dati
    - Ad esempio, un numero **negativo** per terminare una sequenza di numeri **positivi**

# Calcolare la somma di numeri **positivi**

```
print("Questo programma somma numeri positivi")
print("Scrivi i numeri da sommare, uno per riga")
print("Termina i dati scrivendo un numero negativo")
sum = 0
num = float(input()) # leggo il primo valore
# uso una sentinella non positiva
while num > 0 :
    sum = sum + num
    num = float(input()) # leggo il valore successivo

print("Somma:", sum)
"""
    controlliamo cosa avviene nel caso limite...
    se il primo valore è subito la sentinella,
    visualizza zero (il valore iniziale di sum):
    corretto
"""
```

**Lezione 13**  
**29/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

**Sentinella fuori dominio**

# Approccio a sentinella

## □ Sentinella

- L'utente introduce un dato “non valido” che segnala la fine dei veri dati

## □ **Problema:** non sempre è possibile individuare tali valori “non validi”, cioè *non appartenenti al dominio di elaborazione del problema*

- **Un programma che somma numeri** (non necessariamente positivi) deve poter accettare qualunque numero...
- Si può terminare la sequenza di dati con una stringa non numerica (cioè usando **una sentinella di tipo diverso dai dati**) anche se questo complica il codice che legge i numeri

# Sentinella di tipo diverso dai dati elaborati

```
# questo programma si propone di sommare numeri di
# qualsiasi valore, quindi non può usare una sentinella
# numerica: usiamo una stringa non numerica!
STOP = "STOP" # sentinella (è una costante...)
print("Scrivi i numeri da sommare, uno per riga")
print("Termina i dati scrivendo", STOP)
sum = 0
# ricordo che done è il participio passato di "to do" ☺
done = False # ho finito? ancora no...
while not done : # finché non ho finito...
    readString = input()
    # la stringa acquisita va convertita in numero
    # soltanto DOPO aver verificato che non sia la
    # sentinella, altrimenti la funzione float()
    # lancia un errore quando riceve la sentinella
    if readString == STOP : # uguaglianza tra stringhe
        done = True # terminerà al prossimo controllo
    else : # così float() non riceve mai la sentinella
        sum = sum + float(readString)
print("Somma:", sum)
```

# Alternativa...

```
STOP = "STOP"
print("Scrivi i numeri da sommare, uno per riga")
print("Termina i dati scrivendo", STOP)
sum = 0
# uso una variabile booleana avente un significato
# diverso, quindi ne cambio il nome
more = True # ci sono ancora numeri da elaborare?
while more : # finché ci sono numeri da elaborare...
    readString = input()
    if readString == STOP :
        more = False # terminerà al prossimo controllo
    else :
        sum = sum + float(readString)
print("Somma:", sum)
```

# Alternativa: l'enunciato **break**

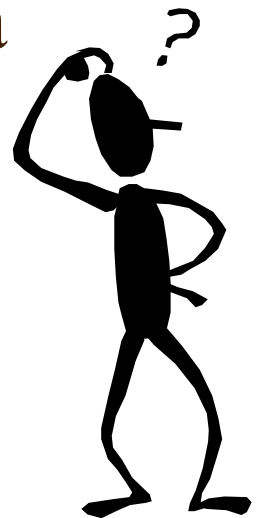
```
STOP = "STOP"
print("Scrivi i numeri da sommare, uno per riga")
print("Termina i dati scrivendo", STOP)
sum = 0
# non uso più la variabile booleana di controllo
while True : # sembra un ciclo infinito...
    readString = input()
    if readString == STOP :
        break # termina il ciclo immediatamente!
    else :
        sum += float(readString)
print("Somma:", sum)
```

- ❑ Quando l'interprete esegue l'enunciato **break**, pone termine **immediatamente** al ciclo che sta eseguendo
  - ❑ L'iterazione in corso **NON** viene completata, il ciclo termina "bruscamente" ma è ciò che voglio...
- ❑ L'esecuzione dell'enunciato **break** deve (ovviamente?) essere condizionata, cioè deve essere nel corpo di un **if** o **else**, altrimenti il ciclo si interrompe **sempre** durante la sua prima iterazione: sarebbe privo di senso, non sarebbe un ciclo



# Leggere una sequenza di dati

- ❑ A volte, però, un programma non può porre limiti ragionevoli al dominio dei dati... in alcuni casi non si può definire una sentinella!
- ❑ Es. un programma che chiede all'utente di digitare un testo generico su più righe, come un romanzo... per poi elaborarlo... che sentinella uso ?!?!?
- ❑ Neanche una riga vuota è accettabile, perché un testo qualsiasi può avere legittimamente una riga vuota (la funzione **input** restituisce la stringa vuota)
- ❑ Vedremo una soluzione più avanti



Chiedere ripetutamente un  
valore finché non rispetta i  
vincoli stabiliti

# Dati validi in input

- ❑ Finora abbiamo scritto programmi che confidano nel fatto che l'utente fornisca correttamente i dati richiesti
- ❑ Riusciamo a scrivere un programma che chiede un numero positivo e continua a chiederlo finché l'utente non fornisce effettivamente un numero positivo?
  - Posso ovviamente imporre una condizione qualsiasi...

```
while True :  
    value = float(input("Un numero positivo: "))  
    if value <= 0 :  
        print("Numero non valido:", value) # ciclo prosegue  
    else :  
        break  
# qui posso elaborare value: sono sicuro che è positivo!
```

- Attenzione: volendo essere rigorosi, questo non sarebbe un algoritmo, perché non c'è **garanzia** che il programma termini... se l'utente continua a sbagliare "per sempre"... lo consideriamo accettabile, perché il programma ha una **durata "infinita" soltanto in conseguenza di "infiniti" errori dell'utente**
- ❑ Rimane ancora il problema dell'utente che **non** fornisce un numero... problema con **float/int**... vedremo!

# Dati validi in input

- ❑ Realizzazione alternativa usando una variabile booleana per interrompere il ciclo

```
done = False
while not done :
    value = float(input("Un numero positivo: "))
    if value <= 0 :
        # non modifico done, quindi il ciclo proseguirà
        print("Numero non valido:", value)
    else :
        done = True # alla prossima verifica
                   # il ciclo si interromperà
# qui posso elaborare value: sono sicuro che è positivo!
```

- ❑ È migliore questa soluzione o la precedente?
  - Sono sostanzialmente equivalenti, possiamo dire che sia una questione di "stile personale"
  - È importante saper "leggere e capire" entrambe le forme

**Numeri casuali**

# Numeri casuali

□ Come facciamo a **generare un numero casuale** nel calcolatore? Ad esempio, per simulare il lancio di un dado? **Azione molto utile per i "giochi"...**

- sappiamo che il calcolatore fa sempre esattamente ciò che gli viene detto di fare mediante un programma, si comporta cioè in modo *deterministico*, e non casuale!

□ Esistono complesse teorie matematiche che forniscono algoritmi in grado di generare **sequenze di numeri pseudo-casuali** (cioè *quasi casuali*), che sono un'ottima approssimazione di sequenze di numeri casuali



- A partire, ad esempio, dall'istante di tempo in cui vengono eseguiti: un valore che cambia di volta in volta

# Numeri casuali

- ❑ Il linguaggio Python mette a disposizione la funzione `random()` per generare sequenze di numeri pseudo-casuali
- ❑ Ogni invocazione della funzione restituisce un numero frazionario pseudo-casuale nell'intervallo `[0, 1[` (cioè 0 compreso e 1 escluso...)
- ❑ Per ottenere, ad esempio, numeri interi casuali compresi nell'intervallo `[a, b]`, basta fare un po' di calcoli...


```
x = random()
```

```
x = int(a + (b-a+1)*random())  
# int(...), oltre ad accettare una stringa,  
# accetta anche un'espressione numerica, di  
# cui restituisce la parte intera (cioè la "tronca")
```

Oppure c'è un'altra comoda funzione, **randint**, che svolge esattamente lo stesso compito: restituisce un numero intero in `[a, b]`

```
x = randint(a, b)
```

# Numeri casuali e `import`

-  ☐ Per motivi che vedremo più avanti, per utilizzare le funzioni **`random`** e **`randint`** nei nostri programmi dobbiamo scrivere, all'inizio del programma, i seguenti enunciati di "importazione" del *modulo* **`random`**

```
from random import random
from random import randint
# oppure from random import random, randint
```

- ☐ Naturalmente se mi serve una sola delle due funzioni importerò soltanto quella
- In ogni caso un "import" inutile non fa danni...
- ☐ Se non lo scrivo... l'interprete Python mi segnala errore (es. *name 'randint' is not defined...*)



# Esempio: lanci di un dado

```
# averageDice.py
from random import randint
TRIALS = int(input("Quanti lanci? "))
sum = 0
count = 0
while count < TRIALS :
    # nuovo lancio di un dado a 6 facce
    d = randint(1, 6)
    sum += d # variabile d eliminabile
    count += 1
avg = sum / TRIALS
print("Average:", avg) # circa 3.5
# cosa succede se TRIALS non è positivo?
# meglio aggiungere un controllo iniziale...
```

# Esempio: lanci di un dado

- ❑ Eseguendo il programma `averageDice.py` più volte, si ottiene un diverso valore medio ogni volta, a conferma che i numeri casuali generati variano a ogni esecuzione, come dev'essere
- ❑ Si osserva che il valore medio che si ottiene è sempre *piuttosto vicino* a 3.5
- ❑ La “legge dei grandi numeri” (o Teorema di Bernoulli) afferma che il valore 3.5 viene raggiunto **esattamente** con un numero **infinito** di lanci del dado
  - dal punto di vista pratico, è sufficiente un numero di lanci di qualche milione...



## ❑ Come si può **scegliere casualmente tra DUE alternative**?

- Esempio: giocando contro il computer, spesso questo dovrà poter prendere delle decisioni "casuali" tra **due** alternative, del tipo "mossa verso destra o mossa verso sinistra?"
- È una decisione totalmente equivalente al risultato del lancio di un dado a DUE facce, cioè posso usare le formule precedenti con **a = 1** e **b = 2**
  - Es. Se il risultato è 1, mossa verso destra, altrimenti mossa verso sinistra [oppure il contrario!]
  - Vediamo alcune soluzioni equivalenti: **moveLeft** e **moveRight** sono due variabili booleane che, nel seguito, hanno sempre valori tra loro diversi (ovviamente, uno **True** e l'altro **False**), scelti a caso

```
if randint(1, 2) == 1 :  
    moveRight = True  
else:  
    moveRight = False  
moveLeft = not moveRight
```

```
moveRight = False  
if randint(1, 2) == 1 :  
    moveRight = True  
# qui else inutile...  
moveLeft = not moveRight
```

```
# un po' criptico ma ok  
moveRight = randint(1, 2) == 1  
moveLeft = not moveRight
```

```
if random() < 0.5 :  
    # non <=, perché?  
    moveRight = True  
else:  
    moveRight = False  
moveLeft = not moveRight
```

❑ Come si può **scegliere casualmente tra DUE alternative**, facendo però in modo che le due alternative **non** siano **equiprobabili**?

- Esempio: giocando contro il computer, spesso questo dovrà poter prendere delle decisioni "casuali" tra **due** alternative, del tipo "mossa verso destra o mossa verso sinistra?", ma **vogliamo che la probabilità di andare verso destra sia il doppio della probabilità di andare verso sinistra**



- La funzione **random()** restituisce un numero in virgola mobile compreso tra 0 (incluso) e 1 (escluso) con **distribuzione di probabilità uniforme** all'interno dell'intervallo, cioè **tutti i numeri appartenenti all'intervallo hanno la stessa probabilità di essere generati**, ossia non ci sono "numeri preferiti"
- Allegoria. Lancio una freccetta in modo che colpisca casualmente un punto di un segmento che rappresenta l'intervallo e ogni punto ha la stessa probabilità di essere colpito (il segmento ha, evidentemente, lunghezza unitaria). Se considero il sotto-segmento iniziale (o finale) di lunghezza uguale a  $\frac{2}{3}$  della lunghezza dell'intero segmento, qual è la probabilità di colpire tale sotto-segmento e qual è la probabilità di non colpirlo? Rispettivamente,  $\frac{2}{3}$  e  $\frac{1}{3}$ , quindi una delle due probabilità è il doppio dell'altra
- Con il codice qui a fianco, si avrà uno spostamento a destra nel 66.66% dei casi e uno spostamento verso sinistra nel 33.33% dei casi
- Analogamente per qualsiasi altro rapporto tra le probabilità o per scelte tra **più di due** alternative

```
if random() < 2/3 :  
    moveRight = True  
else:  
    moveRight = False  
moveLeft = not moveRight
```

❑ Come si può **scegliere casualmente tra TRE alternative**, facendo però in modo che le tre alternative siano **equiprobabili**?

- Suddivido idealmente la lunghezza del segmento unitario in **tre parti uguali**

- Analogamente per più alternative, anche non equiprobabili: basta calcolare le misure dei sotto-segmenti

```
scelta1 = False
scelta2 = False
scelta3 = False
x = random()
if x < 1/3 :      # 0 <= x < 1/3
    scelta1 = True
elif x < 2/3 :   # 1/3 <= x < 2/3
    scelta2 = True
else :           # 2/3 <= x < 1
    scelta3 = True
# i tre intervalli hanno la stessa ampiezza
# quindi le tre scelte sono equiprobabili
```

- **Attenzione: non così.**  
**Perché?**

```
scelta1 = False
scelta2 = False
scelta3 = False
if random() < 1/3 :
    scelta1 = True
elif random() < 2/3 :
    scelta2 = True
else :
    scelta3 = True
```

Il materiale necessario per il  
Laboratorio 03  
termina qui

**Lezione 14**  
**30/10/2024**  
**ore 10.30-12.30**  
**aula Ve**

**Esercizi:**  
**Cicli che elaborano stringhe**



# Cicli che elaborano stringhe

Esercizio già visto per un numero... ma quanto è più semplice così!

- ❑ Capita molto frequentemente di utilizzare cicli che elaborano stringhe, perché ci sono molti problemi che si risolvono compiendo **la medesima elaborazione su ciascun carattere di una stringa**, una **situazione ripetitiva** che ben si presta alla **realizzazione di un ciclo**

```
s = "AUGURI"
index = 0
while index < len(s) :
    print(s[index])
    index += 1
```

A  
U  
G  
U  
R  
I

- ❑ La struttura di questi cicli è sempre la stessa
  - Si usa una "variabile indice", che storicamente viene chiamata semplicemente **i** (anziché **index**) e assume, uno dopo l'altro, i valori corrispondenti a tutti (e soli) gli indici validi all'interno della stringa
  - Tale indice viene usato per **estrarre sottostringhe di lunghezza unitaria** dalla stringa elaborata: l'elaborazione, poi, è **identica** per ciascuna di tali sottostringhe e costituisce il corpo del ciclo

```
s = "AUGURI"
i = 0
while i < len(s) :
    print(s[i])
    i += 1
```

# Cicli che elaborano stringhe

- ❑ Un altro esempio: contiamo i caratteri della stringa che soddisfano una determinata condizione

```
s = "AUGURI"
count = 0
letter = "U" # lettera cercata
i = 0
while i < len(s) :
    if s[i] == letter :
        count += 1
    i = i + 1
print("Contiene", count, letter)
```

- ❑ Osserviamo che **la struttura del ciclo** è identica alla precedente: cambia solo l'elaborazione eseguita dal corpo
- ❑ Attenzione: ci sono due contatori... ma il contatore che controlla il ciclo è **i**, l'altro (**count**) serve a contare gli eventi che mi interessano...

- ❑ Data una stringa, ne vogliamo costruire un'altra che abbia **uno spazio interposto tra ciascuna coppia di caratteri consecutivi**, per poi visualizzarla

```
s = "AUGURI"
r = "" # stringa vuota che crescerà...
i = 0
while i < len(s) :
    r = r + s[i] + " " #anche r += s[i] + " "
    i += 1
print(r) # A U G U R I
```

- ❑ Di nuovo, **la struttura del ciclo** è identica alla precedente: cambia solo l'elaborazione eseguita dal corpo
- ❑ Algoritmo frequente: **"far crescere una stringa"**
  - In realtà, ad ogni iterazione del ciclo **viene costruita una nuova stringa** (dall'operatore di concatenazione), assegnata, poi, alla medesima variabile, quindi **"sembra"** che la stringa **r** si allunghi

# Cicli che elaborano stringhe

❑ Come si progetta un ciclo come questo?

❑ Per prima cosa **bisogna concentrarsi su ciò che dovrà fare una qualsiasi delle sue iterazioni**

(cioè il corpo del ciclo), senza pensare ai casi speciali (che di solito sono la prima e l'ultima iterazione)

```
s = ...  
r = ""  
i = 0  
while i < len(s) :  
    r = r + s[i] + " "  
    i += 1  
print(r)
```

- Questo spesso coinvolge **la variabile che funge da contatore**, se si tratta di un ciclo a contatore (cioè **il codice del corpo utilizza la variabile contatore**)
- In pratica, **bisogna ipotizzare che le iterazioni precedenti abbiano svolto il loro compito, producendo una soluzione parziale...** che va integrata durante l'iterazione generica che si sta prendendo in esame, **fornendo i dati corretti per l'iterazione successiva**
  - In questo esempio, la soluzione parziale è la stringa memorizzata in **r**
- Poi si verifica che tale corpo funzioni correttamente anche nelle iterazioni estreme (la prima e l'ultima)
  - Questo spesso **suggerisce come inizializzare alcune variabili**, in modo che la prima iterazione funzioni bene (es. **r = ""**)
  - A volte c'è bisogno di introdurre un **if** nel corpo, per gestire in modo diverso la prima e/o l'ultima iterazione (vedremo...)

# Cicli che elaborano stringhe

## □ Come si progetta un ciclo come questo?

- Per prima cosa **bisogna concentrarsi su ciò che dovrà fare una qualsiasi delle sue iterazioni** (cioè il corpo del ciclo), senza pensare ai casi speciali (che di solito sono la prima e l'ultima iterazione)

```
s = ...  
r = ""  
i = 0  
while i < len(s) :  
    r = r + s[i] + " "  
    i += 1  
print(r)
```

- Questo spesso coinvolge **la variabile che funge da contatore**, se si tratta di un ciclo a contatore (cioè il codice contiene la variabile contatore)
- In pratica, **bisogna ipotizzare che le iterazioni precedenti abbiano svolto il loro compito, producendo una soluzione parziale...** che va integrata durante l'iterazione generica che si sta prendendo in esame
  - In questo esempio, la soluzione parziale è la stringa memorizzata in **r**
- Poi si verifica che tale corpo funzioni correttamente anche nelle iterazioni estreme (la prima e l'ultima)
  - Questo spesso **suggerisce come inizializzare alcune variabili**, in modo che la prima iterazione funzioni bene (es. **r = ""**)
  - A volte c'è bisogno di introdurre un **if** nel corpo, per gestire in modo diverso la prima e/o l'ultima iterazione (vedremo...)

## □ Poi si sistema **la logica di controllo del ciclo**


- In questo caso, il valore iniziale e finale del contatore, oltre al suo incremento

- **Infine, si collauda!** Prima "a mente" o "a mano su carta", poi con l'interprete, eventualmente aggiungendo enunciati **print** interni al ciclo

# Cicli che elaborano stringhe

- Data una stringa, ne vogliamo costruire un'altra che abbia **uno spazio interposto tra ciascuna coppia di caratteri**

```
s = "AUGURI"
r = "***" # stringa che crescerà...
i = 0
while i < len(s) :
    r = r + s[i] + " "
    i += 1
r = r + "***"
print(r) # ***A U G U R I ***
```



- Piccolo problema... in realtà viene inserito uno spazio DOPO ogni carattere... quindi non soltanto tra ciascuna coppia di caratteri, ma anche DOPO L'ULTIMO... **non era previsto nelle specifiche**
  - È anche un problema di difficile diagnosi: durante il collaudo, lo spazio aggiunto NON SI VEDEVA!
  - Nel collaudo, si può usare **len(...)**

# Cicli che elaborano stringhe

- ❑ Piccolo problema... in realtà inserisce uno spazio DOPO ogni carattere... quindi non soltanto tra ciascuna coppia di caratteri, ma anche DOPO L'ULTIMO... non era previsto nelle specifiche
- ❑ Per risolvere il problema, basta fare **un'elaborazione speciale in occasione dell'ultimo carattere (quando  $i$  vale  $\text{len}(s) - 1$ )**

```
s = "AUGURI"
r = ""
i = 0
while i < len(s) :
    r = r + s[i] # per tutti
    if i != len(s) - 1 : # per tutti tranne l'ultimo
        r = r + " "
    i += 1
print(r) # ora OK
```

- ❑ Altre volte sarà necessario trattare in modo diverso IL PRIMO carattere... (quando  $i$  vale 0)

# Cicli che elaborano stringhe

## ❑ Costruiamo e visualizziamo la stringa inversa

```
s = "AUGURI"
r = "" # comodo che esista la stringa vuota...
        # la uso per iniziare a costruire il risultato
i = 0
while i < len(s) :
    # ricordo che la concatenazione non è commutativa!
    r = s[i] + r # NON si può abbreviare r += s[i]
    i += 1
print(r) # "IRUGUA"
```

## ❑ Algoritmo alternativo (cosa c'è di diverso?)

```
s = "AUGURI"
r = ""
i = len(s) - 1
while i >= 0 :
    r = r + s[i] # cambia anche il corpo del ciclo...
    i -= 1      # qui sopra si può abbreviare r += s[i]
print(r) # "IRUGUA" come prima
```

## ❑ Soluzioni equivalenti: stile personale...



# Cicli che elaborano stringhe

❑ Ci sono (almeno) due caratteri consecutivi uguali??

```
s = "Auguri Marcello"
found = False # trovati? ancora no...
i = 0
while i < len(s) - 1 : # ATTENZIONE al -1 ... perché?
    if s[i] == s[i + 1] :
        found = True
        break # inutile cercare ancora... ho trovato!
    i += 1
# found mi serve per conoscere, al termine del ciclo,
# il risultato della ricerca...
if found :
    print("Yes")
else :
    # alternativa senza usare break?
    print("No") # while i < len(s)-1 and not found :
```

❑ **Attenzione a non cercare di estrarre caratteri oltre la fine della stringa!!**

# Cicli che elaborano stringhe

❑ Ci sono (almeno) due caratteri consecutivi uguali??

```
s = "Auguri Marcello"
found = False # trovati?
i = 1 # ATTENZIONE
while i < len(s) :
    if s[i - 1] == s[i] :
        found = True
        break # inutile cercare ancora... ho trovato!
    i += 1
if found :
    print("Yes")
else :
    print("No")
# senza break l'algoritmo è ugualmente corretto
# (anche il precedente), ma (a volte) spreca tempo
```

❑ **Soluzione equivalente alla precedente**

- ❑ Combinando le soluzioni di problemi semplici si possono risolvere problemi più complessi, magari con **cicli annidati**!

```
STOP = "STOP"
print("Scrivi stringhe e termina con", STOP)
while True :
    s = input()
    if s == STOP : break # esempio di "corpo in linea"
    # elabora la stringa s, senza pensare di essere
    # all'interno di un ciclo più esterno 😊
    found = False # cerca caratteri consecutivi
    i = 1          # duplicati nella stringa s
    while i < len(s) : # CICLO ANNIDATO
        if s[i - 1] == s[i] :
            found = True
            break # interrompe solo il ciclo più interno,
                  # quello nel quale è inserito
        i += 1
    if found : print(s, "contiene caratteri duplicati")
    else : print(s, "non contiene caratteri duplicati ")
    #
    # qui sopra: esempi di "corpi in linea", per if/else
    # o while, quando il corpo è un solo enunciato
```

**Stringhe e sottostringhe**

# Creazione di una sottostringa

- ❑ Con una sintassi simile all'estrazione di un singolo carattere possiamo **generare** una stringa costituita da una porzione di caratteri **consecutivi** di un'altra stringa, specificando la posizione del carattere iniziale e la posizione di quello finale
  - Si parla di **sottostringa** (*substring*)
  - Di nuovo, a volte si dice "estrarre", un po' improprio

## ❑ Attenzione alla regola sul secondo indice...

```
greeting = "Hello, World!"  
s = greeting[0:4]  
print(s)
```

He1l

|   |   |   |   |   |   |   |   |   |   |    |    |    |
|---|---|---|---|---|---|---|---|---|---|----|----|----|
| H | e | l | l | o | , |   | W | o | r | l  | d  | !  |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

# Creazione di una sottostringa

## ❑ Alcune proprietà utili, da ricordare

- La posizione dell'ultimo carattere di una stringa corrisponde alla lunghezza della stringa meno 1, oppure semplicemente  $-1$
- **La differenza tra i due indici che definiscono una sottostringa corrisponde alla lunghezza della sottostringa generata**
  - La regola vale se i due indici sono concordi: se hanno segni diversi, bisogna aggiungere o sottrarre la lunghezza della stringa
- Se manca il primo indice si sottintende 0, cioè "dall'inizio"
- Se manca il secondo indice si sottintende la lunghezza della stringa, cioè "fino alla fine"
  - Se mancano entrambi?  
Risposta standard...  
**provate!!**
- Se i due indici sono uguali, si genera una sottostringa vuota (coerentemente con la seconda proprietà qui elencata)

```
s = "Hello"
print(s[:3]) # Hel
print(s[1:]) # ello
print(s[:])  # ???
```

# Creazione di una sottostringa

## ❑ L'estrazione di una sottostringa non genera mai errori

- (purtroppo) è una scelta progettuale **non coerente** con quella vista per l'estrazione di un singolo carattere (che è a tutti gli effetti una sottostringa di lunghezza unitaria...), dobbiamo tenercela così ☺

## ❑ Cosa succede se si usano indici (positivi o negativi) che fanno riferimento a posizioni della stringa che non esistono? Es. **s = "xyz" [5:7]**

## ❑ Cosa succede se il primo indice è maggiore del secondo (entrambi positivi)? Es. **s = "xyz" [2:1]**

## ❑ **Provare!**

- Ma prima ragionare... è mai possibile che vengano "inventati" caratteri inesistenti?

# Creazione di una sottostringa

❑ Questa sintassi è soltanto una scorciatoia

- Ma molto utile...
- `s[index]` equivale a `s[index:index+1]` se `index >= 0`, altrimenti equivale a `s[index-1:index]`
- "Equivale" ma diverso se `index` non è valido

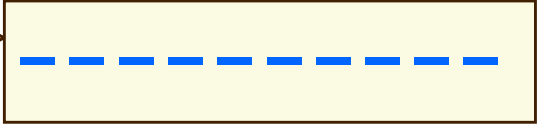
❑ Si potrebbe utilizzare un ciclo

```
s = "Hello"
a = 1
b = 4
x = s[a:b] # x = "ell"
y = "" # crescerà...
i = a
while i < b :
    y += s[i]
    i += 1
# y = "ell" come s[a:b]
```



# Concatenazione mediante ripetizione o "stringa ripetuta"

```
s = "-" * 10  
print(s)
```



A diagram illustrating the execution of the code. A box on the left contains the Python code `s = "-" * 10` and `print(s)`. An arrow points from this box to another box on the right, which contains ten blue dashes, representing the output of the code.

- ❑ Python mette anche a disposizione una "curiosa" forma di **concatenazione mediante ripetizione**, che può essere comoda
  - Si tratta soltanto di una "scorciatoia", lo stesso risultato si può ottenere con un ciclo [fare come esercizio]
- ❑ La stringa ripetuta può anche avere lunghezza maggiore di uno

```
s = "Ciao " * 4  
print(s)
```



A diagram illustrating the execution of the code. A box on the left contains the Python code `s = "Ciao " * 4` and `print(s)`. An arrow points from this box to another box on the right, which contains the string "Ciao Ciao Ciao Ciao" in blue text, representing the output of the code.

- ❑ Questa operazione è, in realtà, "commutativa", nel senso che il fattore (intero) di ripetizione può indifferentemente essere il primo o il secondo operando (e l'altro deve ovviamente essere una stringa), ma è consuetudine che la stringa sia il primo operando
  - Il fattore di ripetizione può, in generale, essere una espressione numerica intera

**Lezione 15**  
**05/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

# *Sequenze di escape*

# Sequenze di “escape”

- ❑ Proviamo a visualizzare una stringa che *contiene* delle virgolette

```
Hello, "World"!
```

# NON FUNZIONA!

```
print("Hello, "World"!")
```

- ❑ L'interprete identifica le seconde virgolette come la fine della prima stringa "Hello, ", ma poi non capisce il significato della parola **World**, che è fuori dalle virgolette
- ❑ Ci sono due possibili soluzioni...

- Usiamo come delimitatori le *virgolette singole* anziché doppie

```
print('Hello, "World"!')
```

- Usiamo una *sequenza di escape*



```
print("Hello, \"World\"!")
```

# Sequenze di “escape”

```
# FUNZIONA!  
print("Hello, \"World\"!")
```

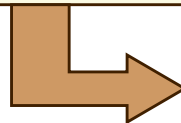
## ❑ Il carattere *backslash* all'interno di una stringa non rappresenta se stesso

- si usa per codificare altri caratteri che sarebbe *difficile* inserire in una stringa, per vari motivi, e si parla di *sequenza di escape*, perché serve a *far uscire* l'interprete dalla normale analisi lessicale

## ❑ Ma come si fa ad inserire veramente un carattere *backslash* in una stringa?

- si usa la sequenza di escape `\\`

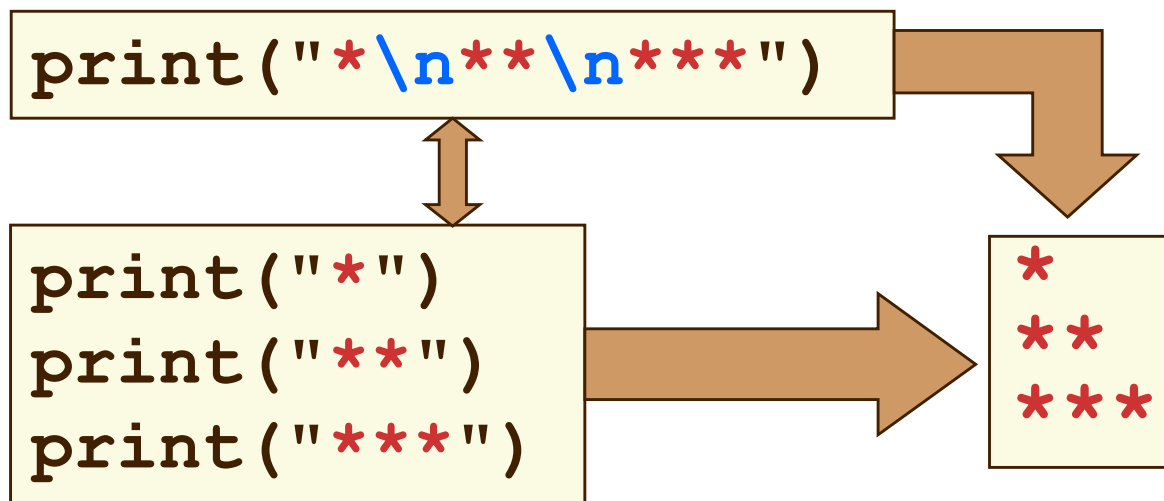
```
print("File C:\\autoexec.bat")
```



File C:\autoexec.bat

# Sequenze di “escape”

- Un'altra sequenza di escape che si usa spesso è `\n`, che rappresenta il carattere di “nuova riga” (*newline*) o “andare a capo”



# Sequenze di “escape”


- ❑ Le sequenze di escape si usano anche per inserire caratteri di lingue straniere o simboli che non si trovano sulla tastiera
- ❑ Ad esempio, per scrivere parole italiane con lettere accentate senza avere a disposizione una tastiera italiana `print("Perch\u00E9?")`
- ❑ Queste sequenze di escape utilizzano la codifica standard **Unicode**



Perché?

<http://www.unicode.org>

per rappresentare i caratteri di tutti gli alfabeti del mondo con *4 caratteri esadecimali* (*codifica a 16 bit*, 65536 simboli diversi)



# Sequenze di “escape”

- ❑ Ciò che viene effettivamente visualizzato sullo schermo quando si invia al flusso di uscita un carattere Unicode dipende, in realtà, da come è stato configurato il sistema operativo e da quali *font* di caratteri vi sono stati installati, perché è il sistema operativo che ha il compito di "disegnare" i caratteri corrispondenti a un determinato codice
  - **A volte NON si ottiene il simbolo desiderato, ma non è "colpa" di Python, bensì del font selezionato nel sistema operativo**
- ❑ Una cosa strana: "visualizzando" il carattere 0007 del codice Unicode... il computer emette un "beep" udibile! (che si chiama *alert*, da cui anche `\a`)

```
print("\u0007")
```



```
print("\a")
```

- Attenzione: in alcune versioni di IDLE non funziona...



# Python e Unicode

- ❑ Python mette a disposizione due funzioni predefinite che riguardano la codifica dei caratteri
  - La funzione **ord** riceve un carattere (cioè una stringa di lunghezza unitaria) e restituisce il numero intero non negativo che lo rappresenta nella codifica Unicode
  - La funzione **chr**, al contrario, riceve un numero intero non negativo e restituisce il carattere (cioè una stringa di lunghezza unitaria) che corrisponde a tale codice nello standard Unicode

```
print(chr(3+ord("a"))) # visualizza d
```

- Funzioni utili, ad esempio, per programmi di crittografia (come il Cifrario di Cesare, che vedremo...)

```
# maiuscole e minuscole...
dist = ord("a") - ord("A")
# dist è anche uguale a ord("b") - ord("B")
# e così via per ciascuna coppia di lettere
print(chr(ord("G") + dist)) # g
print(chr(ord("h") - dist)) # H
```

**Ancora print...**

# print senza andare a capo

- ❑ Come sappiamo, ogni invocazione di **print** visualizza una riga di testo, perché visualizza i propri argomenti e, poi, "va a capo"
- ❑ A volte, però, potremmo aver bisogno di **visualizzare una porzione di riga, senza andare a capo**, per completare la riga in seguito
- ❑ Questo effetto si può ottenere aggiungendo, nell'invocazione di **print**, un ulteriore argomento conclusivo, del tipo **argomento con nome** (di cui forse vedremo altri esempi più avanti)

Se manca l'argomento **end**, è come se fosse  
`end="\n"`

```
print("Hello, ", end="")  
print("World!")  
# visualizza un'unica riga
```

- ❑ L'argomento **end** (che, se presente, deve essere l'ultimo), a cui **diamo un valore** con una (specie di) assegnazione, contiene **una stringa** che viene visualizzata DOPO tutti i caratteri visualizzati da **print**, INVECE di andare a capo
  - Può essere qualsiasi stringa
  - Ma non facciamo "giochini" inutili... come questo

```
print("Hello, ", end=" ")  
print("World!")  
# come prima ma "brutto"
```

# print senza andare a capo

- ❑ Cerchiamo sempre di non confondere ciò che è **possibile** fare sintatticamente e ciò che, invece, è **sensato** fare!
- ❑ **Qualsiasi** stringa può essere usata come parametro **end**

```
print("Hello, ", end="Worl")  
print("d!")
```

Hello, World!

- ❑ **Ma l'unica stringa che ha senso usare con il parametro **end** è la stringa vuota!**
- ❑ Si può anche "esagerare" per aumentare la chiarezza...

```
NON_ANDARE_A_CAPO = "" # meglio NO_NEWLINE  
print("Hello, ", end = NON_ANDARE_A_CAPO)  
print("World", end = NON_ANDARE_A_CAPO)  
print("!")
```

# Esercizio

- ❑ Data una stringa, vogliamo visualizzare soltanto i suoi caratteri di indice dispari, uno di seguito all'altro **su un'unica riga**, senza spazi interposti
- ❑ Vediamo due soluzioni, la seconda forse più "naturale"...
- Se la stringa **s** è molto lunga, la prima soluzione usa molta memoria...

```
s = "qualcosa"
x = ""
i = 0
while i < len(s) :
    if i % 2 == 1 :
        x += s[i]
    i += 1
print(x) # "uloa"
# ho costruito
# una stringa con
# il risultato
```

```
s = "qualcosa"
i = 0
while i < len(s) :
    if i % 2 == 1 :
        print(s[i], end="")
    i += 1
print() # a capo finale
# ho visualizzato il
# risultato man mano che
# lo elaboravo, senza
# memorizzarlo
```

Il materiale necessario per il  
Laboratorio 04  
termina qui

**Ancora matematica...**

# Numeri in notazione scientifica

- ❑ Sappiamo che, in Python, per scrivere un valore numerico frazionario (tecnicamente, un "letterale" in *floating-point*) dobbiamo usare il carattere "punto" come separatore decimale, al posto della virgola
- ❑ I numeri frazionari si possono esprimere **anche** in "notazione scientifica" (molto utile per numeri con valore assoluto molto grande o molto piccolo), come  $2.3 \times 10^{-12}$ 

`fp = 2.3E-12`

  - La parte 2.3 si chiama ***mantissa***, mentre  $-12$  è l'***esponente***
  - Tanto l'esponente quanto la mantissa possono avere un segno
  - La mantissa può essere un numero frazionario (o intero), mentre **l'esponente deve essere un numero intero**
- ❑ La lettera E può essere minuscola? Si può mettere uno spazio prima e/o dopo la E? E più spazi? **Provare!**



# Valori numerici

❑ Come sappiamo, in Python esistono due tipi di valori numerici

- **Numeri interi**
- **Numeri frazionari (eventualmente con parte frazionaria uguale a zero), anche detti "in virgola mobile"**

❑ Non sappiamo ancora come e perché, ma dal punto di vista "tecnico", nella programmazione di calcolatori, il valore 2 è diverso dal valore 2.0



- 2 è un numero intero, 2.0 è un numero frazionario
- La differenza si vede facilmente fornendo i numeri alla funzione **print**
  - I numeri interi vengono visualizzati SENZA separatore decimale
  - I numeri in virgola mobile vengono sempre visualizzati con il separatore decimale, anche quando la parte frazionaria è uguale a zero

# Valori numerici

- ❑ È molto importante sapere che in **alcune situazioni Python richiede la presenza di numeri interi**, che sono DIVERSI dai numeri in virgola mobile con parte frazionaria uguale a zero
- ❑ **Esempio: gli indici nelle stringhe!**

```
s = "pippo"  
print(s[2])      # visualizza p  
print(s[2.0])  
TypeError: string indices must be integers
```

- ❑ Ovviamente nessuno scriverebbe un numero frazionario come indice in una stringa, il problema può nascere quando si usa un'espressione (o anche una singola variabile, che è comunque un'espressione...)
- ❑ **Che tipo di numero viene generato quando l'interprete valuta un'espressione numerica?**

# Valori numerici

- ❑ I **letterali** numerici sono del tipo "naturale", cioè
  - Se contengono esplicitamente il punto decimale (anche con parte frazionaria uguale a zero) sono frazionari
  - (Stranamente) Se sono espressi con la notazione scientifica (es. 2E3), sono frazionari anche se base ed esponente sono interi
  - Altrimenti sono interi
- ❑ Il valore restituito dalla funzione **int(...)** è sempre intero
- ❑ Il valore restituito dalla funzione **float(...)** è sempre frazionario
- ❑ Il risultato di operazioni aritmetiche è intero se e solo se tutti gli operandi sono interi
  - Quindi basta un solo operando frazionario per generare un risultato frazionario
  - Eccezione
    - **L'operatore / genera sempre un risultato frazionario, anche quando entrambi gli operandi sono interi!**
      - Questa regola può dare errori inattesi (es. `s[4/2]`):  
**quando si dividono due numeri interi, ricordarsi di usare //**  
**se si vuole ottenere un risultato intero**

# Alcune funzioni matematiche

- ❑ Per assegnare a una variabile il **valore assoluto** di un'espressione numerica si può ovviamente scrivere codice come questo

```
variabile = espressione  
if variabile < 0 :  
    variabile = - variabile
```

- ❑ Trattandosi di un'operazione frequente, Python mette a disposizione una funzione predefinita: **abs**

```
variabile = abs(espressione)
```

- ❑ **Ricordiamo:** quando un **corpo** (di un enunciato **if**, di una clausola **else/elif**, di un enunciato **while**, ecc.) è costituito da **un unico enunciato** si può evitare di andare a capo

```
variabile = espressione  
if variabile < 0 : variabile = -variabile
```

# Alcune funzioni matematiche

- ❑ Un'altra utile funzione predefinita è quella che arrotonda un numero all'intero più vicino

```
x = round(2.6) # x vale 3
```

- ❑ La funzione **round** può anche ricevere **un secondo argomento**, un numero intero che specifica il numero di cifre che devono comporre la parte frazionaria del numero **arrotondato** restituito (se non è specificato, il valore restituito sarà intero)

```
x = round(2.345742, 3) # x vale 2.346
```

- ❑ Se, invece, si vuole ottenere **la parte intera di un numero frazionario (senza arrotondamento)**, si può usare la funzione **int** che già conosciamo

- **int**, oltre a poter ricevere una stringa da convertire in numero intero, può anche ricevere un numero frazionario, che convertirà in numero intero mediante **troncamento** alla parte intera

```
x = int(2.6) # x vale 2
```

**Lezione 16**  
**06/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Riprendiamo l'esempio già visto

```
# FORMA LENTA
```

```
NUM = ...
```

```
s = "a" * NUM
```

```
x = ""
```

```
i = 0
```

```
while i < len(s) :
```

```
    if i % 2 == 1 :
```

```
        x += s[i]
```

```
    i += 1
```

```
print(x)
```

```
# FORMA VELOCE
```

```
NUM = ...
```

```
s = "a" * NUM
```

```
i = 0
```

```
while i < len(s) :
```

```
    if i % 2 == 1 :
```

```
        print(s[i], end="")
```

```
    i += 1
```

```
print()
```

❑ NUM = 1000000

- veloce circa 0.4s, lenta circa 4s, 10 volte più lento

❑ NUM = 2000000 (x2)

- veloce circa 0.8s (x2), lenta circa 24s (x6), 30 volte più lento

❑ NUM = 3000000 (x3)

- veloce circa 1.2s (x3), lenta circa 168s (x42), 140 volte più lento

❑ NUM = 4000000 (x4)

- veloce circa 1.6s (x4), lenta circa 640s (x160), 400 volte più lento

**Ancora matematica...**



# Alcune funzioni matematiche

- ❑ Per calcolare il valore minimo in un insieme di valori, si può usare la funzione predefinita **min**, che può ricevere valori numerici qualsiasi, interi o frazionari

```
x = min(2.6, 3, 1.22) # x vale 1.22
```

- ❑ Analogamente, la funzione **max** calcola il valore massimo

```
y = -0.6E2 # notazione scientifica...  
radiceDi2 = 2**0.5  
x = max(2 * y, radiceDi2, 1)  
# x vale 1.4142135623730951
```

- ❑ Stranamente queste funzioni segnalano errore se ricevono un solo parametro
  - mentre il concetto matematico di minimo/massimo di un insieme con un solo elemento è ben definito

# Funzioni predefinite "sparite"

- ❑ Sappiamo che non possiamo usare le **"parole chiave"** (*keyword*) del linguaggio come nomi di variabili

```
and = 3 # SyntaxError
```

- ❑ Ma possiamo (inavvertitamente) usare come nome di variabile il nome di una funzione predefinita!

```
max = 3
```

- Cosa significa? I nomi di funzioni sono nomi... come i nomi di variabili... **usando un nome identico una zona di memoria dove vengono archiviate informazioni...** l'interprete aveva archiviato nella zona di memoria di nome **max** il codice di una funzione predefinita... e noi "sovrascriviamo" (cioè cancelliamo) tale informazione, scrivendoci un numero!

- Cosa succede?

```
max = 3
x = max(2, 3)
# TypeError: 'int' object is not callable
```


- L'interprete ci ricorda che al nome **max** abbiamo associato un numero intero e ci comunica che un oggetto di tipo "numero intero" non è "invocabile" (***object is not callable***), perché non è il codice di una funzione!
- Purtroppo l'assegnazione **max=3** non è un errore di sintassi, me ne accorgo quando provo a invocare la funzione **max**... dobbiamo soltanto **ricordarci che il problema esiste e interpretare correttamente il messaggio d'errore**
- Attenzione: la cosa da ricordare non è "stiamo attenti a non usare nomi di funzioni come nomi di variabili" (sarebbe impossibile, perché non conosciamo tutte le funzioni...) ma "ricordiamoci il significato di **questo messaggio d'errore**"


**Metodi che elaborano  
stringhe**

# Stringhe: funzioni o metodi?

❑ Python mette a disposizione molti altri strumenti per elaborare stringhe, **oltre agli operatori e alle funzioni** che abbiamo visto

❑ Sono basati sul fatto che in Python le stringhe sono **oggetti**

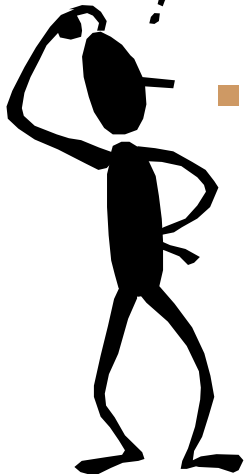
 ■ Come Python, i linguaggi moderni si caratterizzano per la capacità di implementare la **programmazione orientata agli oggetti** (**OOP** oppure **O<sup>2</sup>P**, *object-oriented programming*)

 ■ Nella programmazione, un **oggetto** è un'entità (software) che rappresenta il **valore** di un dato avente un determinato **comportamento**

• es. una stringa, una finestra in un ambiente grafico, un file di testo

■ Il comportamento di un oggetto è definito dai **metodi** con i quali lo si può manipolare

• **I metodi**, come le funzioni, **hanno un nome e sono sequenze di istruzioni** che portano a termine un compito specifico **elaborando un oggetto di un determinato tipo**



# Invocazione di un metodo

- ❑ La **semantica** dell'invocazione di un metodo è identica a quella dell'invocazione di una funzione
  - sospensione del metodo/funzione invocante, eventuale passaggio di argomenti, esecuzione del metodo, eventuale restituzione di un valore, "risveglio" dell'invocante
- ❑ La **sintassi** dell'invocazione di un metodo, invece, è un po' diversa da quella dell'invocazione di una funzione

```
s1 = "qualcosa"  
s2 = s1.upper()  
print(s2) # visualizza: QUALCOSA  
print(s1) # visualizza: qualcosa
```

"dot syntax"

- Inizia con **il nome di una variabile che contiene un dato del tipo che viene elaborato dal metodo**
  - o un letterale dello stesso tipo
- Prosegue con un carattere "punto", seguito dal **nome del metodo da invocare**, a sua volta seguito da una coppia di parentesi tonde che contengono gli eventuali argomenti

```
s = "qualcosa".upper()
```

# Invocazione di un metodo

- ❑ Se esistesse la **funzione predefinita** `upper` anziché il **metodo** `upper`, queste due invocazioni sarebbero equivalenti

```
s1 = "una stringa"  
s2 = s1.upper() # metodo  
s3 = upper(s1)  # funzione (in realtà non esiste)
```

- ❑ In pratica, un metodo non è altro che **una funzione che ha un parametro "speciale"** che viene fornito con la sintassi "punto" anziché all'interno delle parentesi (dove comunque possono essere presenti altri parametri, come vedremo in altri esempi)
- ❑ Come scegliamo tra metodo o funzione?
  - Per ora, **come programmatori "utenti" di codice scritto da altri, non possiamo scegliere**: dobbiamo soltanto documentarci, cercare un metodo o una funzione che svolga il compito che ci serve, e usarlo seguendo le scelte fatte dal progettista di quella funzione o di quel metodo
  - Più avanti impareremo a progettare funzioni e anche a progettare oggetti con i loro metodi... e vedremo qualche linea guida per scegliere tra una e l'altro



# Metodi utili per stringhe

Metodo con  
parametri

□ Vediamo alcuni metodi applicabili a stringhe

- **s.upper()** – Restituisce una stringa avente lo stesso contenuto di **s**, con lettere maiuscole al posto di lettere minuscole
  - Cosa succede alle lettere già maiuscole? Cosa succede ai caratteri che **non sono lettere** (es. **"Hello!".upper()**)? **Provare!**
- **s.lower()** – Restituisce una stringa avente lo stesso contenuto di **s**, con lettere minuscole al posto di lettere maiuscole
- **s.replace(old, new)** – Restituisce una stringa costruita a partire dal contenuto di **s**, sostituendo ogni eventuale occorrenza della stringa **old** con la stringa **new**
  - Fare esempi "strani", come **"xxxxx".replace("xx", "xxx")**

□ **Attenzione:** nessuno di questi metodi modifica il contenuto della stringa **s**, che viene soltanto usata come sorgente di informazioni per costruire la (nuova) stringa restituita

# Altri metodi che elaborano stringhe

- ❑ Nella elaborazione di stringhe, possono essere molto utili questi **metodi** (spesso applicati a stringhe di lunghezza unitaria, ma non solo)
  - **s.isalpha()** restituisce **True** se e solo se la stringa **s** contiene soltanto lettere, maiuscole o minuscole, e non è vuota
    - Cioè, in pratica, "è una parola", eventualmente un po' "strana" (es. "**ciAo**")
  - **s.isdigit()** restituisce **True** se e solo se la stringa **s** contiene soltanto cifre (da 0 a 9) e non è vuota
    - Cioè, in pratica, "è un numero intero non negativo" (con eventuali zeri iniziali)
  - **s.isalnum()** restituisce **True** se e solo se la stringa **s** contiene soltanto cifre (da 0 a 9) e lettere, maiuscole o minuscole, e non è vuota
  - **s.islower()** restituisce **True** se e solo se la stringa **s** contiene almeno una lettera e tutte le lettere contenute sono minuscole
    - Possono esserci anche caratteri che non siano lettere
  - **s.isupper()** restituisce **True** se e solo se la stringa **s** contiene almeno una lettera e tutte le lettere contenute sono maiuscole
    - Possono esserci anche caratteri che non siano lettere
- ❑ In generale, metodi o funzioni il cui nome inizia con "**is**" **restituiscono un valore booleano**, perché rispondono a una domanda del tipo "è ... ?"



# Esempio: contiamo le lettere maiuscole

```
s = "Ciao Marcello"
count = 0
i = 0
while i < len(s) : # solito ciclo di scansione...
    if s[i].isupper() : # falso se non è una lettera
        count += 1      # oppure è minuscola...
    i += 1

#
# ovviamente avrei potuto usare anche un ciclo con
# indice che si decrementa, analizzando i caratteri
# procedendo "a ritroso" nella stringa... a volte la
# direzione di scansione è importante, altre volte è
# indifferente
#
if count > 1 :
    print("Contiene", count, "lettere maiuscole")
elif count == 1 :
    print("Contiene una sola lettera maiuscola")
else : # certamente count vale 0
    print("Non contiene lettere maiuscole")
```

*for / range*

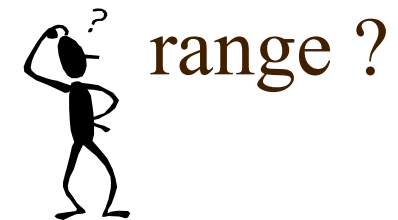
# Cicli a contatore

- Come abbiamo visto, è piuttosto frequente la programmazione di cicli controllati da una variabile che funge da contatore

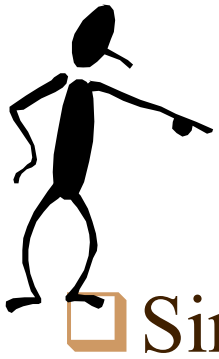
```
i = 0
while i < valoreLimite :
    fai qualcosa (che spesso dipende da i)
    i += 1
```

- Quasi tutti i linguaggi di programmazione, come Python, mettono a disposizione una struttura iterativa particolarmente dedicata alla **realizzazione di cicli a contatore**

```
for i in range(valoreLimite) :
    fai qualcosa (spesso usando i)
```



- La parola chiave **for** significa "for each", cioè "per ogni"
- Questi due cicli sono perfettamente **equivalenti**
  - È più difficile fare errori... ad esempio, non posso dimenticarmi l'incremento della variabile di controllo, perché viene eseguito automaticamente e implicitamente **al termine di ogni iterazione!**



# L'enunciato for

❑ Sintassi:

```
for variabile in contenitore :  
    enunciato
```

❑ Scopo: eseguire ripetutamente un *enunciato* (detto “corpo”), una volta per ogni elemento appartenente al *contenitore*

- Ad ogni iterazione del ciclo, la *variabile* assume, uno dopo l'altro **ordinatamente**, i valori presenti nel *contenitore*

- Solitamente *enunciato* utilizza la *variabile*, ma non è necessario

```
for x in range(5) :  
    print("Ciao")
```

- Se il contenitore è vuoto, l'*enunciato* non viene mai eseguito

❑ Il *contenitore* è, in generale, un elenco di valori (anche di tipi diversi), separati da virgole e solitamente racchiusi da una coppia di parentesi tonde o quadre (tali parentesi non sono necessarie, ma **è meglio metterle**, per chiarezza)

```
for x in 2, "ab", 0 :  
    print(x)  
for x in (2, "ab", 0) :  
    print(x)  
# questi contenitori sono,  
# in realtà, rari...  
# vedremo in seguito i  
# casi più frequenti
```



# Contenitori per cicli `for`

- ❑ Esempio di **contenitore** utilizzabile in un ciclo `for`:  
**una stringa!**

```
for c in "pippo" :  
    fai qualcosa con c
```


- ❑ Attenzione: **una stringa** è "un contenitore di caratteri", quindi i valori assunti dalla variabile sono i singoli caratteri della stringa, NON i loro indici!

- ❑ **Ciclo equivalente**

```
s = "pippo"  
i = 0  
while i < len(s) :  
    c = s[i]  
    fai qualcosa con c  
    ma senza usare i  
    i += 1
```

- ❑ Se devo elaborare caratteri e NON mi servono gli indici, il ciclo `for` è decisamente più comodo!

# Contenitori per cicli `for`

- ❑ Quando, invece, come contenitore ci serve una sequenza di numeri interi, viene in aiuto la "funzione" predefinita **`range`**
  -  tecnicamente non è proprio una funzione, ma si comporta come se lo fosse...
- ❑ **`range` genera e restituisce un contenitore di numeri interi**, costruito secondo regole che dipendono dagli argomenti che riceve, che devono essere valori **interi** (non necessariamente letterali)
  - **Tre argomenti (*start*, *stop*, *step*):**  
numeri interi  $x = start + i * step$ , con  $i$  intero crescente a partire da 0, fintantoché  $x < stop$  se  $step > 0$  oppure  $x > stop$  se  $step < 0$ ;  
il contenitore può risultare vuoto senza che questo sia un errore (es. **`range(0, 0, 1)`**);  $step = 0$  è l'unico errore (**`ValueError`**)
  - Forme abbreviate... perché frequenti
    - **Due argomenti (*start*, *stop*):** [come se fosse  $step = 1$ ]  
numeri interi **consecutivi crescenti** (anche negativi, se  $start$  è negativo) da  $start$  (incluso) a  $stop$  (escluso); se  $stop \leq start$ , il contenitore è vuoto
    - **Un solo argomento (*stop*):** [come se fosse  $start = 0$ ,  $step = 1$ ]  
era il nostro primo esempio, molto frequente: numeri interi consecutivi crescenti da 0 (incluso) a  $stop$  (escluso); se  $stop$  non è positivo, il contenitore è vuoto

**Lezione 17**  
**08/10/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Esempi

- Data una stringa,  
quanti spazi contiene?  
**Non servono gli indici...**

```
s = ... # una stringa
count = 0
for c in s :
    if c == ' ' :
        count += 1
print(count)
```

- Data una stringa, contiene  
almeno una coppia di caratteri  
consecutivi uguali?  
**Servono gli indici...**

```
s = ... # una stringa
hasDuplicates = False
for i in range(1, len(s)) :
    if s[i-1] == s[i] :
        hasDuplicates = True
        break
print(hasDuplicates)
```

```
s = ... # una stringa
hasDuplicates = False
for i in range(len(s)-1) :
    if s[i] == s[i+1] :
        hasDuplicates = True
        break
print(hasDuplicates)
```

- Osservare analogie e  
differenze tra queste due  
soluzioni equivalenti

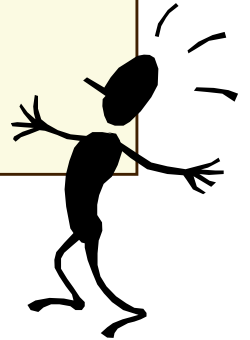


# Esempi

- ❑ Fare attenzione: nel ciclo **while** la *condizione di continuazione* viene **valutata nuovamente** all'inizio di ogni iterazione (per decidere se proseguire il ciclo), mentre nel ciclo **for** che usa **range** **il contenitore viene creato prima dell'inizio del ciclo**
- ❑ Questo può avere effetti "strani" (anche se, ovviamente, completamente prevedibili) se il corpo del ciclo modifica alcune delle variabili coinvolte nella creazione del contenitore

```
s = ... # una stringa
i = 0
while i < len(s) :
    print(s[i])
    s = s[i:]
    i += 1
# analisi: cosa fa ?
```

```
s = ... # una stringa
for i in range(len(s)) :
    print(s[i])
    s = s[i:]
# analisi: cosa fa ?
```



- ❑ **Esercizio (di analisi):** **prevedere** ("a mano" o "a mente") il funzionamento di questi due esempi, APPARENTEMENTE equivalenti, **poi verificarlo** mediante esecuzione del codice

# Progettazione di funzioni

# Ripetizione di codice

- ❑ Immaginiamo un programma che debba chiedere all'utente di fornire tre numeri interi positivi, ciascuno con un diverso significato, reiterando la richiesta di singoli valori errati

```
while True :  
    n1 = int(input("Numero 1: "))  
    if n1 > 0 : break  
while True :  
    n2 = int(input("Numero 2: "))  
    if n2 > 0 : break  
while True :  
    n3 = int(input("Numero 3: "))  
    if n3 > 0 : break
```

- ❑ Per ogni acquisizione reiterata servono tre righe di codice, che sono tra loro identiche! Tranne per il messaggio da visualizzare...
  - Avere sezioni di codice ripetuto, con le diverse sezioni che devono svolgere la stessa elaborazione, è sempre fonte di errori: quando modifico una sezione, devo ricordarmi di modificare coerentemente le altre (e se sono in zone del file molto distanti tra loro?)
  - Es. voglio aggiungere in tutte le richieste:  
**else: print("Avevo chiesto un numero positivo")**

# Ripetizione di codice

- ❑ Non posso pensare di fare un ciclo "esterno" che esegua per **tre volte** l'azione di acquisizione reiterata, perché non saprei che nome dare alla variabile!

```
i = 0
while i < 3 :
    while True :
        ??? = int(input("Numero " + str(i+1) + ": "))
        if ??? > 0 : break
    i += 1
```



Vedremo una soluzione  
usando una "lista"

- ❑ Anche questo non va bene: basta che un numero sia sbagliato e l'utente li deve fornire di nuovo tutti

```
while True :
    n1 = int(input("Numero 1: "))
    n2 = int(input("Numero 2: "))
    n3 = int(input("Numero 3: "))
    if n1 > 0 and n2 > 0 and n3 > 0 : break
```

- ❑ E se la singola azione fosse costituita da 10 righe di codice? Ancora peggio...

# Ripetizione di codice

- ❑ Siamo abituati a **svolgere azioni ripetute in punti diversi** del nostro file sorgente (es. `print`), senza preoccuparci di quale sia il codice che realizza tali azioni ripetute... ma sappiamo che l'elaborazione sarà svolta in modo identico ogni volta che ci serve
  - **Invochiamo una funzione!**
    - Predefinita o di un modulo...
  - Come avranno fatto "gli altri" a progettare le funzioni che troviamo nei moduli?
    - **Dobbiamo imparare anche noi!**
  - Cosa fa una funzione ogni volta che viene invocata?
    - Riceve eventualmente argomenti
    - Svolge un'elaborazione
    - Restituisce eventualmente un valore
  - Vediamo quale sintassi si usa per **progettare una funzione** che acquisisca un numero intero visualizzando un messaggio di richiesta e reiterando l'acquisizione finché il numero acquisito non è positivo, per poi restituirlo all'invocante

# Definizione di funzione



```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n
```

- ❑ La definizione di una funzione inizia con la parola chiave **def**, seguita dal **nome della funzione** (solitamente un verbo)
  - Scelto dal progettista in modo che sia descrittivo dell'elaborazione svolta, diverso da quello di altre funzioni, seguendo le normali regole sui nomi: tutto minuscolo, iniziali maiuscole per le parole intermedie, ecc...
- ❑ Dopo il nome della funzione dobbiamo scrivere una coppia di parentesi tonde, seguita dal carattere "due punti"



- Dentro alle parentesi mettiamo i **nomi delle variabili** che, durante l'esecuzione della funzione, memorizzeranno i **valori degli argomenti ricevuti**: in questo caso, un messaggio da visualizzare, che, così, può essere deciso da chi invoca la funzione, in generale diverso a ogni invocazione
- Se sono previsti più argomenti, i nomi delle variabili corrispondenti devono essere separati da virgole
- Le parentesi possono essere vuote, ma ci devono essere

# Utilizzo di una "nostra" funzione

- ❑ Prima di procedere con l'analisi della definizione della nostra prima funzione, vediamo come la utilizzeremo per risolvere il problema precedente
- ❑ Come diventa il nostro programma?



```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n  
  
# inizio del programma  
n1 = inputPositiveInteger("Numero 1: ")  
n2 = inputPositiveInteger("Numero 2: ")  
n3 = inputPositiveInteger("Numero 3: ")  
...
```

- ❑ Abbiamo risolto il problema!!
  - Il codice di acquisizione di un numero è stato scritto una volta sola

# Definizione di funzione

```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n
```

- ❑ La prima riga della definizione di una funzione, contenente la parola chiave **def**, viene solitamente chiamata *firma* (*signature*) o *intestazione* (*header*) della funzione stessa
- ❑ Il **corpo** (*body*) della funzione, che definisce le istruzioni che verranno eseguite ogni volta che la funzione verrà invocata, deve essere **indentato**, come ogni corpo (corpo degli **if**, corpo dei **while**, ...)
- ❑ Al suo interno scriviamo il codice della funzione usando la sintassi ordinaria, con due sole differenze
  - Le variabili (dette *variabili parametro*) il cui nome è stato definito nella firma della funzione sono disponibili all'interno del corpo e assumeranno i valori forniti come argomenti durante ciascuna invocazione della funzione
    - Solitamente non ha senso assegnare loro un valore nel corpo della funzione!
  - Il corpo della funzione può utilizzare un nuovo tipo di istruzione: **return**
    - L'istruzione **return** pone fine all'esecuzione della funzione e restituisce all'invocante il valore presente alla sua destra, se ce n'è uno
    - Se viene eseguita l'ultima istruzione di una funzione senza che venga eseguita un'istruzione **return**, si assume la presenza di un **return** implicito (ovviamente senza alcun valore restituito)



# Interprete e definizione di funzioni

```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n  
# inizio del programma  
n1 = inputPositiveInteger("Numero 1: ")  
...
```

- ❑ Sappiamo che l'interprete esegue le istruzioni di un programma procedendo dall'alto in basso
  - Ma le prime 5 righe di questo programma non vengono eseguite "subito"...
    - **Costituiscono un'unica istruzione (composita) che definisce una funzione!**  
La parola chiave **def** introduce la definizione di una funzione, che è costituita dalla sua firma e dal suo corpo: finché non è finito il corpo, le istruzioni vengono **lette dall'interprete ma non eseguite**, servono soltanto a comporre la definizione della funzione stessa, che viene memorizzata dall'interprete (come ha già fatto per le cosiddette "funzioni predefinite" e per le funzioni importate)
  - La prima istruzione che viene eseguita dal programma è quella che assegna un valore a **n1**, invocando per la prima volta la funzione **inputPositiveInteger**
    - A questo punto vengono eseguite le istruzioni della funzione, perché la funzione è stata invocata

# Nomi "locali"

```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
        return # potremmo non restituire n?  
# ora la funzione non restituisce un valore  
# quindi non può stare a destra in una  
# assegnazione... l'invocazione sta da sola  
inputPositiveInteger("X: ") # genera n...  
print(n) # e ce lo andiamo a prendere...  
# NameError: name 'n' is not defined
```

- ❑ I nomi delle **variabili definite all'interno di una funzione** (comprese le sue variabili parametro) **non sono visibili al di fuori di essa!**
  - Meno male! Altrimenti non potremmo usare nessun nome di variabile già utilizzato all'interno delle funzioni predefinite o dei moduli che usiamo... il cui codice sorgente non ci è nemmeno noto!
  - Si chiamano **variabili locali**
- ❑ Invece, le variabili definite al di fuori di una funzione (come tutte quelle che abbiamo usato finora) si chiamano **variabili globali**
- ❑ Si parla di **ambito di visibilità** delle variabili (*variable scope*)
  - L'ambito di visibilità di una variabile locale è la funzione in cui è definita

# Nomi "locali"


```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n  
# inizio del programma  
message = "Ciao"  
x = inputPositiveInteger("X: ")  
print(message) # "Ciao"
```

- ❑ Tra queste due variabili **message** non c'è nessuna relazione, fanno riferimento a due zone di memoria diverse perché si trovano in **ambiti di visibilità diversi**
  - È come se l'interprete conoscesse "la variabile **message** definita nella funzione **inputPositiveInteger**" e "la variabile **message** definita nel programma principale", cioè a livello globale: **non c'è nessuna relazione tra loro, anche se hanno lo stesso nome**
- ❑ Il fatto che, durante l'esecuzione della funzione **inputPositiveInteger**, alla sua variabile **locale message** venga implicitamente assegnata la stringa "X: " (che è stata ricevuta come argomento) non ha alcun effetto sul valore della variabile **globale message**, che rimane "Ciao", come verifichiamo con l'enunciato **print** conclusivo


# Nomi "locali"

- ❑ Quindi, il codice di una funzione NON ha accesso alle variabili definite nel codice invocante e **tutte le informazioni di cui la funzione ha bisogno devono essere ricevute come argomenti**

```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n  
# inizio del programma  
x = inputPositiveInteger("X: ")
```

-  In realtà il discorso è un po' più complesso (perché le variabili globali sono comunque accessibili in qualunque punto del codice, anche all'interno delle funzioni...), ma **noi usiamo questa semplificazione perché agevola la manutenzione/modifica del codice**

- ❑ Analogamente, **tutte le informazioni che una funzione vuole trasferire all'invocante devono essere "riassunte" dal valore restituito**, perché le sue variabili (locali e parametro) non sono visibili all'invocante (si dice che tali variabili "spariscono" dopo che la funzione ha terminato la propria esecuzione)

-  Vedremo come si possa restituire un unico valore che... contiene molti valori!
  - In modo analogo all'utilizzo di una stringa, che è una sequenza di caratteri... quindi è già un'informazione "composita"...
  - Es. restituisco la stringa "**123 43**", che contiene due numeri separati da un carattere concordato, da cui l'invocante potrà estrarre i due numeri... lavorando un po' 😊

## ❑ Riassumendo:

- Valori **che saranno elaborati** da una funzione: trasferiti nei **parametri**
- Valori **prodotti** da una funzione: trasferiti nel **valore restituito**

**Lezione 18**  
**12/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Interprete e definizione di funzioni

```
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n  
# inizio del programma  
n1 = inputPositiveInteger("Numero 1: ")  
...
```

- ❑ La prima istruzione che viene eseguita è quella che assegna un valore a **n1**, invocando per la prima volta **inputPositiveInteger**
- ❑ Quando l'interprete **ESEGUE** l'invocazione di una funzione, ne deve conoscere il codice (cioè la firma e il corpo), in uno dei modi seguenti
  - Perché è una funzione predefinita del linguaggio (cioè auto-importata...)
  - Oppure perché è una funzione importata da un modulo (mediante **import**)
  - Oppure perché è una funzione definita **IN PRECEDENZA** nel file stesso
- ❑ Quindi la definizione della funzione **inputPositiveInteger** **deve precedere qualsiasi suo utilizzo**, altrimenti l'interprete segnala l'errore **NameError**, perché il nome della funzione non è noto: è come usare una variabile senza averle assegnato un valore!

# La funzione `main`

- ❑ Per rispettare le esigenze dell'interprete, bisognerebbe iniziare un programma con la definizione di tutte le funzioni che servono (così come con tutte le importazioni che servono...)
  - Così facendo, però, la descrizione dell'elaborazione **principale** svolta dal programma si troverebbe in fondo al file, più "scomoda" per un lettore umano, magari più interessato alla logica complessiva del programma piuttosto che ai dettagli delle singole specifiche funzioni
- ❑ Di solito si usa una strategia diversa
  - all'inizio del file si definisce una "funzione principale" che, in realtà, contiene (nel proprio corpo) le istruzioni del programma: per convenzione, tale funzione viene solitamente chiamata **main** (cioè, appunto, *principale*)
  - successivamente si definiscono tutte le funzioni "ausiliarie", invocate dalla funzione principale
  - infine, si scrive la prima e unica istruzione del programma, che è l'invocazione della funzione **main()**



# La funzione `main`

```
def main() : # nome convenzionale
    n1 = inputPositiveInteger("Numero 1: ")
    n2 = inputPositiveInteger("Numero 2: ")
    n3 = inputPositiveInteger("Numero 3: ")
    ...

def inputPositiveInteger(message) :
    while True :
        n = int(input(message))
        if n > 0 : break
    return n
# eventuali altre definizioni di funzioni...

# ecco il programma da eseguire
main()
```

Nella seconda/terza/quarta riga, l'interprete legge il nome della funzione *inputPositiveInteger*, che ancora non conosce, ma non la deve eseguire! Quando la eseguirà (perché eseguirà la funzione *main*), l'avrà conosciuta.

Usiamo questa  
strategia

□ Questa convenzione è molto seguita

- Quando vedo un file che inizia con la definizione della funzione **main**, non ho bisogno di andare in fondo al file per scoprire quali sono le istruzioni del programma, perché so che ci sarà soltanto l'invocazione di **main()**, quindi so che la funzione **main** contiene, in realtà, il programma principale



# Perché definire funzioni?

- ❑ La possibilità di definire funzioni è uno strumento che, in generale, può essere di grande aiuto nella **scomposizione di un problema complesso in problemi più semplici**
  - Può essere utile anche se la funzione viene invocata una sola volta, perché l'isolamento di una porzione di un algoritmo ne consente il collaudo senza dover collaudare il programma completo
    - Si parla di *unit testing* o collaudo di unità (nel senso di *unità funzionale*)
- ❑ La definizione di **funzioni invocate più volte in un programma**, poi, consente di concentrare in un unico punto le modifiche da apportare alla porzione di algoritmo implementata da ciascuna funzione (concetto simile all'utilizzo di costanti al posto di valori letterali), assegnando anche un **nome esplicativo** alla funzionalità svolta da una sezione di codice
  - Vediamo un esempio che riguarda la nostra funzione **inputPositiveInteger**

# Manutenzione del codice

- ❑ Tra le attività di manutenzione del codice, c'è il **miglioramento** degli algoritmi utilizzati per realizzare singole specifiche attività
- ❑ Ad esempio, **la nostra funzione**, che continua a chiedere dati all'utente finché non viene inserito un numero intero positivo, **richiede comunque la collaborazione dell'utente stesso**, perché, se viene introdotta una stringa che non sia un numero intero, la funzione interrompe l'esecuzione del programma
  - Possiamo rendere la nostra funzione più "robusta" ?

```
def inputPositiveInteger(message) :  
    while True :  
        s = input(message)  
        if s.isdigit() :  
            n = int(s) # certamente nessun errore!  
            if n > 0 : break  
    return n
```

Problema con segno + e spazi...  
accettati da `int` ma non da `isdigit`

- Ora funziona addirittura se l'utente va semplicemente a capo...  
(nel qual caso **input** restituisce la stringa vuota)

# Manutenzione del codice

```
def inputPositiveInteger(m) :  
    while True :  
        s = input(m)  
        if s.isdigit() :  
            n = int(s)  
            if n > 0 : break  
    return n  
... # il programma che c'era prima, senza modifiche
```

- ❑ Avendo definito all'interno di una funzione il codice che acquisisce un numero intero, è sufficiente modificare soltanto tale definizione e, rieseguendo il programma, tutte le invocazioni di **inputPositiveInteger** trarranno beneficio della nuova funzionalità!
- ❑ Rimane ancora una scomodità: una funzione come questa sarà **probabilmente utile per molti programmi**, quindi ne dobbiamo copiare il codice in molti file
  - La manutenzione e i miglioramenti tornano a essere scomodi: **devo fare le stesse modifiche in molti file diversi**
  - Ma in Python esistono i **moduli** importabili!

# Progettazione di un modulo

- ❑ Per creare **un modulo** di funzioni importabili, è sufficiente inserire **in un normale file sorgente Python** le definizioni delle funzioni che fanno parte del modulo, **SENZA inserire un programma eseguibile**
  - Solitamente sono funzioni che hanno un "tema" comune ma tecnicamente questo non è necessario, è solo una convenzione

```
# file unProgramma.py
from myinput import * # è come se il codice di myinput.py
                      # fosse scritto qui...
```

```
    # il file myinput.py deve essere nella stessa cartella
```

```
def main() :
```

```
    n1 = inputPositiveInteger("Numero 1: ")
```

```
    n2 = inputPositiveInteger("Numero 2: ")
```

```
    n3 = inputPositiveInteger("Numero 3: ")
```

```
    ...
```

```
    # fine di main()
```

```
# definizione di altre
```

```
# eventuali funzioni usate
```

```
# dal programma e non
```

```
# importate
```

```
main()
```

```
# (ad esempio) file myinput.py
def inputPositiveInteger(message) :
    # la funzione che abbiamo
    # progettato prima...
```

```
def unAltraFunzioneDiInput(...) :
```

```
    ...
```

```
# programma eseguibile ASSENTE
```

```
# altrimenti? Cosa succede?
```

# Impaginazione dei risultati

# Impaginazione dei risultati

- Spesso quando si visualizzano i risultati dell'elaborazione svolta da un programma (o, più in generale, si visualizzano informazioni), si vuole avere un **controllo preciso sul formato** delle informazioni visualizzate
  - Ad esempio, visualizzando una quantità di denaro in dollari o euro, si vorranno visualizzare **soltanto due cifre** della sua parte frazionaria
  - È un problema che sappiamo già risolvere...

```
euro = 2.453
```

```
print(round(euro, 2), "€")
```

2.45 €

- Però solitamente in questi casi vogliamo visualizzare **SEMPRE due cifre** della parte frazionaria, anche quando l'ultima è zero o entrambe lo sono

```
euro = 2.4
```

```
print(round(euro, 2), "€")
```

2.4 €

**BRUTTO!**

- Avremmo decisamente preferito

2.40 €

# Impaginazione dei risultati

- ❑ Ovviamente possiamo scrivere codice che produca il risultato voluto, ma se tutte le volte dovessimo fare così...

Esercizio di  
analisi, da  
fare in  
proprio,  
molto  
interessante

```
euro = 2.4 # oppure 2 oppure 2.45 oppure 2.4532
s = str(round(euro, 2)) # al massimo due cifre decimali
# cerco il punto in s, ad esempio partendo dalla fine
i = -1 # ad esempio, uso indici negativi (non necessario)
while i >= -len(s) : # sempre attenzione ai limiti...
    if s[i] == "." : break # trovato il punto!
    i -= 1 # attenzione, quando viene eseguito break il
           # valore di i non diminuisce: alla fine, i è
           # l'indice della posizione in cui si trova il punto
if i == -len(s) - 1 : # ciclo terminato senza break
    # ciclo terminato senza trovare il punto: numero intero
    s = s + ".00"
elif i == -2 : # il punto è il penultimo carattere, quindi
    # "punto trovato", ma c'è una sola cifra decimale
    s = s + "0" # aggiungo uno zero finale
# altrimenti i vale sicuramente -3 e non devo fare niente
# perché ci sono già esattamente due cifre decimali
print(s, "€") # qui s ha SEMPRE due cifre decimali!
```

# Impaginazione dei risultati

- ❑ Per aiutarci a risolvere questi problemi di impaginazione, Python mette a disposizione

l'**operatore di formato per stringhe**

```
eu = 2.4
x = "%.2f €" % round(eu, 2)
print(x) # 2.40 €
```



- ❑ Abbiamo già visto l'operatore `%`
  - Quando i suoi due operandi sono espressioni numeriche, calcola il **resto della divisione intera**
  - Quando il suo operando di sinistra è una stringa, agisce come **operatore di formato** o di impaginazione
    - **Il risultato prodotto è una stringa**
      - che, in questo caso, verrà visualizzata da **print**, ma l'**operatore di formato è utilizzabile in generale per "costruire" stringhe**
    - Spesso si usa direttamente come argomento di **print**

```
eu = 2.4
print("%.2f €" % round(eu, 2)) # come prima
```



# Operatore di formato

I caratteri che **non fanno**  
**parte di indicatori di formato**  
rimangono come sono

- ❑ A sinistra dell'operatore di formato deve essere presente **una stringa**, che diventa il risultato prodotto dall'operatore, con l'eccezione degli **indicatori di formato** (*format specifier*) presenti in essa, che vengono **sostituiti** seguendo regole che ora vedremo



```
x = "% .2f €" % 2.406 # x = "2.41 €"
```


- ❑ Se la stringa contiene un solo indicatore di formato, questo viene sostituito dall'unico valore che deve essere presente alla destra dell'operatore di formato
  - Se la stringa contiene più indicatori di formato, questi vengono sostituiti dai corrispondenti valori presenti alla destra dell'operatore di formato: questi vanno separati da virgole e **racchiusi da una coppia di parentesi tonde**



```
x = "% .2f TT % .1f" % (2.406, 3)  
# equivale a x = "2.41 TT 3.0"
```

Gli spazi sono  
caratteri  
come gli altri!

# Indicatori di formato

❑ Gli indicatori di formato iniziano con un carattere %, seguito da  **informazioni opzionali** e da un carattere che identifica il tipo di dato che verrà inserito nella stringa prodotta, **al posto dell'indicatore stesso**

- **%f** – Un numero frazionario (con 6 cifre decimali, arrotondando)
  - il separatore decimale sarà presente anche se la parte frazionaria è uguale a zero, le 6 cifre decimali sono sempre presenti (eventualmente zeri...)
- **%e** – Un numero frazionario usando la cosiddetta notazione esponenziale o scientifica
- **%d** – Un numero intero, **troncando** alla parte intera se l'argomento è un numero con parte frazionaria diversa da zero
- **%s** – Una stringa
- **%%** – Il carattere %
  - analogamente a quanto visto per le sequenze di escape...
- ... (altri che non vediamo)

# Indicatori di formato


- ❑ Le informazioni **opzionali** vanno inserite dopo il carattere iniziale % ma prima del carattere finale che identifica il tipo di valore
  - Un numero intero indica il **minimo** numero di posizioni che saranno occupate dal valore inserito nella stringa (se il valore necessita di un numero di caratteri maggiore di tale numero, il minimo verrà ignorato), es. %**4d**
    - Le posizioni non occupate dal valore conterranno il "carattere spazio"
    - Il valore viene allineato a destra all'interno dello spazio così riservato, a meno che il numero di spazi non sia **negativo**: in tal caso l'effettivo numero di spazi è ovviamente il valore assoluto di tale numero, ma l'allineamento è a sinistra all'interno dello spazio, es. %**-4d**
    - Utile anche per %**s**

|   |
|---|
| <code>x = "%5s" % "yz" # x = "   yz" 3 spazi</code> |
|---|
  - Un numero intero preceduto da un carattere "punto" specifica il numero di cifre "dopo la virgola" (informazione valida solo per l'indicatore %**f**); il valore verrà arrotondato, es. %**.4f**
  - Se le informazioni precedenti sono presenti entrambe, vanno scritte nell'ordine qui presentato, es. %**10.4f**
  - ... (altri che non vediamo)

# La stringa di formato...

- ❑ Ricordiamo che la **"stringa di formato"**, cioè l'operando sinistro dell'operatore di formato, è semplicemente "una stringa", quindi **non necessariamente una stringa letterale** come in tutti gli esempi precedenti
  - Tale stringa di formato può anche essere **"costruita"** durante l'elaborazione svolta dal programma, con effetti interessanti

```
x = "paperino"  
y = "pippo"  
m = max(len(x), len(y)) # m = 8  
s = "%" + str(m) + "s"  # s = "%8s"  
print(s % x)  
print(s % y)
```



```
paperino  
pippo
```

- ❑ Ricordiamo che tutto quanto si ottiene usando l'operatore di formato si potrebbe ottenere anche senza usarlo, con codice opportuno ma, in generale, molto più lungo, con cicli, anche annidati
  - Quindi, l'operatore di formato è comodo!

Funzione exit

# Funzione `exit()`

- ❑ Durante la fase iniziale di un programma, quando si verifica la validità dei dati acquisiti in input, spesso si vorrebbe terminare il programma prematuramente (in caso di dati errati)
- ❑ Questo è possibile con la funzione **`exit`** del modulo **`sys`**, la cui invocazione provoca la terminazione immediata del programma, dopo aver visualizzato il messaggio fornito come argomento (facoltativo)
  - A parte il messaggio, in pratica agisce come **`return`** nelle funzioni (che non si può usare nel programma principale)
- ❑ In mancanza di questa funzione, la porzione di programma da eseguire quando i dati sono corretti deve costituire il corpo della clausola **`else`** di tale enunciato **`if`**: scomodo, bisogna spostare tutti gli enunciati a destra...

```
from sys import exit
if ... : # errore grave
    exit("Errore grave")
# Qui else non serve, perché?
# Il programma prosegue qui sotto
# SENZA indentazione
```

```
if ... : # errore grave
    print("Errore grave")
else :
    il programma vero e proprio
    spostato a destra...
```

Il materiale necessario per il  
Laboratorio 05  
termina qui

**Lezione 19**  
**13/11/2024**  
**ore 10.30-12.30**  
**aula Ve**



*Grafica al calcolatore*

# Grafica al calcolatore

- ❑ Finora abbiamo sempre realizzato programmi che presentano all'utente un'**interfaccia di comunicazione di solo testo**
- ❑ Al contrario, la nostra esperienza quotidiana con i computer è ricca di programmi con **interfaccia grafica per l'interazione con l'utente** (**GUI**, *Graphical User Interface*), sia per l'acquisizione dei dati (magari con l'utilizzo di un mouse) sia per la **visualizzazione dei risultati dell'elaborazione**
- ❑ Iniziamo a occuparci di **questo secondo aspetto**
  - Vogliamo realizzare programmi che visualizzano informazioni grafiche all'interno di una finestra
  - Esempi
    - Simulazione del "Game of Life" di Conway
    - Visualizzazione grafica dell'andamento di una funzione di una variabile
    - Visualizzazione di una scacchiera che consenta di giocare a tris (*tic-tac-toe*) contro il computer

# Grafica in Python

- ❑ Python mette a disposizione dei programmatori numerosi moduli della libreria standard che si occupano di far interagire il programma con il **sistema di gestione delle finestre** (*window manager*, che fa parte del sistema operativo)
  - In particolare, il modulo **tkinter**
  - L'utilizzo di queste risorse, però, è un po' complesso
- ❑ Utilizzeremo un modulo che si interpone tra il programmatore e **tkinter**, rendendone più semplice l'utilizzo
  - È messo a disposizione gratuitamente dagli autori del libro "di testo" (e **non è necessario acquistare il libro**)
  - Si tratta del modulo **ezgraphics**, contenuto nel file **ezgraphics.py**, **scaricabile dal sito del corso**, oltre che da **<http://ezgraphics.org>**
    - Tale file va copiato nelle cartelle in cui si trovano i file sorgenti che lo **utilizzano**, in modo che (non facendo parte della libreria standard) l'interprete lo possa trovare (e leggere) quando viene importato

# Il modulo ezgraphics



❑ Tra le altre cose, il modulo **ezgraphics** (si legge "easygraphics") contiene la *definizione di una classe*, **GraphicsWindow**, che descrive il comportamento di "oggetti di tipo finestra grafica"

❑ Quindi i nostri programmi grafici inizieranno con

```
from ezgraphics import GraphicsWindow # oppure *
```

❑ Una *finestra grafica* è un oggetto che rappresenta una porzione rettangolare dello schermo, dotata di un bordo e di una barra superiore, contenente un titolo e alcuni pulsanti standard che dipendono dal sistema operativo

- solitamente un pulsante che elimina la finestra, uno che la "chiude" (rendendola invisibile ma attiva) e uno che ne consente il ridimensionamento

❑ Con questo enunciato si crea una finestra e la si memorizza in una variabile, per poterla poi "utilizzare"

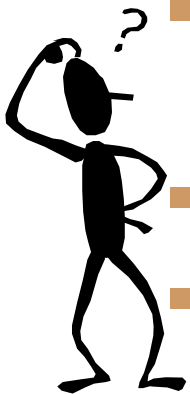
```
win = GraphicsWindow()
```

(vedremo cosa significherà *utilizzare una finestra*)

# Oggetti di tipo `GraphicsWindow`

```
win = GraphicsWindow()
```

- ❑ L'espressione `GraphicsWindow()` *sembra* l'invocazione di una funzione (senza argomenti): infatti è un nome seguito da una coppia di parentesi tonde...
- ❑ In effetti, come vedremo, è l'invocazione di un *costruttore*, una funzione *speciale* che ha il compito di costruire (e restituire) un nuovo oggetto di un determinato tipo
  - Di che tipo? In questo caso, di tipo `GraphicsWindow`
  - L'oggetto restituito va memorizzato in una variabile, per poterlo poi utilizzare
  - Cosa vuol dire "costruire un oggetto"? Vedremo...
  - Come si utilizzano gli oggetti?
    - Come le stringhe... si usano (anche) per invocare metodi!



# Oggetti di tipo `GraphicsWindow`

- ❑ La **creazione** di un **nuovo** oggetto di tipo `GraphicsWindow` (che, all'interno del programma, rappresenta una finestra sullo schermo) **provoca la visualizzazione di una nuova finestra sullo schermo!** `win = GraphicsWindow()`
- ❑ Purtroppo questa finestra scompare immediatamente, perché l'interprete passa a eseguire l'istruzione successiva, che in questo esempio semplice non esiste e, quindi, il programma termina!
  - **Quando un programma grafico termina, tutte le finestre che ha creato scompaiono**
- ❑ Come ultima istruzione di un programma grafico, dobbiamo inserire qualcosa che ne impedisca la terminazione... `win.wait()`
  - invochiamo il metodo `wait()` con una qualsiasi delle finestre create dal programma: l'interprete si blocca in attesa che tale finestra venga chiusa dall'utente (agendo sui pulsanti della barra)

# Oggetti di tipo GraphicsWindow

❑ Il più semplice programma grafico...  
che finestra crea?

```
from ezgraphics import *  
win = GraphicsWindow()  
win.wait()
```

- **Provate...** otterrete una semplice finestra rettangolare "vuota"  
(cioè bianca, ma con bordo e barra superiore standard)

❑ Che dimensione ha?

- In un programma grafico le dimensioni si misurano in *pixel*
  - Un *pixel* è un elemento grafico **elementare** dello schermo, il più piccolo "quadratino" del quale si può controllare individualmente il colore
  - Le sue **dimensioni fisiche** dipendono dalle dimensioni dello schermo e dalla sua **risoluzione**, che indica il numero di pixel che lo compongono
    - es. risoluzione 2736 x 1824, che è il numero di pixel di ciascuna riga seguito dal numero di pixel di ciascuna colonna della matrice che rappresenta lo schermo, quindi qualche milione di punti...
  - Ovviamente, a parità di dimensione fisica dello schermo, maggiore è il numero di pixel (cioè maggiore è la sua risoluzione), migliore sarà la qualità delle immagini visualizzate (altrimenti si vedono i quadratini!)

# Oggetti di tipo GraphicsWindow

- ❑ La finestra che viene creata misura 400 x 400 pixel

```
from ezgraphics import *  
win = GraphicsWindow()  
win.wait()
```

- Si chiama larghezza o ampiezza (*width*) la misura orizzontale e altezza (*height*) la misura verticale

- ❑ Fornendo **parametri al costruttore**, possiamo **creare finestre della dimensione desiderata**

```
from ezgraphics import *  
win = GraphicsWindow()  
win2 = GraphicsWindow(300, 800)  
win.wait()
```

- Il primo parametro è la larghezza, il secondo è l'altezza
  - Qui creo **due finestre** e il programma termina quando viene chiusa la prima delle due, la cosiddetta "**finestra principale**" del programma (decido io quale sia, l'importante è invocare **wait()** soltanto con quella)



# Oggetto di tipo *canvas*

```
from ezgraphics import *  
win = GraphicsWindow()  
canvas = win.canvas()  
  
...  
win.wait()
```

- ❑ La superficie **interna** a una finestra (escludendo, cioè, la barra del titolo e i bordi) si chiama *canvas* ("tela da disegno" o "pannello") ed è lì che possiamo effettivamente disegnare (il bordo, invece, è gestito dal sistema operativo)
- ❑ Con l'oggetto di tipo **GraphicsCanvas**, restituito dal metodo **canvas()** della finestra, invocheremo i metodi che consentono di disegnare
  - **drawLine** – disegna un segmento di retta
  - **drawRect** – disegna un rettangolo
  - **drawOval** – disegna un'ellisse (o un cerchio)
  - **drawText** – disegna una stringa di testo
- ❑ **Per poter disegnare in una superficie piana dobbiamo imparare un sistema di coordinate**

Tale oggetto non va creato, viene creato insieme alla finestra e restituito dal suo metodo **canvas()**

# Coordinate nel *canvas*

- Tutti i metodi di disegno si basano sulle coordinate di punti nel piano cartesiano, che però usa una convenzione un po' diversa da quella normalmente adottata dalla geometria analitica
  - **L'origine, cioè il punto di coordinate (0, 0), si trova nel vertice superiore sinistro del *canvas***
  - **Anche tutti gli altri punti del *canvas* hanno entrambe le coordinate non negative, con la prima coordinata (x) lungo la direzione orizzontale (crescente verso destra) e la seconda coordinata (y) lungo la direzione verticale (crescente verso il basso)**
  - Le dimensioni fornite al costruttore della finestra sono, in realtà, le dimensioni del suo "pannello disegnabile"
  - Se il pannello ha larghezza 100 e altezza 200, queste sono le coordinate dei suoi vertici
    - (0, 0) – vertice superiore sinistro
    - (99, 0) – vertice superiore destro
    - (0, 199) – vertice inferiore sinistro
    - (99, 199) – vertice inferiore destro

# Disegno di segmenti e rettangoli

- ❑ Per disegnare un segmento (*line*) devo indicare le coordinate dei suoi due punti estremi

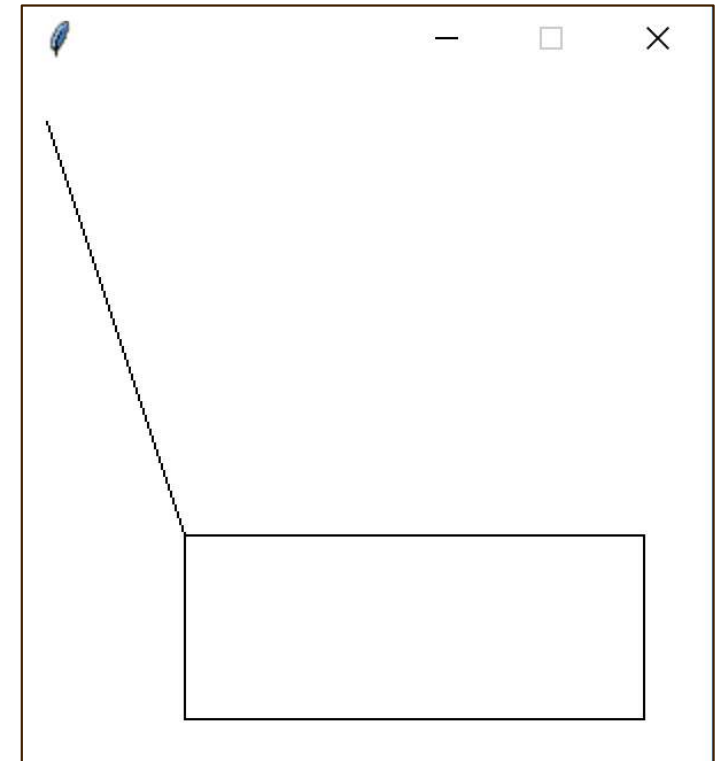
- ❑ Un singolo punto?

```
# x, y  
canvas.drawPoint(70, 200)
```

```
from ezgraphics import *  
win = GraphicsWindow(300, 300)  
canvas = win.canvas()  
canvas.drawLine(10, 20, 70, 200)  
# x1, y1, x2, y2  
win.wait()
```

- ❑ Per disegnare un rettangolo devo indicare le coordinate del suo **vertice superiore sinistro**, seguite da **larghezza** (cioè dimensione orizzontale) e **altezza** (cioè dimensione verticale)

```
# x, y, w, h  
canvas.drawRect(70, 200, 200, 80)
```



# Disegno di ellissi e cerchi

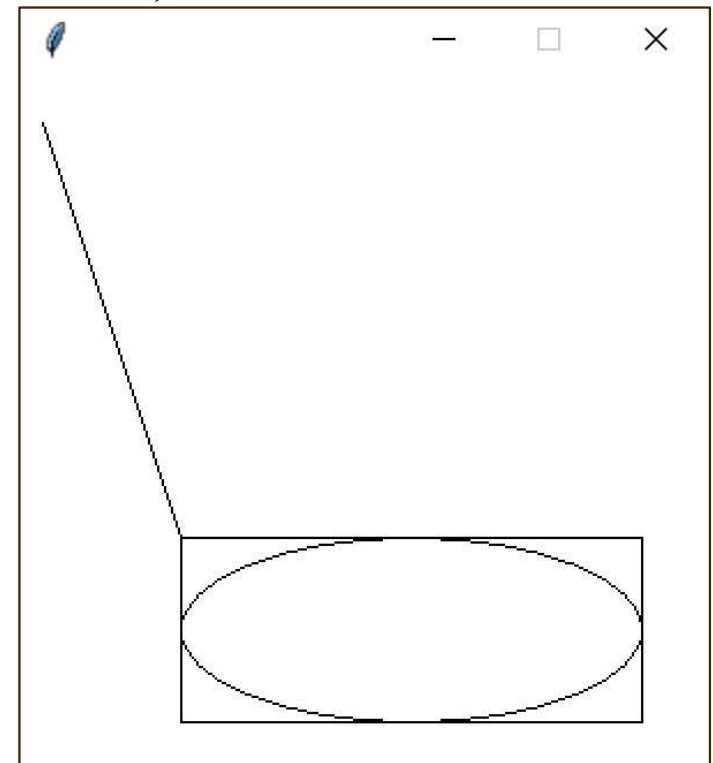
- ❑ Per disegnare un'ellisse devo indicare gli stessi parametri che servono a disegnare un rettangolo, facendo riferimento al rettangolo minimo "immaginario" che la contiene, detto *rettangolo di delimitazione*
  - Se la larghezza è uguale all'altezza (cioè il rettangolo è un quadrato), si ottiene un cerchio
  - ATTENZIONE: i primi due argomenti sono le coordinate del **vertice superiore sinistro del rettangolo di delimitazione**, NON le coordinate del centro del cerchio (o dell'ellisse)

```
from ezgraphics import *
win = GraphicsWindow(300, 300)
canvas = win.canvas()

        # x1, y1, x2, y2
canvas.drawLine(10, 20, 70, 200)

        # x, y, w, h
canvas.drawRect(70, 200, 200, 80)

        # x, y, w, h
canvas.drawOval(70, 200, 200, 80)
win.wait()
```

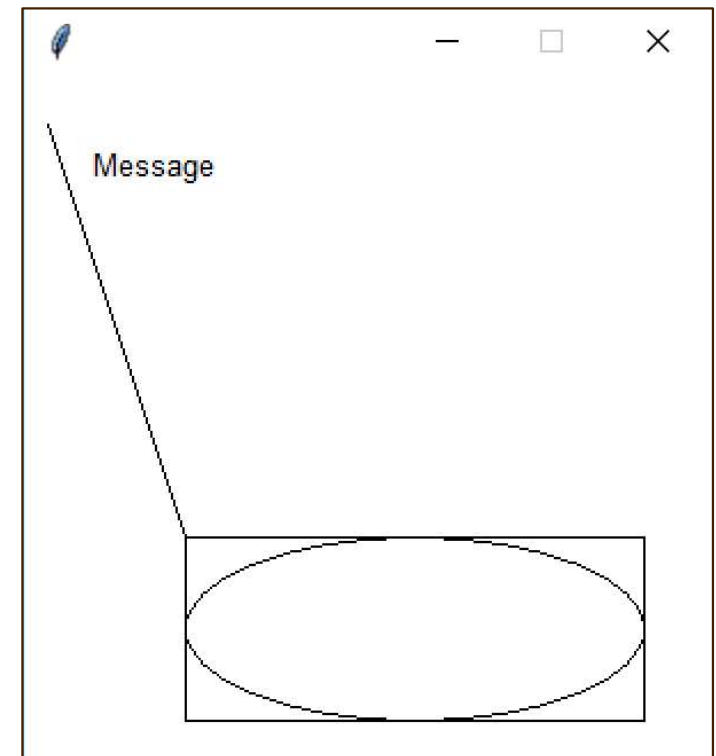


# Disegno di stringhe

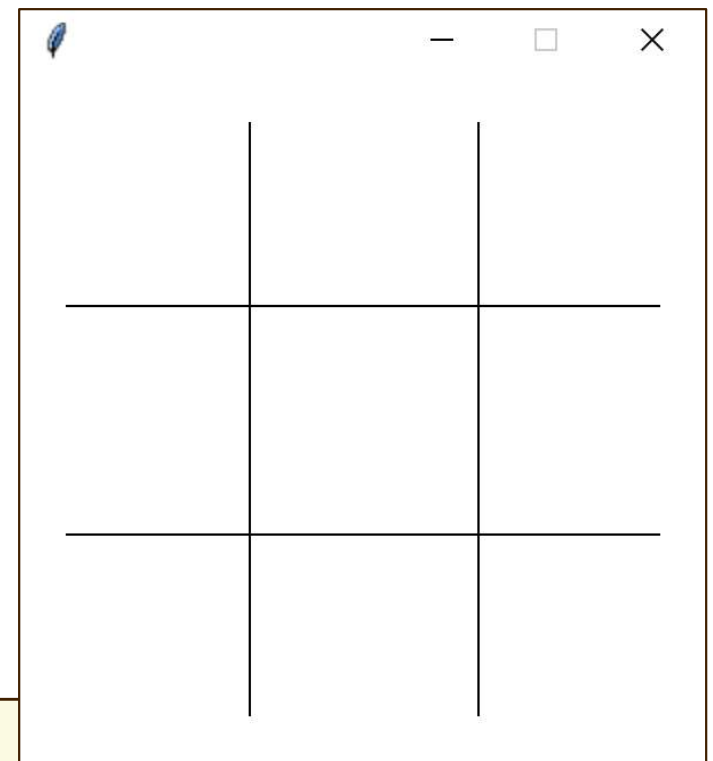
- ❑ Per inserire stringhe di testo nel disegno le devo "disegnare", usando il metodo **drawText** (non si usa **print** !)
- ❑ I suoi primi due argomenti sono le coordinate del vertice superiore sinistro del rettangolo "immaginario" che delimita la stringa
  - Le dimensioni di tale rettangolo vengono calcolate automaticamente dal metodo
  - Il posizionamento va fatto "per tentativi" ...
- ❑ Il suo terzo argomento è **la stringa da visualizzare**

```
from ezgraphics import *
win = GraphicsWindow(300, 300)
canvas = win.canvas()

        # x1, y1, x2, y2
canvas.drawLine(10, 20, 70, 200)
        # x, y, w, h
canvas.drawRect(70, 200, 200, 80)
        # x, y, w, h
canvas.drawOval(70, 200, 200, 80)
canvas.drawText(30, 30, "Message")
win.wait()
```



# Scacchiera del tris o filetto



```
from ezgraphics import *
width = height = 300 # piccola novità sintattica... utile
win = GraphicsWindow(width, height)
canvas = win.canvas()
side = width / 3 # lato di un quadrato della scacchiera
margin = width / 15 # margine rispetto al bordo esterno
# disegno completamente parametrico, buona strategia...
canvas.drawLine(side, margin, side, height - margin)
canvas.drawLine(2 * side, margin, 2 * side, height - margin)
canvas.drawLine(margin, side, width - margin, side)
canvas.drawLine(margin, 2 * side, width - margin, 2 * side)
win.wait() // capire cosa disegna ciascuna riga di codice;
           // verifica: commentate le altre 3 righe...
```

# Coordinate non valide?

- ❑ Cosa succede se vengono utilizzate **coordinate non valide**, cioè punti che NON appartengono alla finestra?
- ❑ **Nessun errore**, il disegno viene eseguito usando le coordinate fornite, visualizzando però soltanto la porzione di disegno che rientra nella finestra
  - Come se la finestra fosse "sovrapposta" a un foglio gigantesco
  - Quali sono le coordinate massime utilizzabili?  
Un esperimento?

# Esercizio

## Una nuvola di punti



# Nuvola di punti

- ❑ Vogliamo riempire la finestra di punti disposti casualmente, con una determinata densità (che è un numero compreso tra 0 e 1 e rappresenta in pratica la probabilità che un pixel sia annerito)
- ❑ Per ogni pixel della finestra, genero un numero casuale: se è minore della densità disegno quel pixel, altrimenti non lo disegno

- ❑ Devo soltanto programmare un ciclo che prenda in esame tutti i possibili pixel

```
# coordinate di un pixel
x = ...
y = ...
if random() < density :
    canvas.drawPoint(x, y)
```

- ❑ Cioè devo generare tutte le possibili coppie di coordinate che si riferiscono a punti della finestra
- ❑ **Dati due intervalli di numeri interi (anche di dimensioni diverse) come si generano tutte le possibili coppie aventi il primo elemento appartenente al primo intervallo e il secondo elemento appartenente al secondo intervallo?**

# Nuvola di punti

- ❑ **Dati due intervalli di numeri interi (anche di dimensioni diverse) come si generano tutte le possibili coppie aventi il primo elemento appartenente al primo insieme e il secondo elemento appartenente al secondo insieme?**

```
x = 0
while x < width : # potrei usare for x in range(width) ...
    # nel corpo del ciclo, x assume, uno dopo l'altro, tutti
    # i valori interi appartenenti all'intervallo [0, width)
    # cioè tutti i possibili valori della prima coordinata
    # di un punto della finestra
    ...
    x += 1
```

- ❑ Cosa devo fare nel corpo del ciclo, per ogni possibile valore della prima coordinata?
  - **Devo generare tutti i possibili valori della seconda coordinata: MI SERVE UN CICLO dentro al ciclo**

# Nuvola di punti

- ❑ Dati due intervalli di numeri interi (anche di dimensioni diverse) come si generano tutte le possibili coppie aventi il primo elemento appartenente al primo insieme e il secondo elemento appartenente al secondo insieme?
- ❑ Cosa devo fare nel corpo del ciclo, per ogni possibile valore della prima coordinata?
  - Devo generare tutti i possibili valori della seconda coordinata:  
**MI SERVE UN CICLO dentro al ciclo, cioè un ciclo annidato**

```
x = 0
while x < width :
    # nel corpo del ciclo, x assume, uno dopo l'altro, tutti
    # i valori interi appartenenti all'intervallo [0, width)
    # cioè tutti i possibili valori della prima coordinata
    # di un punto della finestra
    y = 0
    while y < height :
        # nel corpo del ciclo, y assume tutti i valori interi
        # dell'intervallo [0, height)
        ... # qui (x, y) sono le coordinate di un punto
        y += 1
    x += 1 # fare molta attenzione alle indentazioni...
```

```
# cloud.py
from ezgraphics import *
from random import random
width = height = int(input("Dimensione della finestra? "))
density = float(input("Densità (tra 0 e 1): "))
win = GraphicsWindow(width, height)
canvas = win.canvas()
win.hide() # nascondo mentre disegno, la finestra comparirà
           # alla fine, non sappiamo perché ma è più veloce
count = 0 # conto i pixel effettivamente disegnati
x = 0
while x < width : # potrei usare for x in range(width)
    y = 0
    while y < height : # potrei usare for y in range(height)
        if random() < density :
            count += 1
            canvas.drawPoint(x, y)
        y += 1
    x += 1
win.show() # ora compare la finestra disegnata
print("Densità effettiva:", count / (width * height))
win.wait()
```

# Grafica


## I colori

# I colori nei programmi grafici

- ❑ Finora i nostri programmi grafici hanno usato disegni neri su sfondo bianco, ma è possibile modificare questa impostazione predefinita
  - Innanzitutto, vediamo come si descrivono i colori all'interno di un calcolatore e, quindi, di un programma
- ❑ Ciascun pixel dello schermo può avere un colore diverso, cioè **il pixel è il più piccolo elemento dello schermo modificabile (o "colorabile") singolarmente**
- ❑ Esistono diversi standard per descrivere i colori, il sistema grafico di Python usa il **modello RGB** (*Red, Green, Blue*, cioè rosso, verde e blu)
  - Ciascun colore può essere descritto come **composizione di diverse percentuali di tre colori primari additivi:**  
il rosso, il verde e il blu



# I colori nei programmi grafici

-  □ Ciascun colore può essere descritto come composizione di diverse percentuali di tre colori primari additivi: il rosso, il verde e il blu
- In realtà non si usano percentuali, ma, per ciascun colore, valori numerici interi appartenenti all'intervallo chiuso  $[0, 255]$
  - Il valore 255 indica la presenza massima di quel colore, il valore 0 ne indica l'assenza totale
    - Di solito si usano triplette ordinate, del tipo (r, g, b):
      - il primo valore indica la quantità di rosso,
      - il secondo indica la quantità di verde e il terzo indica la quantità di blu
    - Ad esempio
      - (0, 0, 0) è il nero (assenza di tutti i colori componenti)
      - (255, 255, 255) è il bianco (ciascuna componente al massimo livello)
      - (255, 0, 0) è il rosso, (255, 255, 0) è il giallo (rosso + verde)
      - (x, x, x) è una tonalità di grigio (più scuro per valori di x minori)
    - Oppure si possono usare alcuni nomi predefiniti (stringhe):  
**"black", "yellow", "light gray"**  
<https://www.tcl.tk/man/tcl8.6/TkCmd/colors.htm>

# Il contesto grafico di un *canvas*

- Per usare i colori nel disegno all'interno di un *canvas* ci sarebbero fondamentalmente due strategie alternative
  - Per ogni operazione di disegno (**draw** . . . ), si specifica il colore da usare
  - **Si specifica il colore da usare nel canvas e tutte le successive operazioni di disegno useranno quel colore, fino alla prossima (eventuale) modifica, che, di nuovo, avrà effetto sulle operazioni future**
    - La seconda strategia è solitamente più comoda e quasi tutti i moduli di grafica, come il nostro **ezgraphics**, la usano
    - Il colore "attivo" nel canvas viene memorizzato in una proprietà del canvas stesso, che si chiama *contesto grafico* (*graphic context*), che contiene anche molte altre caratteristiche dei disegni futuri
      - Ne vedremo soltanto alcune



**Lezione 20**  
**15/11/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Il contesto grafico di un *canvas*

## □ Il contesto grafico del canvas **c** contiene

- Colore dei punti, delle linee e dei bordi delle figure piane, oltre che del testo (*outline color*), inizialmente nero
  - **c.setOutline(*color*)**
  - **c.setOutline()** rende il bordo "trasparente", per disegnare figure piane con colore interno ma senza bordo (mentre disegnare linee e punti trasparenti non ha senso, anche se è possibile)
- Colore con cui vengono riempite le figure piane (*fill color*), inizialmente nessuno (cioè le figure sono "trasparenti")
  - **c.setFill(*color*)**
  - **c.setFill()** ripristina la "trasparenza" interna delle figure piane, per i disegni successivi
- **c.setColor(*color*)** esegue entrambi i metodi precedenti
- Colore dello sfondo (*background color*), cioè delle zone del canvas che non sono state disegnate, inizialmente bianco
  - **c.setBackground(*color*)**
- ...

# Il contesto grafico di un *canvas*

- Il contesto grafico del canvas **c** contiene
  - ...
  - Spessore delle linee (*line width*), un numero intero positivo
    - **c.setLineWidth(*width*)** (inizialmente uguale a 1)
  - Stile delle linee (*line style*), che può essere *continuo* ("**solid**") o *tratteggiato* ("**dashed**")
    - **c.setLineStyle(*style*)** (inizialmente "**solid**")
  - Font con cui vengono disegnate le stringhe
    - **c.setTextFont(*family*, *size*, *style*)**
    - ***family*** può essere '**helvetica**', '**arial**', '**courier**' oppure '**times**' (inizialmente '**helvetica**')
    - ***size*** deve essere un numero intero positivo, la dimensione del font (inizialmente 10)
    - ***style*** può essere '**normal**', '**bold**' (*grassetto*), '**italic**' (*corsivo*) oppure '**bold italic**' (*grassetto corsivo*) (inizialmente '**normal**')
  - ...

# Grafica all'esame? No

- ❑ La grafica al calcolatore fa parte del programma del corso, ma NON del programma d'esame
  - **All'esame non si parlerà di programmazione grafica** (né nel questionario né nella prova di programmazione), perché il debugging di programmi grafici è un po' troppo complesso per la durata tipica della prova d'esame
- ❑ Nonostante questo, è importante imparare a utilizzare il pacchetto **ezgraphics** per svolgere le prossime esercitazioni, dove si realizzeranno alcuni programmi con interfaccia grafica che contengono, in particolare, validi esempi di cicli annidati (usati molto spesso in strutture "a scacchiera") e di progettazione mediante funzioni
  - **Non "ignore" la programmazione grafica:** non serve all'esame ma serve per prepararsi meglio su tutti gli altri argomenti!

Liste

# Problema che forse non sappiamo risolvere...

- ❑ L'utente digita una sequenza di stringhe (di lunghezza non nota a priori, fino all'introduzione di una sentinella, ad esempio una riga vuota)...
  - Questo lo sappiamo già fare...
- ❑ ... mentre legge le stringhe, il programma deve visualizzare un messaggio ogni volta che la stringa letta è uguale a **una qualsiasi delle precedenti** (per poi proseguire comunque fino alla sentinella)
  - Per risolvere questo problema non ci sono alternative: **bisogna memorizzare tutte le stringhe** e, ogni volta che ne arriva una nuova, bisogna confrontarla con tutte le precedenti
  - **Possiamo usare un "trucco"**: concateniamo le stringhe man mano che arrivano, creando un'unica stringa che le memorizza tutte; per memorizzare i punti di separazione tra una stringa e l'altra dobbiamo usare l'unico carattere che non può far parte di una stringa digitata dall'utente: "**\n**"

```

s = "" # stringa che accumulerà tutte le stringhe lette
while True :
    read = input()
    if read == "" : # sentinella: riga vuota
        break
    i = 0
    # cerco read in s, separando le stringhe componenti
    while i < len(s) :
        j = i # inizio di una delle stringhe presenti in s
        while s[i] != "\n" :
            i += 1
            if read == s[j:i] :
                print("Duplicated")
                break
            i += 1 # "salto" il \n
        # aggiungo a s la nuova stringa con il separatore \n
        s += read + "\n" # ha senso perché read non può
                        # avere un \n al suo interno

```

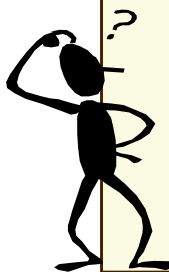
Analizzare questo programma per capire bene cosa fa... attenzione allo strano uso degli indici *i* e *j*... ci sono due cicli annidati, ma usano lo stesso indice... eseguire passo dopo passo!

- ❑ La soluzione sembra un po' complicata... forse c'è una soluzione migliore!

- ❑ Cosa ci servirebbe per risolvere in modo più "elegante" (e magari più efficiente) questo problema?
- ❑ **Non possiamo memorizzare le singole stringhe in variabili, perché non sappiamo quanti saranno**
- ❑ Ci serve **un "contenitore" di dimensioni variabili**
  - Infatti, come "trucco", abbiamo usato l'unico contenitore che già conosciamo... la stringa!
    - In realtà una stringa non ha dimensioni variabili, ma sappiamo crearne una nuova sempre più lunga... che è quasi la stessa cosa
- ❑ Come tutti i linguaggi di programmazione, anche Python mette a disposizione una **struttura** di questo tipo, **in grado di accogliere al proprio interno più dati disposti in sequenza**
  - Si chiama **lista**
    - In molti altri linguaggi si chiama **array** o vettore
    - Abbiamo già visto i contenitori, nel ciclo **for**...  
in effetti, vedremo che il ciclo **for** usa una lista!

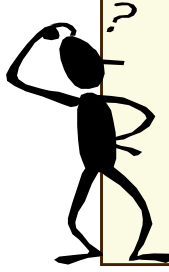


# Problema risolto con una lista



```
x = [] # lista vuota, sarà una lista di stringhe
while True :
    read = input()
    if read == "" : # sentinella
        break
    if read in x : # cerco read nella lista
        print("Duplicated")
    x.append(read) # aggiungo (in fondo) alla lista
```

- ❑ Soluzione decisamente migliore, funziona perfettamente anche con una sequenza di numeri senza bisogno di alcuna modifica se non nella gestione dei dati in input



```
x = [] # sarà una lista di numeri
while True :
    read = input()
    if read == "" : break
    n = int(read)
    if n in x : print("Duplicated")
    x.append(n)
```

Avrei potuto gestire i numeri come stringhe, ma l'operatore **in** non avrebbe riconosciuto come "uguali", ad esempio, i numeri 3 e +3, mentre questa soluzione, che converte le stringhe in numeri, lo farà

# Liste in Python

- ❑ In Python, **una lista è un contenitore di dati** (non necessariamente omogenei, anche di tipi diversi tra loro), **organizzati in sequenza**
  - Quindi in una lista (non vuota) esiste il primo dato, il secondo dato, il terzo dato... e l'ultimo dato
    - All'interno di una lista, **ciascun dato è associato a una posizione, identificata da un indice intero non negativo** (come le posizioni dei caratteri in una stringa)
      - La **prima posizione** è identificata dall'**indice zero**, e così via, con indici consecutivi crescenti verso posizioni successive nella sequenza e decrescenti verso posizioni precedenti
      - Ogni posizione è anche identificata da un indice intero negativo, con **la stessa regola vista per le stringhe**
  - Le liste sono **oggetti**, quindi le useremo (anche) per invocare metodi

# Liste in Python

- ❑ Creazione di una lista **letterale**

```
x = [2, 3, 43, 7, 2, 3]
```

- ❑ Per conoscere la lunghezza di una lista si usa la funzione predefinita **len**, come per le stringhe

```
print(len(x)) # 6
```

- La lista **x** contiene 6 elementi: ha lunghezza 6

- ❑ Si può anche creare una lista vuota, avente lunghezza zero

```
y = []
```

```
print(len([])) # 0
```

- ❑ Per ispezionare un dato della lista usiamo la stessa sintassi vista con le stringhe

```
z = x[2] # z = 43
```

```
z = x[-4] # z = 43
```

- Per gli indici validi (anche negativi) valgono le stesse regole viste per le stringhe
    - Ovviamente una lista vuota non ha indici validi
    - L'uso di un indice non valido provoca **IndexError**

# Liste in Python

❑ **Novità** rispetto alle stringhe (che sono sequenze **immutabili**)

- **Possiamo modificare il contenuto di una lista!**

```
x = [2, 3, 43, 7, 2, 3]
x[3] = 12 # MODIFICO IL QUARTO ELEMENTO
print(x) # visualizza [2, 3, 43, 12, 2, 3]
```

❑ Intanto abbiamo anche scoperto che (ovviamente...) la funzione **print** può visualizzare anche il contenuto di una lista...

- Visualizza una lista nello stesso formato che useremmo per definire il suo contenuto: elementi separati da virgole e racchiusi tra parentesi quadre
  - In realtà, non è la funzione **print** che genera la stringa da visualizzare... per farlo, **print** invoca **implicitamente** la funzione **str**, che, in generale, converte "qualsiasi cosa" in una stringa

```
x = [2, 3, 43, 7, 2, 3]
print(x) # equivale a print(str(x))
```

- È così anche per i numeri...

```
print(55) # equivale a print(str(55))
```

- In pratica, **print** visualizza soltanto stringhe! La funzione magica è **str**.

# Porzioni di lista

- ❑ Usando la stessa sintassi vista per le stringhe, possiamo **estrarre una porzione di una lista**, generando così una nuova lista

```
x = [2, 3, 43, 7, 2, 3]
y = x[1 : 4] # y contiene UNA NUOVA LISTA
print(y) # [3, 43, 7]
# se ora modifico il contenuto di x,
# il contenuto di y NON cambia
```

- Questa operazione si chiama *slicing* (fare a fette...), come per le stringhe
- ❑ Dato che le liste (diversamente dalle stringhe) sono **modificabili**, l'operazione di slicing ha anche alcune applicazioni interessanti (e complicate...), usando la sintassi slicing (cioè [ : ]) **A SINISTRA** di un'assegnazione

```
x = [2, 3, 43, 7, 2, 3]
# sostituisco una porzione con un'altra
# anche di dimensioni diverse
x[1 : 4] = [5, 1]
print(x) # [2, 5, 1, 2, 3]
x[0 : 1] = [ ] # cancello il primo elemento
print(x) # [5, 1, 2, 3]
x[len(x) : ] = [7, 8] # aggiungo in fondo
print(x) # [5, 1, 2, 3, 7, 8]
```

FATE  
ESPERIMENTI !

Sintassi poco  
chiara..  
Attenzione a non  
fare confusione

# Liste immutabili: tuple

- ❑ Spesso capita di dover usare **liste** che, per la natura dell'algoritmo che le utilizza, sono **immutabili**
  - In Python, una **lista immutabile** si chiama **tupla** (*tuple*)
  - Per creare una tupla si usano le **parentesi tonde** anziché quadre

```
x = (2, 3, 43, 7, 2, 3)
y = x[1 : 4] # quadre anche per le sotto-tuple
print(y) # (3, 43, 7), notare le tonde
x[1] = 5 # ERRORE, non è modificabile
x[1 : 3] = (5, 6) # ERRORE, non è modificabile
```

- Si potrebbe creare una tupla anche omettendo le parentesi tonde, ma è preferibile evitarlo, per chiarezza

```
x = 2, 3, 43, 7, 2, 3 # NON facciamo così
```



**Se sappiamo che una lista sarà immutabile, è meglio usare una **tupla**, per motivi di EFFICIENZA D'ESECUZIONE**

- **Tutto quello che diremo per le liste vale anche per le tuple, tranne per le operazioni che ne modificano il contenuto**

# Tuple e stringhe

- ❑ Si potrebbe pensare che una stringa sia, in pratica, una tupla di caratteri
- ❑ **Dal punto di vista del contenuto informativo è vero, ma dal punto di vista funzionale questi due tipi di dati differiscono per operatori, funzioni e metodi applicabili**

```
s = "abc"
t = ("a", "b", "c") # stessa informazione di s...
print(s) # visualizza abc
print(t) # visualizza ('a', 'b', 'c')
print(s.upper()) # visualizza ABC
print(t.upper()) # ERRORE, questo metodo non
                  # è applicabile a tuple
```

- ❑ In pratica, la stringa è "una tupla che può contenere solo caratteri" e, in quanto tale, ha **metodi specifici**
  - Come se fosse una tupla specializzata... ma tecnicamente NON è una tupla
  - Analogamente, una tupla di caratteri NON è tecnicamente una stringa

**Lezione 21**  
**19/11/2024**  
**ore 10.30-12.30**  
**aula Ve**



# Scansione di una lista o di una tupla

- ❑ Per scandire il contenuto di una lista/tupla si può procedere in due modi diversi (dipende da ciò che serve...)
  - Un ciclo a contatore (**for** o **while**) che usi tutti gli indici validi nella lista, come abbiamo visto per le stringhe (anche a ritroso)

```
x = [2, 3, 43, 7, 5, 3]
for i in range(len(x)) :
    fa qualcosa con x[i] e/o con i
i = 0
while i < len(x) :
    fa qualcosa con x[i] e/o con i
    i += 1
```
  - Un ciclo **for** che scandisca gli elementi della lista **senza usare esplicitamente gli indici**, perché una lista/tupla può svolgere direttamente il ruolo di **contenitore nel ciclo**

```
x = [2, 3, 43, 7, 5, 3]
for v in x :
    fa qualcosa con v
    # non ci sono indici
```
- ❑ Ovviamente **la prima soluzione è più flessibile**, ma quando la seconda è sufficiente... è più semplice (es. calcolare il prodotto di tutti gli elementi di una lista di numeri...)

# Concatenazione/Replicazione di liste/tuple

- ❑ Due liste possono essere **concatenate** usando l'operatore **+**, che **genera una nuova lista**

```
x = [2, 3]
y = [1, 7]
z = x + y # z = [2, 3, 1, 7]
# x e y non vengono modificate
```

- ❑ Analogamente si concatenano tuple (generando una tupla), ma **non si concatena una lista e una tupla**

- Per concatenare una lista e una tupla generando una lista, devo prima "trasformare" la tupla in lista, usando la funzione predefinita **list**
- Viceversa, si usa la funzione **tuple**
- Le funzioni **list** e **tuple** accettano, come argomenti, sequenze di ogni tipo (**stringhe**, liste, tuple e altre che vedremo)

```
x = [2, 3]
y = (1, 7)
z = x + list(y)
# z = [2, 3, 1, 7]
```

```
x = [2, 3]
y = (1, 7)
z = tuple(x) + y
# z = (2, 3, 1, 7)
```

```
y = list("ab")
# y = ['a', 'b']
```

- ❑ Come per le stringhe, è applicabile l'**operatore di replicazione (\*)**, molto comodo per creare una lista/tupla di elementi identici (ad esempio una lista di zeri)

```
x = [0] * 12
print(len(x)) # 12
y = [2, 4] * 3
# y = [2, 4, 2, 4, 2, 4]
```

# Il metodo `append` per liste

- ❑ Il metodo **`append`** aggiunge un nuovo elemento **in fondo** a una lista, aumentandone la lunghezza di un'unità

```
x = [3, 2, 3]
x.append(7)
# x = [3, 2, 3, 7]
```

- ❑ Esempio: acquisizione di una lista di dati
  - in questo caso stringhe, ma potrebbero essere numeri interi o altri dati

```
SENTINEL = "STOP"
x = [ ] # lista inizialmente vuota
while True :
    s = input("Per finire scrivere " + SENTINEL)
    if s == SENTINEL :
        break
    x.append(s) # come x[len(x):] = [s] ma più chiaro
               # come x += [s] ma più efficiente
# a questo punto del programma x contiene le stringhe lette,
# nell'ordine in cui sono state acquisite:
# la prima in posizione 0, la seconda in posizione 1...
#
# osservate che la sentinella NON entra nella lista
# ma potremmo inserirla, come ultimo elemento:
# basta anticipare l'enunciato append, prima di if
```

(ovviamente?) **`append`**  
non funziona con tuple

# Operatori di confronto tra liste/tuple

- ❑ Gli operatori di uguaglianza e diversità funzionano correttamente con liste e tuple: due liste/tuple sono uguali se e solo se hanno la stessa lunghezza e hanno elementi identici in posizioni corrispondenti

```
x = [2, 3]
y = [2, 3]
if x == y :
    print("OK") # OK
```

- **Liste e tuple aventi lo stesso contenuto sono DIVERSE**

- ❑ Gli operatori di confronto < e > funzionano in modo "ragionevole", utilizzando **un'estensione ovvia dell'algoritmo di confronto lessicografico** tra i dati in sequenza

- Ovviamente funzionano anche gli operatori <= e >=

```
x = [2, 3]
y = [2, 1]
if y < x :
    print("OK") # OK
```

- Se **i tipi di dati non sono compatibili**, si verifica un errore

```
x = [2, 3]
y = ["1", 4]
if y < x : ... # ERRORE
```

# Operatori `in` e `not in` per liste/tuple

```
friends = ["Alan", "Helen", "Mike" ]
if "Alan" in friends : print("OK") # OK
if "Goofy" not in friends : print("OK") # OK
if not ("Goofy" in friends) : print("OK") # OK
# la prima forma, not in, è forse più naturale...
# in ogni caso sono equivalenti
if "ele" not in friends : print("OK") # OK
# in NON verifica sottostringhe di elementi...
x = (2, 5, 3, 7) # con liste di ogni tipo di dato
if 3 in x : print("OK") # OK
a = 4
if a + 1 in x : print("OK") # OK
# dal fatto che l'enunciato precedente
# visualizza OK deduco che l'operatore in
# ha una priorità inferiore rispetto
# all'operatore di addizione... in qualche
# tabella potrò recuperare questa informazione,
# ma lo si può imparare facendo esperimenti,
# usando l'interprete in modalità interattiva...
```

Come si risolvono gli stessi problemi usando cicli di scansione anziché questi operatori?

# Operatori **in** e **not in** per stringhe

- ❑ Gli operatori **in** e **not in** funzionano anche quando il secondo operando è una stringa (anziché una lista/tupla)
  - in questo caso, questi operatori vogliono due stringhe come operandi, una a sinistra e una a destra, e producono come risultato il valore booleano **True** se e solo se la stringa di sinistra è (o, rispettivamente, non è) una sottostringa della stringa di destra

```
s = "topolino e minnie"  
if "lino" in s : print("OK")      # OK  
if "xx" not in s : print("OK")   # OK
```

- Più precisamente, **s1 in s2** è un'espressione che vale **True** se e solo se esiste almeno una coppia di numeri interi, **x** e **y**, tali che **s1 == s2[x : y]**
  - Ricordando le proprietà dell'estrazione di sottostringa:  
" " in "qualsiasi" vale **True**  
Ricordando le proprietà dell'estrazione di sottostringa:  
" " in " " vale **True**
- L'utilità di questi operatori risulta evidente pensando a come si risolvono gli stessi problemi usando cicli di scansione

# Funzioni che elaborano liste/tuple

## □ Altre utili funzioni predefinite

- **sum** restituisce la somma dei valori contenuti in una lista/tupla di **valori numerici**

```
x = sum((3, 1, 2)) # x = 6
```

- non funziona con liste/tuple di stringhe (cioè non concatena stringhe...)
- se è vuota restituisce zero

```
x = sum([]) + sum(()) # x = 0
```

- **min** restituisce il minimo tra i valori numerici contenuti nella lista/tupla, che non deve essere vuota (ma può anche avere un solo valore)

```
x = min([3.2, 1, 2]) # x = 1
```

- funziona anche con una lista/tupla di stringhe, usando il confronto lessicografico
- funziona anche con una stringa, che viene automaticamente convertita in una tupla di singoli caratteri

```
x = min(["aca", "ab"]) # x = "ab"
```

```
x = min("zxaf") # x = "a"
```

- **analogamente, max** restituisce il valore massimo

- ## □ Osserviamo, quindi, che **una funzione può ricevere una lista/tupla**, ma lo sapevamo già... abbiamo visto **print**, **str**...

# Liste/tuple come argomenti di funzioni

- ❑ Come abbiamo visto, una lista o una tupla, **come qualsiasi altro tipo di dato**, può essere fornita a una funzione come argomento
  - Quindi anche a una funzione progettata da noi! Emuliamo la funzione predefinita **sum**

```
def mysum(values) : # come sum
    total = 0
    for element in values :
        total += element
    return total
```

```
x = [1, 3, 5, 4, 7]
print(mysum(x)) # 20
x = (1, 3, 5, 4, 7)
print(mysum(x)) # 20
```

- ❑ Per il momento, **immaginiamo che le liste, quando vengono passate a una funzione, siano immutabili come le tuple**
  - In realtà le liste sono modificabili dalla funzione che le riceve, ma ci manca qualche dettaglio che vedremo più avanti
  - Ovviamente le tuple sono sempre immutabili, anche quando vengono passate a una funzione





# Liste/tuple come valori restituiti

- Una funzione può restituire **un solo dato, ma di qualsiasi tipo**: quindi anche una lista o una tupla

- Non ci meravigliamo... conosciamo già funzioni che restituiscono una stringa, che è un contenitore... anche se di dati omogenei (solo caratteri)

- Un'osservazione sul codice di questa funzione: dopo l'esecuzione dell'enunciato **break**, l'unico altro enunciato eseguito è **return x**

- Possiamo "semplificare" il codice!  
Anche se non è essenziale
- Ragionare sull'esempio...

```
def inputStrings(sentinel) :  
    x = []  
    while True :  
        s = input()  
        if s == sentinel :  
            break  
        x.append(s)  
    return x  
  
ss = inputStrings("STOP")
```

```
def inputStrings(sentinel) :  
    x = []  
    while True :  
        s = input()  
        if s == sentinel :  
            return x  
        x.append(s)  
  
ss = inputStrings("")  
# riga vuota come sentinella
```

**Tabelle o matrici**

# Tabelle o matrici

- ❑ Quando i dati da elaborare si presentano sotto forma di **tabella o matrice** (ad esempio, le celle del "mondo" del *Game of Life* di Conway), ci piacerebbe avere, all'interno del programma, **una struttura di memorizzazione con** le medesime caratteristiche, cioè **un certo numero di righe e un certo numero di colonne**
- ❑ Python **non** ha una struttura **specific**a per memorizzare tabelle o matrici, perché sfrutta la seguente osservazione
  - **Una tabella o matrice è una lista/tupla di righe (o di colonne) e ciascuna riga (o colonna) è, a sua volta, una lista/tupla di dati**
- ❑ Dato che, in Python, **i singoli elementi di una lista/tupla possono essere dati di tipo qualsiasi**, possono anche essere liste/tuple!
  - **Si usa una diversa lista/tupla per contenere i dati di ciascuna singola riga, poi una lista/tupla per contenere tutte tali liste/tuple che rappresentano le singole righe**
    - oppure, analogamente, una lista/tupla di "colonne", ciascuna delle quali viene memorizzata in una lista/tupla
  - Una singola lista/tupla è **una riga o una colonna**: dipende da come consideriamo i dati, al programma "non interessa"... ci deve pensare il programmatore

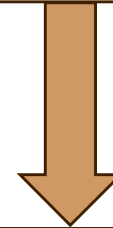
# Tabelle o matrici

- ❑ Due tabelle con tre righe e cinque colonne

```
x = [  
    [2, 4, 6, 8, 10],  
    [3, 6, 9, 12, 15],  
    [4, 8, 12, 16, 20]  
]
```

(oppure tre  
colonne e  
cinque righe...)

```
y = [  
    for i in range(3) :  
        y.append([0]*5)  
    print(y)
```




```
[ [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0] ]
```

- ❑ Cos'è `x[0]` ? È il primo elemento della lista `x`, cioè la lista che contiene le prime cinque potenze di 2: `[2, 4, 6, 8, 10]`
  - Quindi `len(x)` è il numero di righe di questa tabella (**NON** il numero di celle della matrice) e `len(x[0])` è il numero di colonne (o viceversa...), che, ovviamente, è anche uguale a `len(x[1])` e `len(x[2])`
- ❑ Come si accede ai singoli elementi della tabella/matrice?
  - `x[0]` è la **prima** riga, quindi `x[0][2]` è il **terzo** elemento della **prima** riga, e così via

# Stampare una matrice

```
x = [  
    [2, 4, 6, 8, 10],  
    [3, 6, 9, 12, 15],  
    [4, 8, 12, 16, 20]  
]  
  
# si può fare in vari modi...  
for i in range(len(x)) : # con indici  
    # x[i] è la riga i-esima  
    for j in range(len(x[i])) :  
        print(str(x[i][j]) + " ", end="")  
    print()  
print() # riga vuota separatrice  
for i in range(len(x)) : # con indici  
    row = x[i] # riga i-esima  
    for j in range(len(row)) :  
        print(str(row[j]) + " ", end="")  
    print()  
print() # riga vuota separatrice  
for row in x : # senza indici  
    for element in row :  
        print(str(element) + " ", end="")  
    print()
```

Oppure `len(x[0])`



|   |   |    |    |    |
|---|---|----|----|----|
| 2 | 4 | 6  | 8  | 10 |
| 3 | 6 | 9  | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |

|   |   |    |    |    |
|---|---|----|----|----|
| 2 | 4 | 6  | 8  | 10 |
| 3 | 6 | 9  | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |

|   |   |    |    |    |
|---|---|----|----|----|
| 2 | 4 | 6  | 8  | 10 |
| 3 | 6 | 9  | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |

Il materiale necessario per il  
Laboratorio 06  
termina qui

**Lezione 22**  
**20/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

**Attenzione...**



# Attenzione...

- ❑ Scambio del contenuto di due posizioni in una lista: serve **una variabile temporanea!**

```
x = [...] # una lista
i = ...
j = ...
temp = x[i]
x[i] = x[j]
x[j] = temp
```

- ❑ Come, in generale, per lo scambio del contenuto di due variabili

```
a = ...
b = ...
temp = a
a = b
b = temp
```

```
x = [...]
i = ...
j = ...
# NON IN QUESTO MODO !
x[i] = x[j]
x[j] = x[i]
# ERRORE (LOGICO): cosa succede?
# errore di difficile diagnosi...
```

Attenzione all'**ordine** in cui si scrivono le assegnazioni per fare lo scambio

- ❑ Per molti aspetti, una posizione in una lista si comporta come una singola variabile dello stesso tipo
  - cioè, ad esempio, in una lista **x** di stringhe, **x[unIndice]** si comporta come una singola variabile contenente una stringa

# Tupla letterale di lunghezza unitaria

- ❑ C'è un "piccolo problema sintattico" nella definizione di una **tupla letterale di lunghezza unitaria**, perché l'interprete la "confonde" con una espressione tra parentesi

```
x = ()          # tupla letterale vuota
print(len(x))  # 0
x = (5, 2)      # tupla di lunghezza due
print(len(x))  # 2
x = (5)         # tupla di lunghezza unitaria?
print(len(x))  TypeError: object of type 'int' has no len()
```

- ❑ Dobbiamo usare un "trucco": aggiungiamo **una virgola dentro le parentesi** (magari, per metterla in evidenza, aggiungiamo anche uno spazio dopo la virgola)
- ❑ Il problema non c'è con le liste, perché le parentesi quadre non generano confusione

```
x = (5, )
print(len(x))  # 1
```

```
x = [5]
print(len(x))  # 1
```

**Altri metodi che operano  
su liste e tuple**

# Metodi che operano su liste/tuple

- ❑ Liste e tuple sono oggetti, quindi ci sono **metodi** con cui elaborarle
  - Naturalmente i metodi **modificatori** non saranno disponibili con le tuple
- ❑ Come abbiamo visto, per verificare la presenza di uno specifico elemento in una lista possiamo usare l'operatore **in**
  - Se, però, voglio sapere **in quale posizione** si trova l'elemento all'interno della lista, posso usare il metodo **index**
    - Il metodo **index** esiste anche per le stringhe e funziona nello stesso modo
    - Se l'elemento cercato è presente più volte, viene restituito l'indice (non negativo) minimo
  - Se l'elemento **non** è presente nella lista, il metodo **index** genera un errore **ValueError**, quindi spesso lo si usa in questo modo, che (ovviamente?) garantisce che non si verifichino errori
  - Il metodo **index** può anche ricevere un secondo parametro, un numero intero che indica l'indice da cui iniziare la ricerca (proseguendo, poi, verso indici crescenti); se tale secondo parametro è assente, viene assunto uguale a zero
    - Forma utile se la lista contiene più elementi uguali a quello cercato e voglio trovare la posizione del secondo o del terzo... esempio nella slide successiva

```
x = [5, 6, 6]
print(x.index(6)) # 1
```

```
x = [5, 6, 7]
if 6 in x :
    print(x.index(6))
```

# Esempio

- Contare quante volte un elemento è presente in una lista/tupla

```
x = [1, 2, 3, 2, 1, 2]
element = ... # elemento da cercare
count = 0 # contatore degli elementi trovati
i = 0
while i < len(x) :
    if element in x[i : ] : # ce n'è almeno uno da "qui" in avanti?
        # sicuramente ce n'è (almeno) un altro
        count += 1
        # dov'è? lo cerco, a partire dalla posizione i
        j = x.index(element, i) # uso index con due parametri
        # so che x[j] è uguale a element, inutile verificarlo...
        print("DEBUG: i=%d j=%d" % (i, j))
        i = j+1 # ora voglio "saltare" l'elemento appena contato,
                # che è in posizione j, quindi riprenderò
                # la ricerca dalla posizione successiva
                # (nella prossima iterazione)
    else :
        break # non ce ne sono più... inutile cercare ancora
        # Non c'è il "solito" i+= 1... Perché?
        # Non è un ciclo a contatore... la variabile i procede "a salti"
print(count) # FARE L'ANALISI passo dopo passo DI QUESTO ESEMPIO
```

□ Oppure...

# Esempio (prosegue)

❑ Contare quante volte un elemento è presente in una lista/tupla...

❑ Diversa soluzione,  
molto più semplice

```
element = ... # da cercare
for v in x :
    if v == element :
        count += 1
# più semplice...
```

❑ Soluzione ancora diversa, semplicissima,  
perché scopriamo l'esistenza del metodo  
**count** per liste/tuple che risolve proprio  
questo problema!

```
element = ... # da cercare
count = x.count(element)
```

❑ Il metodo **count** esiste anche per stringhe!  
Conta quante sono le **sottostringhe**  
uguali a quella cercata (eventualmente zero)

```
s = "mototopo"
print(s.count("to")) # 2
print(s.count("o")) # 4
print(s.count("x")) # 0
```

- Fare esperimenti nel caso  
di sottostringhe  
sovrapposte

```
print("xxxx".count("xx")) # 2 0 3 ? 2
print("xxxxx".count("xx")) # ??? 2
```

Altri metodi modificatori  
che operano su liste  
(e, ovviamente, NON su tuple)

# Metodi modificatori che operano su liste

- ❑ Abbiamo visto il metodo **append** per aggiungere un nuovo elemento **in fondo** a una lista (ovviamente NON a una tupla...)

- ❑ Il metodo **insert** consente di aggiungere un elemento a una lista in qualsiasi sua posizione!

```
x = [1, 2, 3]
x.insert(1, 5)
# x = [1, 5, 2, 3]
```

Riceve due argomenti, nell'ordine:

l'**indice** in cui posizionare il nuovo

elemento e, appunto, **il nuovo elemento**

```
x = [...]
x.insert(index, element)
equivale a
x[index:index] = [element]
ma è più chiaro e più efficiente
```

- Tutti gli elementi aventi indice non minore di **indice** vengono spostati nella posizione immediatamente successiva a quella che occupano, cioè si "crea spazio" per il nuovo elemento
- Se **indice** è uguale alla lunghezza della lista, l'effetto è identico a quello di **append**: nuovo elemento aggiunto in fondo, senza spostare altri elementi
  - Quindi, in pratica, **append** non è altro che una scorciatoia per l'invocazione di **insert** con il primo parametro uguale alla lunghezza della lista
- Se **indice** è maggiore della lunghezza della lista dovrebbe esserci un errore (perché è un'operazione priva di senso), invece tutto va come se **indice** fosse uguale alla lunghezza della lista [STRANEZZA...]
- In ogni caso, la dimensione della lista aumenta di un'unità



# Metodi modificatori che operano su liste

- ❑ Il metodo **remove** elimina dalla lista l'elemento ricevuto, diminuendo di un'unità la lunghezza della lista

```
x = [1, 2, 3, 7]
x.remove(3)
# x = [1, 2, 7]
```

- Se ci sono più elementi uguali a quello da rimuovere, viene eliminato quello avente indice minore (cioè "quello più a sinistra")
- Se l'elemento non è nella lista, si verifica un errore

**ValueError**

```
x = [...]
x.remove(element)
equivale a
index = x.index(element)
x[index:index+1] = []
ma è più chiaro e più efficiente
```

# Metodi modificatori che operano su liste

- Il metodo **pop** riceve come argomento un **indice** ed elimina dalla lista l'elemento che si trova in posizione **indice**, diminuendo di un'unità la lunghezza della lista

```
x = [1, 2, 3, 7]
x.pop(1)
# x = [1, 3, 7]
```

(come al solito, sono validi anche indici negativi)

- Tutti gli elementi aventi indice maggiore di **indice** vengono spostati nella posizione immediatamente precedente a quella che occupano (cioè "si chiude il buco" che si è creato nella lista)
- Se l'indice non è valido nella lista, si verifica un errore **IndexError**
- Il metodo **pop** restituisce il valore eliminato (che, come in questo esempio, può essere ignorato dall'invocante), per eventuali verifiche

```
x = [...]
x.pop(index)
equivale a
x[index:index+1] = []
ma è più chiaro e più efficiente
```

- **Regola generale: non è necessario utilizzare il valore restituito da un metodo/funzione, lo si può ignorare**

# Metodi modificatori che operano su liste

- ❑ Il metodo **sort** ordina la lista in modo che i suoi elementi siano non decrescenti al crescere dell'indice

```
x = [1, 7, 3, 2]
x.sort()
# x = [1, 2, 3, 7]
```

- Spesso diciamo brevemente "**lista in ordine crescente**" anche se, in generale, è meglio dire "in ordine non decrescente", perché possono esserci elementi duplicati
- Se gli elementi sono stringhe, il metodo **sort** usa (automaticamente) il confronto lessicografico

- ❑ Il problema dell'ordinamento dei dati all'interno di una sequenza è, come vedremo, di cruciale importanza nella programmazione



- Il libro dedica un intero capitolo all'argomento
- Vedremo anche perché è così importante...

# Esempi di algoritmi di ricerca in liste/tuple

# Esempi

- ❑ Sappiamo che per calcolare il valore massimo in una lista/tupla possiamo usare la funzione **max**
  - Trova anche la "stringa massima" in una lista/tupla di stringhe, nel senso di stringa che si troverebbe alla fine della sequenza se questa venisse ordinata **secondo il criterio lessicografico**
  - Ma se, invece, volessimo trovare **la stringa più lunga**?

```
x = ... # una lista/tupla di stringhe
longest = x[0] # ipotesi di partenza: Questa è la più
              # lunga... vista finora!
for i in range(1, len(x)) : # se 0 funziona lo stesso...
    if len(x[i]) > len(longest) : # trovata una più lunga
        longest = x[i] # aggiorno "la più lunga finora"
print(longest)
```

- Ovviamente un algoritmo analogo trova la stringa più corta (importante: cambiare nome alla variabile **longest**!)
- E se volessimo trovare **la posizione della stringa più lunga**?
  - La soluzione di questo secondo problema risolve anche il primo...

# Esempi

- ❑ Cerchiamo **la posizione della stringa più lunga**, poi visualizziamo anche la stringa più lunga

```
x = ... # una lista/tupla di stringhe
pos = 0 # ipotesi di partenza...
for i in range(1, len(x)) :
    if len(x[i]) > len(x[pos]) :
        pos = i # aggiorno
print(pos)
print(x[pos])
```

Cosa succede con  
>= al posto di > ?

- ❑ Oppure, modifichiamo la soluzione precedente... trovata la stringa più lunga, ne cerchiamo la posizione usando il metodo **index**

```
x = ...
longest = x[0]
for i in range(1, len(x)) :
    if len(x[i]) > len(longest) :
        longest = x[i]
print(x.index(longest))
print(longest)
```

- ❑ Cerchiamo (la posizione di) un elemento avente specifiche caratteristiche
- ❑ Ad esempio, una stringa avente **iniziale maiuscola**
  - Nello specifico, quella di indice minimo, cioè "la prima da sinistra"

```
x = ... # una lista/tupla di stringhe
found = False
pos = 0
while pos < len(x) and not found :
    # sintassi... se x[pos] è una stringa, x[pos][0] è...
    if x[pos][0].isupper() :
        found = True # potrei usare break senza variabile?
    else :
        pos += 1
# usando break, non saprei se ho trovato oppure no!
# in realtà ragionando un po' ... (nella slide successiva)
if found :
    print("Found in", pos)
else :
    print("Not found")
```

Questo algoritmo si chiama  
*ricerca lineare* o  
*ricerca sequenziale*

## □ Ragioniamo...

- Se il ciclo termina con **break**, che valore può avere **pos**?
- Se, invece, il ciclo termina senza eseguire **break** (e, quindi, non ha trovato un elemento che soddisfi la condizione cercata), che valore ha **pos**?

```
x = ... # una lista/tupla di stringhe
pos = 0
while pos < len(x) : # senza la variabile booleana found
    if x[pos][0].isupper() :
        break
    pos += 1 # else non serve più: perché?
# se il ciclo è terminato per l'esecuzione di break
# certamente pos è un indice valido nella lista/tupla,
# altrimenti pos è diventato uguale a len(x)
if pos < len(x) :
    print("Found in", pos)
else :
    print("Not found")
```



# Algoritmo di ricerca lineare

- ❑ L'algoritmo di **ricerca lineare** o **ricerca sequenziale** in una lista/tupla esamina uno dopo l'altro, **in sequenza**, gli elementi della lista/tupla (ordinatamente dall'inizio alla fine)
  - Termina la ricerca quando trova il primo elemento che soddisfa un criterio prefissato
    - Se procede **dalla fine verso l'inizio** della lista/tupla, trova **l'ultimo** elemento che soddisfa il criterio, ossia quello avente indice massimo
  - Oppure termina quando arriva alla fine della scansione senza successo
    - In tal caso siamo certi che non ci sia alcun elemento che soddisfa il criterio
  - Se il criterio è, banalmente, "essere uguale a...", è sufficiente utilizzare l'operatore **in** (per sapere se ce n'è almeno uno), seguito dal metodo **index** (per trovare la posizione)

```
x = ... # lista o tupla
target = ...
if target in x :
    print("Found in", x.index(target))
else :
    print("Not found")
```

# Esempi

- ❑ Se, invece, vogliamo **contare quanti** elementi della lista/tupla soddisfano un criterio prefissato, portiamo a termine la ricerca lineare, senza fermarci al primo successo

```
x = ... # una lista/tupla di stringhe
count = 0
pos = 0
while pos < len(x) :
    if x[pos][0].isupper() :
        count += 1
    pos += 1
```

```
x = ...
count = 0
# pos in realtà non mi serve...
for element in x:
    if element[0].isupper() :
        count += 1
```

- ❑ Con una piccola modifica è possibile **creare (se mi serve) una lista contenente soltanto gli elementi che soddisfano il criterio**

```
x = ...
ok = [ ]
for element in x:
    if element[0].isupper() :
        ok.append(element)
# ora count = len(ok)
```

**Lezione 23**  
**22/11/2024**  
**ore 16.30-18.30**  
**aula Ve**

**Ancora sulle funzioni**

# Commenti nella definizione di funzione

```
## Acquisisce in input un numero positivo.  
# ... spiegazioni ulteriori...  
# @param message il messaggio per l'utente  
# @return il numero acquisito  
#  
def inputPositiveInteger(message) :  
    while True :  
        n = int(input(message))  
        if n > 0 : break  
    return n
```

- ❑ Ci sono vari stili "standard" per descrivere **in un commento** il compito svolto da una funzione
  - Esistono programmi che leggono i file sorgente alla ricerca di questi commenti e generano automaticamente file HTML con la documentazione della funzione, simili a quella della libreria standard
- ❑ Usiamo quello rappresentato qui
  - Una prima riga (con **##**) che descrive sinteticamente il compito svolto
    - Poi, eventuali altre righe che illustrano meglio il compito svolto
  - Una riga per ogni eventuale parametro, con la parole chiave **@param** che precede il nome del parametro e una sua descrizione sintetica
  - Un'eventuale riga **@return** che descrive l'eventuale valore restituito

# Progettazione mediante *stub*

- ❑ Spesso quando si inizia un progetto ci si vuole concentrare sulla logica principale del programma, senza doversi preoccupare dei dettagli (come può essere l'acquisizione di un numero intero)
  - Purtroppo, senza il codice che svolge anche le funzioni più elementari, non si può collaudare il programma completo
  - Una soluzione può essere quella di progettare inizialmente funzioni che svolgono il loro compito in modo "essenziale", senza occuparsi dei dettagli, dei quali ci si occuperà in seguito
    - Queste funzioni vengono chiamate *stub* ("mozziconi") o *mock* ("imitazione")

```
def main() :  
    n1 = inputPositiveInteger("Numero 1: ")  
    n2 = inputPositiveInteger("Numero 2: ")  
    n3 = inputPositiveInteger("Numero 3: ")  
    ... # voglio concentrarmi prima su questa sezione...  
  
def inputPositiveInteger(message) : # stub o mock  
    print("DEBUG Ricordati di finire inputPositiveInteger")  
    return int(input(message)) # il minimo indispensabile  
  
main()
```

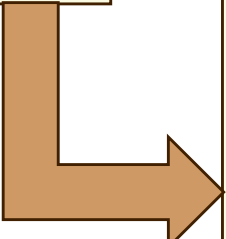
**Argomenti sulla riga di  
comando**

# Argomenti sulla riga di comando

- ❑ Quando si esegue un programma Python, è anche possibile fornire dati al programma scrivendoli sulla riga di comando
- ❑ Tali parametri vengono letti dall'interprete Python (che li riceve dal sistema operativo) e trasformati in **una lista di stringhe** che si chiama **argv** (*vettore di argomenti*)
  - Per poter accedere alla variabile predefinita **argv** occorre importarla dal modulo **sys**
  - Il primo parametro (**argv[0]**) è sempre il nome del file sorgente

C:\python x.py 5 72

- ❑ Molto comodo quando i dati che servono sono pochi (i dati non vengono chiesti all'utente, il quale deve sapere che vanno forniti)



```
# x.py
from sys import argv
if len(argv) == 3 :
    # argv[0] vale "x.py"
    n1 = int(argv[1])# 5
    n2 = int(argv[2])# 72
else :
    n1 = int(input(...))
    n2 = ...
...
```



# Argomenti sulla riga di comando

- ❑ Se vogliamo fornire come argomento una stringa contenente spazi, dobbiamo racchiuderla tra doppie virgolette (non semplici apici, non stiamo scrivendo codice Python... stiamo comunicando con il sistema operativo!)
  - Si possono usare sequenze di escape

```
C:\python x.py "M D"
```

```
# x.py
from sys import argv
print(argv)
# ['x.py', 'M D']
```

```
C:\python x.py "M \"D\""
```

```
# x.py
from sys import argv
print(argv)
# ['x.py', 'M "D"']
```

- In IDLE si usa *Run customized* anziché *Run Module*

# Redirezione di input e output

# Acquisizione di dati in input

- ❑ Quando si utilizzano programmi che acquisiscono una sequenza di dati in ingresso (ad esempio, per sommarli o per trovarne il valor medio), è molto scomodo dover scrivere l'intera sequenza sulla tastiera, magari ricopiando il contenuto di un file...
  - Sarebbe decisamente meglio istruire il programma perché legga il file!
  - Situazione molto simile quando si collauda qualsiasi programma che richiede molti dati: ogni volta che lo vogliamo eseguire, dobbiamo fornire tutti i dati richiesti, magari sempre gli stessi
- ❑ Vedremo diversi meccanismi per la lettura/scrittura di file da parte di programmi Python, per il momento **chiediamo l'aiuto del sistema operativo** o, più precisamente, della *shell* dei comandi (che fa parte del sistema operativo)
  - Attenzione: non la shell di Python... la shell del sistema operativo
  - **Attenzione: in Windows, NON la "nuova" PowerShell ma la "vecchia" shell che si chiama "Prompt dei comandi"**

# Acquisizione di dati in input

- ❑ Quando eseguiamo l'interprete Python, istruendolo perché esegua uno specifico sorgente Python (es. `myProgram.py`), scriviamo nella *shell*

```
python myProgram.py
```

- ❑ Aggiungendo una clausola che inizia con il carattere `<` e contiene **il nome di un file**, chiediamo al sistema operativo di cooperare con l'interprete perché questo usi un *flusso di input* diverso da quello proveniente dalla tastiera

```
python myProgram.py < textFile.txt
```

- ❑ Durante l'esecuzione del programma da parte dell'interprete, quando viene eseguita la funzione `input`, i caratteri vengono letti ordinatamente dal file anziché dalla tastiera
  - **ATTENZIONE:** durante l'esecuzione del programma la tastiera "non funziona più!", cioè con questo meccanismo **non possiamo fornire dati usando sia il file di input sia la tastiera (o l'uno, o l'altra)**
  - I dati devono essere inseriti nel file esattamente come li scriveremmo sulla tastiera, andando a capo quando andremmo a capo, ecc.
  - **Questo non richiede alcuna modifica al sorgente del programma**
  - Il programma in esecuzione non è nemmeno "consapevole" di questo fenomeno, che si chiama **redirezione o reindirizzamento di input**

# Archiviazione di dati in output

- ❑ Quando eseguiamo un programma che visualizza molti dati, sarebbe comodo poterli archiviare in un file, per poterli analizzare successivamente
  - Ovviamente possiamo fare "copia e incolla" nella finestra dove il programma ha visualizzato i dati, inserendoli poi in un file di testo creato da noi (usando un editor come intermediario)
  - In alternativa, possiamo di nuovo chiedere l'aiuto del sistema operativo, con la *redirezione di output*

```
python myProgram.py > output.txt
```

- L'intero flusso di caratteri visualizzato dal programma con le invocazioni della funzione **print** non finisce più nella solita finestra ma nel file
  - Di nuovo, senza **nessun intervento nel codice del programma**
  - **ATTENZIONE:** se il file è già presente, viene *sovrascritto*, cioè il suo contenuto viene cancellato e vi vengono scritti **SOLTANTO** i caratteri prodotti dall'esecuzione del programma

# Redirezione di input/output

- ❑ Il meccanismo di redirezione di input e/o di output è abbastanza flessibile, **non richiede interventi nel codice sorgente** e l'utente può utilizzarlo oppure no, decidendo di volta in volta, a ogni esecuzione del programma

- ❑ **Le due redirezioni si possono anche combinare insieme**

```
python myProgram.py < input.txt > output.txt
```

- ❑ La flessibilità non è totale e per questo motivo vedremo soluzioni migliori (ma più complesse), che consentono di
  - Acquisire i dati da più di un solo file
    - Ad esempio, un programma potrebbe leggere i numeri di matricola degli studenti da un file e i loro dati anagrafici da un altro file
  - Distribuire i risultati prodotti su più di un file
    - Ad esempio, un programma potrebbe scrivere i numeri di matricola degli studenti in un file e i loro dati anagrafici in un altro file
- ❑ **Attenzione: tutto questo NON funziona con IDLE né con la shell interattiva di python, solo con la shell del sistema operativo**

# Gestione di file

# Leggere e scrivere file

- ❑ Mediante la **redirezione di input/output** un programma Python può leggere dati da un file e scrivere dati in un file in modo "trasparente" (o "implicito") cioè senza usare alcuna funzione particolare all'interno del codice
  - Il sistema operativo indirizza i flussi in modo diverso dal solito
- ❑ Ci sono però diversi limiti
  - Si può leggere un solo file e si può scrivere un solo file
  - Se si usa la redirezione di input, non si possono acquisire dati dalla tastiera
  - Se si usa la redirezione di output, non si possono scrivere dati sullo schermo
- ❑ Per superare questi limiti, dobbiamo **manipolare i file in modo esplicito** all'interno del programma



# Leggere file di testo

- ❑ Ci occupiamo soltanto di *file di testo*, cioè file contenenti caratteri, ciascuno dei quali codificato in binario secondo lo standard Unicode



- Sono file leggibili e modificabili con un *editor di testo*

- ❑ Per **leggere** un file di testo (cioè, tecnicamente, acquisire dati che siano memorizzati in esso) bisogna per prima cosa *aprire il file*

- Questa operazione richiede un'interazione con il sistema operativo

- Il quale, a sua volta, gestisce il *file system*, cioè la strategia di organizzazione dei *file* all'interno di un dispositivo di memoria secondaria



- Avviene tramite l'invocazione della funzione predefinita **open**, che restituisce un oggetto che "rappresenta" il file all'interno del programma (in realtà, rappresenta "tutto ciò che serve per interagire con il file tramite il sistema operativo")

- Concetto analogo all'oggetto che rappresenta una finestra grafica

# Leggere file di testo

- ❑ Gli argomenti per la funzione **open** sono il nome del file e una stringa che descrive la *modalità di apertura*

("r" per la lettura, *read*)

```
f = open("input.txt", "r")
```

- Quando la modalità è "r" può essere omessa, ma noi, per chiarezza, la indichiamo (vedremo poi le altre modalità possibili)

- Il nome del file può essere *relativo* o *assoluto*: è assoluto se indica il percorso per trovare il file a partire dalla *radice del file system*, altrimenti il percorso (o anche il solo nome) è relativo alla cartella in cui si sta eseguendo il programma



- In un sistema Windows, i nomi dei percorsi contengono il carattere \ che, ovviamente, va inserito nella stringa come \\

- Terminate le operazioni di lettura, il file va *chiuso* invocandone il **metodo close** (se ce ne dimentichiamo, in realtà non succede nulla di "pericoloso"...)

```
f.close()
```

# Leggere file di testo: metodo **readline**

- ❑ Per acquisire dati (**sempre di tipo stringa**) da un file già aperto, ci sono diverse possibilità, solitamente invocandone metodi
- ❑ Il metodo **readline** è molto simile alla funzione **input** che usiamo per acquisire stringhe dal flusso di input standard (che è solitamente la tastiera, se non c'è stata redirectione di input...)
- ❑ Diversamente da **input**, **readline** termina sempre la stringa restituita con il carattere **\n**, TRANNE quando è arrivato alla fine del file: in tal caso (e solo in tal caso) restituisce la stringa vuota
- ❑ Ogni invocazione di **readline** restituisce la successiva riga del file di testo, a partire dalla prima (l'oggetto **f** si ricorda dove è arrivato a leggere il file a cui è stato associato durante l'apertura)
  - una riga è definita come una sequenza di caratteri, eventualmente vuota, terminata da un carattere **\n**

```
f = open("input.txt", "r")
line = f.readline()
while line != "":
    elabora line
    line = f.readline()
f.close()
```

# Leggere file di testo: contenitore per **for**

- ❑ In un ciclo **for**, se usiamo come contenitore l'oggetto di tipo "file di testo" restituito dalla funzione **open**, Python lo considera come **una sequenza di stringhe** (le singole righe del file, come se le leggesse con **readline**)

```
f = open("input.txt", "r")
for line in f :
    elabora line
f.close()
```

- ❑ La variabile **line** farà riferimento alle righe di testo del file, una dopo l'altra (ciascuna terminata da un carattere **\n**, tenerne conto nell'elaborazione...)

# Leggere file di testo: metodo `readlines`

- Un altro metodo utile può essere `readlines`: restituisce una lista di **stringhe**, ciascuna delle quali è una delle righe del file di testo (nell'ordine in cui compaiono), terminata da un carattere `\n`

```
f = open("input.txt", "r")
lines = f.readlines()
f.close() // già qui
for line in lines :
    elabora line
i = 0
while i < len(lines) :
    line = lines[i]
    elabora line
    i += 1
```

- In pratica, legge l'intero file in un colpo solo
- È comodo soprattutto quando si vogliono fare più elaborazioni successive del contenuto di uno stesso file, perché non c'è bisogno di leggerlo due volte: il contenuto del file rimane disponibile all'interno della lista creata e restituita da `readlines`

# Leggere file di testo: metodo `read`

- ❑ Il metodo `read()` restituisce **un'unica stringa** contenente tutti i caratteri del file (compresi i caratteri `\n` che separano una riga dalla successiva)
- ❑ È utile soprattutto quando si vogliono elaborare i caratteri del testo senza tener conto della sua suddivisione in righe (es. contare le lettere maiuscole) perché, in tal caso, la suddivisione in righe sarebbe solamente **scomoda...**
- ❑ Il metodo `read` può anche ricevere un parametro di tipo intero
  - In tal caso restituisce una stringa contenente al massimo quel numero di caratteri (o meno, se il file termina)
  - Invocazioni successive di `read` ripartono dal punto in cui era arrivato in precedenza (utile per leggere un file "a blocchi", per occupare meno memoria)

```
f = open("input.txt", "r")
s = f.read()
count = 0
for c in s :
    if c.isupper() :
        count += 1
```

```
f = open("input.txt", "r")
lines = f.readlines()
count = 0
for line in lines :
    for c in line :
        if c.isupper() :
            count += 1
```

# Leggere file di testo

- ❑ Indipendentemente dal metodo utilizzato per acquisire dati da un file di testo, se si vuole "ripartire dall'inizio" nella scansione delle righe è necessario
  - Chiudere il file
  - Aprire di nuovo il file, invocando nuovamente **open**
- ❑ Ovviamente, se il contenuto del file è stato acquisito mediante il metodo **readlines**, la lista di righe che è stata restituita può essere scandita più volte: è una lista di stringhe, il fatto che il suo contenuto provenga da un file è del tutto ininfluenza
  - Analogamente per una stringa acquisita con **read**
- ❑ **È possibile avere più file contemporaneamente aperti (memorizzati in oggetti diversi)**
  - Es. leggere righe alternativamente da un file e da un altro file, come in un esempio che vedremo



# Scrivere file di testo

- ❑ Per **scrivere** un file di testo bisogna per prima cosa *aprire il file* in "modalità scrittura" ("**w**")



- Se il file esiste già, viene "svuotato" prima di iniziare a scrivere
- Usando, invece, la modalità *append* ("**a**"), se il file esiste già i nuovi dati vengono aggiunti alla fine (se il file non esiste, questa modalità coincide con la precedente)

- ❑ Per scrivere una stringa si usa il

```
f = open("output.txt", "w")  
f.write("Hello, World! \n")  
f.close()
```

metodo **write**, che, diversamente da **print**, NON aggiunge caratteri **\n**: li dobbiamo inserire esplicitamente

- ❑ Naturalmente la stringa fornita come argomento a **write** può anche essere generata con l'operatore di impaginazione **%** (o in qualunque altro modo...)



**Lezione 24**  
**26/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Gestione di file: un'applicazione

# Applicazione

□ Scrivere un programma che legge due file

- Il primo (**names.txt**) contiene numeri di matricola e nomi di studenti, con questo formato

```
1763452;Minnie  
1122543;Mickey Mouse  
1235436;Donald Duck  
1324321;Daisy Duck
```

- Il formato si chiama **CSV** (*comma-separated values*) ed è usato dai programmi *spreadsheet* o *fogli elettronici* (come Excel o simili)
  - Come separatore si possono usare caratteri diversi (virgola, "due punti", ecc.), qui viene usato il "punto e virgola" [ovviamente da concordare tra chi scrive e chi legge il file]
- Ogni riga si chiama **record** e ogni suo elemento si chiama **campo** o **field** (terminologia tipica delle *basi di dati* o *database*)

- Il secondo (**grades.txt**) contiene numeri di matricola (gli stessi contenuti in **names.txt** e nello stesso ordine) e i voti conseguiti dagli studenti negli esami sostenuti (il numero di voti è variabile)

```
1763452;30;28;30;26  
1122543;28;30;30;30  
1235436;24;18  
1324321;30;30;30;30
```

□ Il programma deve, poi, generare un file... descritto nel seguito

# Applicazione

- ❑ Il programma deve, poi, generare un file contenente i numeri di matricola, i nomi degli studenti e la media dei voti conseguiti

```
1763452;Minnie;28.5  
1122543;Mickey Mouse;29.5  
1235436;Donald Duck;21.0  
1324321;Daisy Duck;30.0
```

- ❑ Il file generato si deve chiamare **averages.txt**
  - In questa prima versione il programma non farà nessun controllo sui file che legge: supponiamo che i dati siano corretti (es. se i numeri di matricola non corrispondono?)
- ❑ La logica del programma è
  - Leggi ripetutamente una riga da ciascuno dei due file di input e genera la riga corrispondente, scrivendola nel file di output
    - In pratica, il file da generare è uguale a **names.txt**, con l'aggiunta della media dei voti al termine di ciascuna riga

```
def computeAvg(s) :
    # s ha questa forma:
    # "1763452;30;28;30;26\n"
    # il numero di matricola va ignorato
    # il numero di voti non è predefinito
    ... # da completare come esercizio
    return ... # la media dei voti

in1 = open("names.txt", "r")    # read
in2 = open("grades.txt", "r")  # read
out = open("averages.txt", "w") # write
for line1 in in1 : # leggo names.txt
    out.write(line1[:-1]) # toglie \n finale
    out.write(";")
    line2 = in2.readline() # leggo grades.txt
    avg = computeAvg(line2)
    out.write(str(avg) + "\n")
    # attenzione: cicli annidati? non avrebbe senso...
    # for line1 in in1 :
    #     for line2 in in2 :
    #         ...
    # cosa farebbero questi cicli annidati?
in1.close()
in2.close()
out.close()
```

# Come modificare un file esistente?

- ❑ Per modificare il contenuto di un file esistente, come risultato di una elaborazione del suo attuale contenuto, la strategia più semplice è quella di
  - Aprire il file in "lettura"
  - Acquisire l'intero contenuto del file, memorizzandolo in una stringa, con **read()**, oppure in una lista di stringhe, con **readlines()**
  - Chiudere il file
  - Aprire il file in "scrittura"
    - Azione che provoca la **cancellazione** del contenuto del file
  - Elaborare il contenuto del file precedentemente acquisito
  - Scrivere nel file il suo nuovo contenuto
  - Chiudere il file
- ❑ È sempre preferibile fare preventivamente un *backup* del file...

# Eccezioni e loro gestione

# Eccezioni "sollevate"

- ❑ Sappiamo che, **in situazioni di errore**, molte funzioni e molti metodi della libreria di Python **"sollevano eccezioni"**, un termine tecnico usato per rappresentare un'azione che provoca l'interruzione brusca dell'esecuzione del programma
  - Viene visualizzato **uno "strano" messaggio d'errore**, significativo per i programmatori, che lo sanno interpretare, ma **inutile (se non dannoso) per i normali utenti dei programmi**
  - **Un programma (semi-)professionale non dovrebbe mai terminare in questo modo**
  - D'altra parte, se a una funzione vengono forniti argomenti errati, cosa può fare?
    - Es. **int("pippo")**: la funzione **int** non ha alternative, non può inventarsi un numero da restituire... quindi solleva l'eccezione **ValueError**



# Eccezioni "sollevate"

- ❑ Se a una funzione vengono forniti argomenti errati, cosa può fare?
  - Es. `int("pippo")`, la funzione `int` solleva **ValueError**
- ❑ Ovviamente cerchiamo di non fornire valori errati alle funzioni che invochiamo, ma cosa possiamo fare quando l'errore viene commesso dall'utente attraverso il flusso di input oppure è frutto dell'elaborazione di valori "sbagliati" acquisiti in input?
  - Prima di fornire alla funzione `int` la stringa acquisita mediante `input()` possiamo sempre verificare se tale stringa contiene effettivamente caratteri che descrivono un numero intero
    - Ma tale algoritmo di verifica verrà già eseguito all'interno della funzione `int`!  
Verrebbe eseguito due volte... e non è un algoritmo semplice...
  - È preferibile che tale verifica venga fatta **solo** nella funzione `int` (che non può evitare di fare tali controlli)
  - **La cosa più efficace è imparare a prendere contromisure nel momento in cui una funzione solleva un'eccezione**

# Gestione delle eccezioni

- ❑ La cosa più efficace è imparare a **prendere contromisure nel momento in cui una funzione solleva un'eccezione**

- ❑ Questa strategia si chiama *gestione delle eccezioni* (*exception handling*)
  - nel momento in cui una funzione solleva un'eccezione, questa viene "catturata" mediante un apposito costrutto sintattico predisposto dal programmatore e viene eseguito codice "riparatore"

```
try :  
    n = int(input("Un numero intero: "))  
    print("Numero valido:", n)  
except ValueError :  
    # contromisura per ovviare all'errore  
    n = int(input("Riprova: "))
```

- ❑ L'enunciato **try** introduce un blocco di codice che *potrebbe* sollevare un'eccezione che ci interessa gestire
- ❑ La clausola **except** dichiara **quale eccezione siamo interessati a gestire** e il suo corpo viene eseguito **soltanto** nel caso in cui tale eccezione venga sollevata, intercettandola e inibendone gli effetti

# Gestione di eccezioni

```
try :  
    n = int(input("Un numero intero: "))  
    print("Numero valido", n)  
except ValueError :  
    # contromisura per ovviare all'errore  
    n = int(input("Riprova: "))
```

- ❑ L'esecuzione dell'enunciato composito **try/except** prevede che **normalmente** venga eseguito soltanto il corpo del **try** (che può essere costituito da più enunciati), senza eseguire il corpo della clausola **except**
  - Se non vengono sollevate eccezioni all'interno del blocco **try**, l'esecuzione dell'enunciato **try/except** è terminata e il programma prosegue con gli enunciati che si trovano dopo il blocco **except**
  - Nel momento in cui, invece, viene sollevata un'eccezione, l'esecuzione del blocco **try** termina (anche se fossero presenti altri enunciati), dopodiché:
    - se l'eccezione è diversa da quella dichiarata nella clausola **except**, tutto va come se la clausola **except** non ci fosse: il programma terminerà con il consueto messaggio "strano", inadatto agli utenti
    - se, invece, l'eccezione è proprio quella dichiarata nella clausola **except**, il suo effetto viene **gestito e non provoca la terminazione del programma**: viene eseguito il corpo della clausola **except**
      - Successivamente, il flusso dell'esecuzione procede con gli enunciati che seguono l'enunciato **try/except**, **NON** si torna a terminare l'esecuzione del blocco **try** interrotto

# Gestione delle eccezioni

- ❑ L'enunciato composto **try/except** è un enunciato come gli altri... quindi può, ad esempio, essere inserito nel corpo di un ciclo

```
while True :  
    try :  
        n = int(input("Un numero intero: "))  
        print("Numero valido", n)  
        break  
    except ValueError :  
        print("Riprova")  
  
# qui n fa sicuramente riferimento a un numero intero
```

- ❑ Se l'utente fornisce un numero intero, viene eseguito soltanto il blocco **try**, che esegue **break** e il ciclo termina dopo aver memorizzato il numero in **n**
- ❑ Altrimenti, **int** lancia l'eccezione **ValueError** e l'esecuzione del blocco **try** si interrompe PRIMA dell'esecuzione di **break**
  - Più precisamente, l'esecuzione del blocco **try** si interrompe prima dell'esecuzione dell'assegnazione di un valore alla variabile **n** (perché non la funzione **int** non ha generato un valore da poter assegnare a **n**)
  - Viene poi eseguito il corpo della clausola **except**, dopodiché l'iterazione del ciclo termina e ne viene eseguita un'altra

# Gestione delle eccezioni

- ❑ A volte, come nel caso precedente, non abbiamo bisogno di eseguire alcuna contromisura: ci basta intercettare l'eccezione senza fare niente (il messaggio "Riprova" non è davvero necessario...)
- ❑ La sintassi, però, prevede che la clausola **except** abbia un corpo, costituito da almeno un enunciato
- ❑ Possiamo usare l'enunciato **pass** che... **non fa niente!**
  - Serve soltanto come "riempitivo" in quelle situazioni, come questa, quando deve necessariamente essere presente un enunciato ma non vogliamo eseguire nessuna azione
    - Potrei scrivere un enunciato inutile, come **unaVariabile = 0** o qualcosa di simile, ma sarebbe veramente fuorviante

```
while True :  
    try :  
        n = int(input("Un numero intero: "))  
        break  
    except ValueError :  
        pass  
  
# qui n fa sicuramente riferimento a un numero intero
```

# Gestione delle eccezioni

- ❑ Attenzione alla logica dell'algoritmo e alla posizione degli enunciati, in relazione a quando vogliamo che vengano eseguiti...

- ❑ Questa soluzione non è corretta

- ❑ Qui il **break** si trova DOPO l'enunciato

```
while True :  
    try :  
        n = int(input("Un intero: "))  
    except ValueError :  
        pass  
    break # posizione sbagliata!  
print(n)
```

**try/except**, quindi viene eseguito COMUNQUE, sia che venga sollevata l'eccezione sia che non venga sollevata

- ❑ È un "finto ciclo" che compie sempre una sola iterazione... e non risolve il problema di dare un valore alla variabile **n**
  - Se l'utente non fornisce un numero, non viene segnalato **ValueError** (perché tale eccezione viene catturata e gestita), ma l'esecuzione di **print** solleva l'eccezione **NameError** perché la variabile **n** non è stata definita (l'esecuzione del blocco **try** è stata interrotta all'interno della funzione **int**, prima che **n** ricevesse un valore)

# Gestione delle eccezioni

```
try :  
    codice "pericoloso", che può sollevare eccezioni  
    e che può essere eseguito anche solo in parte  
except NomeDiEccezione :  
    codice eseguito solo se viene sollevata l'eccezione
```

- ❑ Dobbiamo capire bene quando e se vengono eseguiti i diversi corpi dell'enunciato **try/except**
  - È molto importante ricordare che il codice del blocco **try** può essere eseguito anche solo **parzialmente**, cosa che a volte è proprio ciò che vogliamo... come nell'esempio precedente: quando viene sollevata un'eccezione, vogliamo che l'enunciato **break** NON venga eseguito!

```
while True :  
    try :  
        n = int(input())  
        break  
    except ValueError :  
        pass
```



# Un altro esempio

```
while True :  
    try :  
        name = input("Nome del file da leggere: ")  
        f = open(name, "r")  
        break  
    except FileNotFoundError :  
        print("File", name, "non trovato")  
# qui f è un file valido e già aperto in lettura
```

- ❑ Come facciamo a scoprire il nome dell'eccezione che viene sollevata da una funzione o da un metodo?
  - Consultiamo la documentazione...
  - Facciamo un esperimento! 😊  
Scriviamo il codice **senza try/except** e forniamo dati che inneschino l'eccezione, poi osserviamo quanto viene scritto dall'interprete e scopriamo il nome dell'eccezione che è stata sollevata



# Gestione obbligatoria?

- ❑ Dobbiamo sempre catturare e gestire le eccezioni?
  - Un programma professionale non dovrebbe mai terminare perché è stata sollevata un'eccezione che non è stata gestita, dato che l'utente riceverebbe un messaggio d'errore poco chiaro
  - Nel nostro corso ci "accontentiamo" di gestire quelle eccezioni che ci "aiutano" nella realizzazione del programma, nel senso che l'esecuzione intenzionale di codice "pericoloso" a volte ci consente di risparmiare codice e/o memoria e/o tempo d'esecuzione
- ❑ La sintassi consente anche di scrivere la clausola **except** *senza nominare l'eccezione che si vuole catturare* e gestire... ma non è un buono stile di programmazione, perché maschera eventuali eccezioni impreviste, che è meglio che si manifestino (durante la fase di debugging...)

```
try :  
    n = int(input("Un numero intero: "))  
except :    # qualsiasi eccezione  
    n = int(input("Riprova: "))
```

Il materiale necessario per il  
Laboratorio 07  
termina qui

**Sollevare eccezioni**

# Come sollevare eccezioni

- ❑ Abbiamo visto che, in Python, funzioni e metodi possono sollevare eccezioni, e abbiamo imparato a gestirle o a lasciarle agire (provocando la terminazione del programma)
- ❑ Ma come fanno tali funzioni e metodi a sollevare eccezioni?
  - Evidentemente esiste un enunciato Python la cui esecuzione solleva (*raise*) un'eccezione!
  - L'enunciato è proprio **raise**

```
def strToInt(s) : # funziona come int(...) predefinita
    isNumber = True
    ... # controllo se la stringa s contiene un numero
    if not isNumber :
        raise ValueError
    ... # converto la stringa in numero
    return ...
```

# Come sollevare eccezioni

- ❑ Con **raise**, oltre al tipo di eccezione sollevata, possiamo definire anche una stringa che verrà visualizzata dall'interprete in seguito all'interruzione brusca del programma, se l'eccezione non verrà gestita

```
def strToInt(s) : # funziona come int(...) predefinita
    isNumber = True
    ... # controllo se la stringa s contiene un numero
    if not isNumber :
        raise ValueError(s + " non contiene un numero")
    ... # converto la stringa in numero
    return ...
```

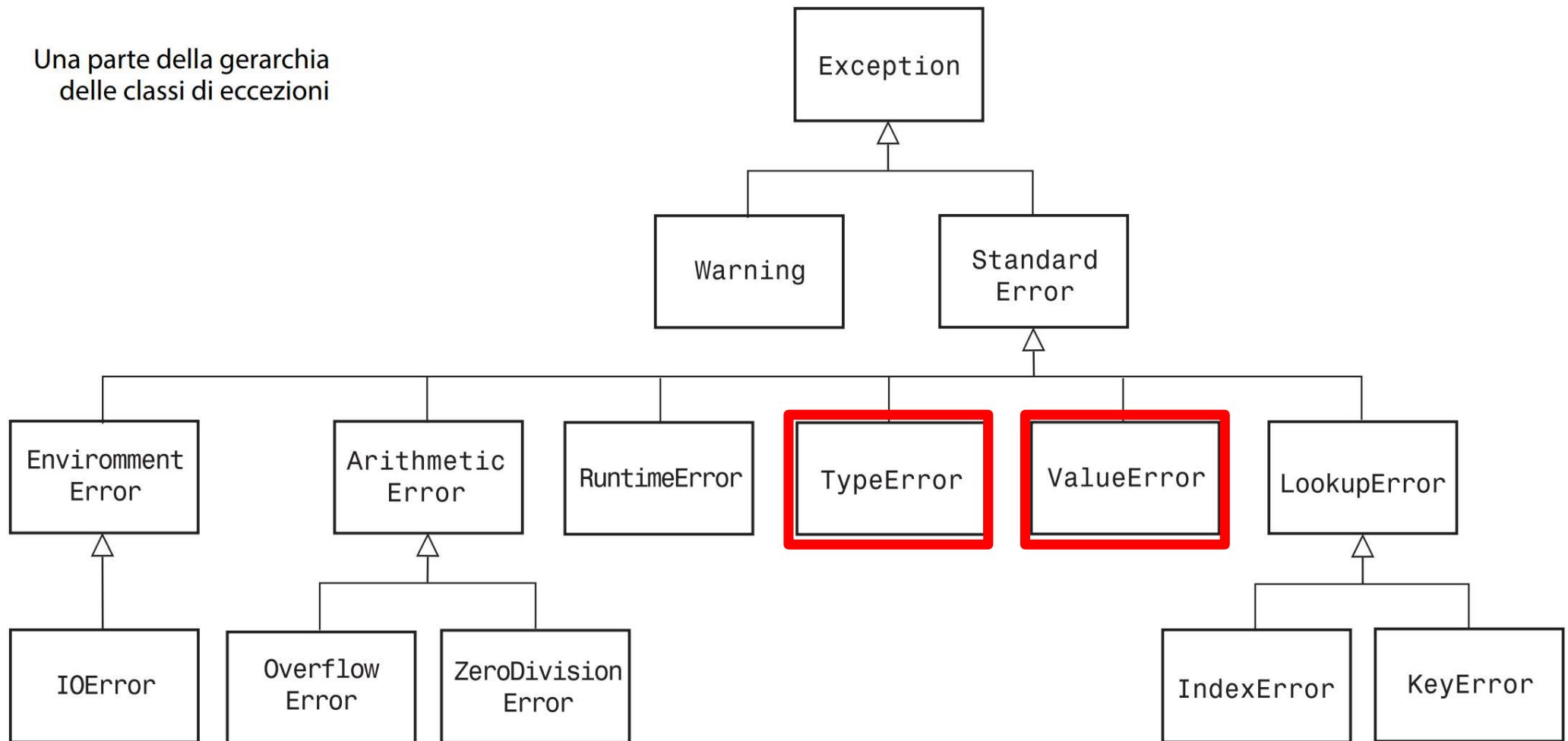
- ❑ Il tipo di eccezione sollevata da una funzione o da un metodo (e i casi in cui questo avviene) fa parte della loro documentazione, perché gli utilizzatori lo devono conoscere

# Come sollevare eccezioni

- ❑ L'esecuzione della funzione o del metodo si **interrompe** nel momento in cui viene eseguito l'enunciato **raise**
- ❑ Viene ripresa l'esecuzione del codice invocante, come se la funzione o il metodo avessero eseguito **return**, ma non c'è un valore restituito
  - Se l'esecuzione riprende all'interno di un blocco **try** che contiene una clausola **except** dedicata all'eccezione sollevata, l'eccezione viene catturata e gestita
  - Altrimenti, anche la funzione o il metodo invocante si interrompono e viene ripresa l'esecuzione del codice che li aveva invocati... e così via, controllando se c'è un blocco **try** adatto alla cattura lungo la catena di invocanti
  - Se si arriva a interrompere il codice principale, il programma termina con la visualizzazione del messaggio d'errore che conosciamo

# Quale eccezione solleviamo?

Una parte della gerarchia delle classi di eccezioni



- (Per il momento) **Scegliamo** una delle eccezioni che conosciamo o comunque **una di quelle predefinite che abbia un nome significativo** per il problema che vogliamo segnalare

Un "range" non è una lista...  
ma quasi



# Un "range" non è una lista... ma quasi

- L'oggetto restituito da un'invocazione della funzione **range** (la quale, propriamente, non è una funzione ma un "costruttore")  
**è molto simile a una lista, ma tecnicamente non lo è**

- È, appunto, un oggetto di tipo **range**
- Per sapere di che tipo è un oggetto si può usare la funzione predefinita **type**, che restituisce il tipo dell'oggetto ricevuto come argomento:  
utile per il debugging  
o anche per altro,  
come vedremo

```
print(type(range(5))) # <class 'range'>
print(type(2))        # <class 'int'>
print(type((2, )))    # <class 'tuple'>
x = "pippo" # anche con variabili
print(type(x))        # <class 'str'>
```

- La funzione **type** si può usare anche per verificare che un valore (ad esempio, ricevuto da una funzione) sia del tipo corretto

- Si confronta il "valore" restituito da **type** con

**il nome del tipo di**

**dato** (int, float, str, list, tuple, GraphicsWindow ...)

```
def unaFunzione(unNumeroIntero) :
    if type(unNumeroIntero) != int :
        raise TypeError
    elaborazione di unNumeroIntero
```

# Un "range" non è una lista... ma quasi

- ❑ Passando un oggetto di tipo **range** alle funzioni **list/tuple** si ottiene una lista/tupla avente il medesimo contenuto
  - Può essere comodo per generare liste/tuple contenenti numeri interi in successione opportuna
    - In alternativa all'uso di un ciclo che costruisca la stessa lista/tupla, ricordando che una tupla si può "allungare" mediante concatenazione...

```
x = list(range(2, 11, 2))  
print(x) # [2, 4, 6, 8, 10]  
y = tuple(range(2, 11, 2))  
print(y) # (2, 4, 6, 8, 10)
```

**Lezione 25**  
**27/11/2024**  
**ore 10.30-12.30**  
**aula Ve**

Funzioni che **MODIFICANO**  
liste ricevute come parametro

# Oggetti e riferimenti

- ❑ Con l'introduzione di "**oggetti modificabili**", dobbiamo fare qualche ulteriore considerazione sull'enunciato di **assegnazione di un valore a una variabile**

- Le liste sono i primi oggetti modificabili che abbiamo visto

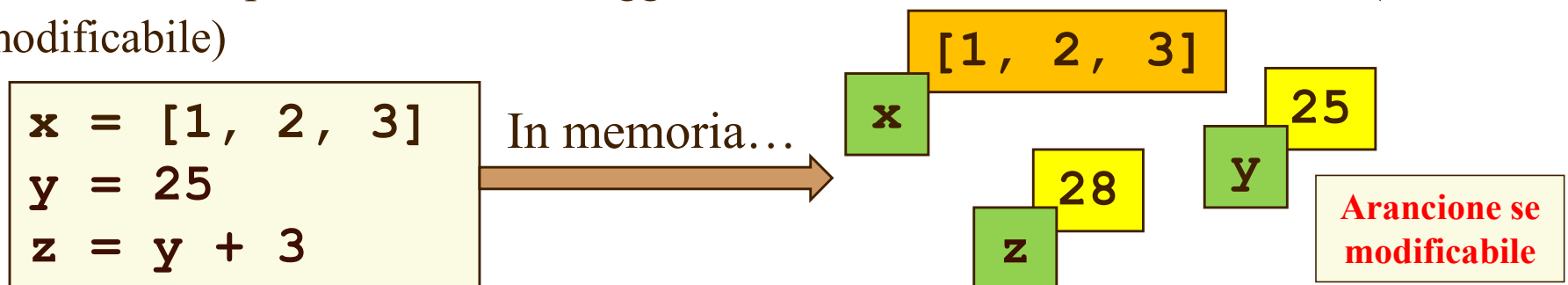
- ❑ Ricordiamo che l'enunciato di assegnazione **calcola un valore**

- il risultato della valutazione dell'espressione che si trova a destra del segno =  
**e lo assegna alla variabile** che si trova a sinistra del segno =

- ❑ Abbiamo sempre detto che i dati (che, in Python, si chiamano tecnicamente *oggetti*) vengono memorizzati nelle variabili, in realtà ora scopriamo che nelle **variabili vengono memorizzati riferimenti agli oggetti**



- Per capire cosa sia un riferimento e come questo funzioni durante l'esecuzione di un programma, **usiamo un'allegoria**
  - Gli **oggetti** sono "scatole" (qui gialle/arancioni) che contengono dati, mentre i **riferimenti** sono "etichette" (qui verdi) che vengono poste sulle scatole per identificarle e poterle utilizzare, leggendo o modificando il loro contenuto (se è modificabile)

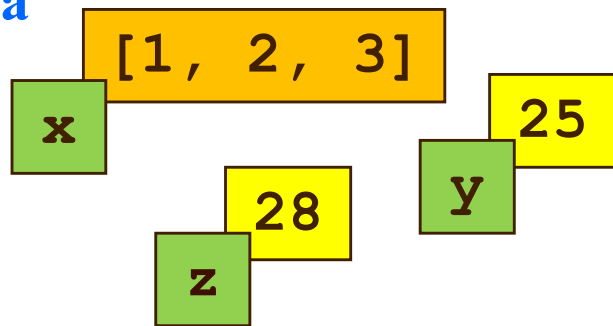


# Oggetti e riferimenti

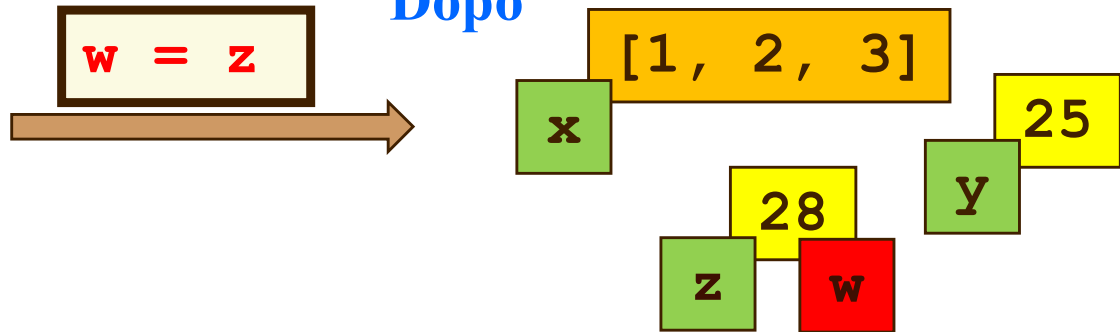
## ❑ Alcune istruzioni di assegnazione hanno un comportamento speciale

- Quando l'espressione a destra del segno = è semplicemente un nome di variabile, come risultato della valutazione di tale "espressione" **NON viene generato un nuovo oggetto** (in pratica, perché è un'espressione "molto semplice"...) bensì viene **riutilizzato il dato**, **aggiungendogli** una ulteriore etichetta

Prima

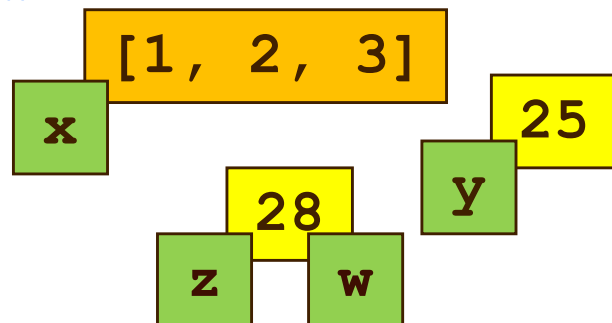


Dopo

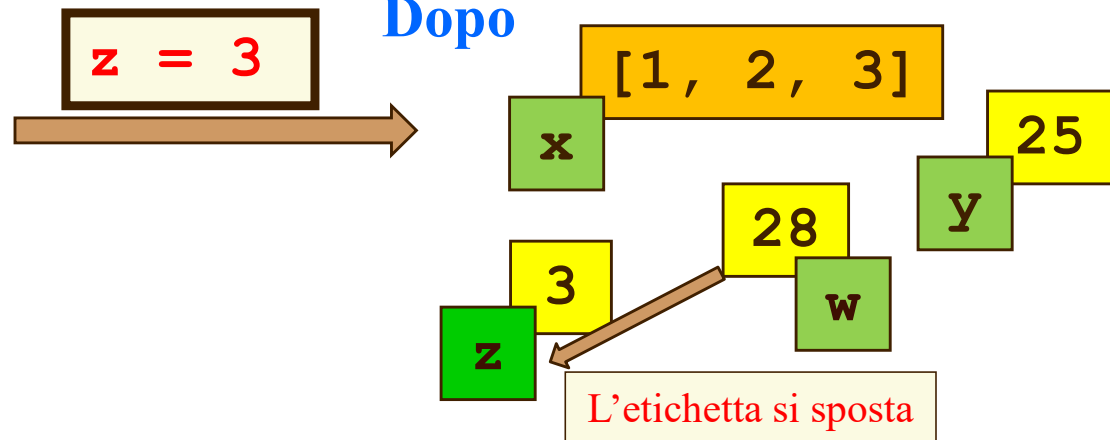


- Si dice che viene creato un **alias** di un oggetto anziché una sua copia (cioè `w` è un alias); dopo la creazione, **gli alias sono etichette come le altre**

Prima



Dopo



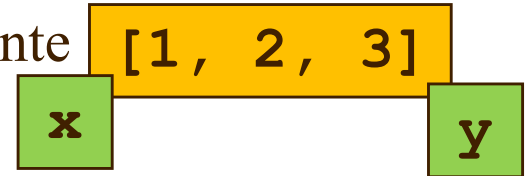
# Clonazione di liste

- ❑ Osserviamo con attenzione queste due istruzioni che "copiano" una lista: differenze?

```
x = [1, 2, 3]
y = x          # alias
z = list(x)    # clone
```

- ❑ **La prima** **NON** crea una nuova lista, cioè una nuova "scatola"

- Assegna una nuova etichetta (**y**) all'unica lista inizialmente presente nel programma, che aveva già l'etichetta **x**
- In pratica, si crea un *alias*, cioè **un altro riferimento, di nome diverso, con cui accedere (in lettura E IN SCRITTURA) alla medesima lista**



- Vedremo a cosa serve un alias

- Se **modifico** la lista usando **y**, vedo le modifiche anche usando **x** (e viceversa): **mediante le due variabili accedo al medesimo oggetto!**

```
y[0] = 7
print(x[0]) # 7
```

- ❑ **La seconda** crea una **nuova** lista, un **clone** della prima

- Al termine dell'esecuzione della funzione **list**, le due liste sono identiche

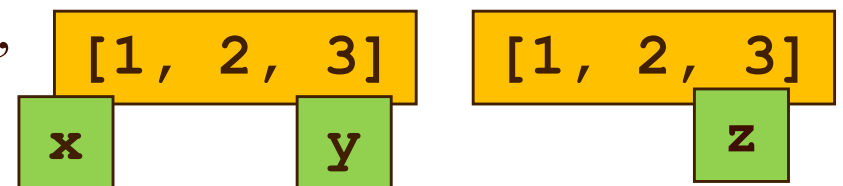
- In futuro, le due liste non avranno alcuna relazione tra loro: **modifiche a una lista non avranno alcun effetto sull'altra**

```
z[2] = 5
print(x[2]) # 3
```

- **Le liste si trovano in due "scatole" distinte, inizialmente con lo stesso contenuto!**



- Finché gli oggetti non erano modificabili, questa distinzione di comportamento tra alias e cloni non serviva!



# Liste come argomenti di funzioni

❑ Una lista, come qualsiasi altro tipo di dato, può essere fornita a una funzione come argomento

❑ **values** sarà un alias di **x** oppure sarà un clone di **x** ?

■ Ricordiamo il meccanismo di passaggio degli argomenti durante l'invocazione di una funzione

- L'inizializzazione (implicita) della variabile parametro **values** effettuata dall'interprete nel momento dell'invocazione **mysum(x)** equivale a **values = x**
  - serve un intervento "speciale" dell'interprete perché le due variabili si trovano in ambiti di visibilità diversi!
- Quindi **values** non è un clone, **è un alias!**

```
def mysum(values) :  
    total = 0  
    for element in values :  
        total += element  
    return total
```

```
x = [1, 3, 5, 4, 7]  
print(mysum(x)) # 20
```



# Liste come argomenti di funzioni

- ❑ Quando, come in questo caso, all'interno della funzione la lista ricevuta come parametro viene solamente "letta" (e non modificata), il fatto che sia un clone o un alias (ovviamente...) non fa differenza

```
def mysum(values) :  
    total = 0  
    for element in values :  
        total += element  
    return total  
x = [1, 3, 5, 4, 7]  
print(mysum(x)) # visualizza 20
```

- ❑ Ma se la funzione **MODIFICA** la lista, dato che agisce tramite un alias, **ciò che viene modificato è il contenuto dell'oggetto** (di tipo lista) **condiviso** tra invocante e invocato!

```
def multiplyBy2(values) :  
    for i in range(len(values)) :  
        values[i] *= 2  
    # nessun valore restituito  
x = [1, 3, 5, 4, 7]  
multiplyBy2(x)  
print(x) # [2, 6, 10, 8, 14]
```

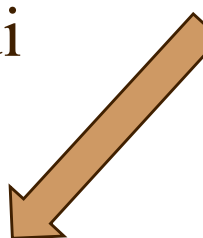
- ❑ L'oggetto (la lista) è "condiviso" nel senso che **ne esiste soltanto uno, con due alias** (cioè due diverse "etichette", appartenenti a due diversi ambiti di visibilità)

- **Gli alias hanno senso quando si trovano in ambiti di visibilità diversi!**

# Assegnare alla variabile parametro...

- ❑ Potremmo avere l'idea di rendere **più semplice** questa funzione

```
def clearList(values) :  
    values = [0] * len(values)  
  
x = [1, 3, 5, 4, 7]  
clearList(x)  
# non ha funzionato!!!  
print(x) # [1, 3, 5, 4, 7]
```



```
# azzera una lista  
def clearList(values) :  
    for i in range(len(values)) :  
        values[i] = 0  
  
x = [1, 3, 5, 4, 7]  
clearList(x)  
print(x) # [0, 0, 0, 0, 0]
```

- ❑ Nella seconda funzione, ho assegnato a **values** il riferimento a **una nuova lista**, ma questo non ha (ovviamente e correttamente) avuto alcun effetto su **x** !
  - Semplicemente, **values** non fa più riferimento alla stessa lista a cui fa riferimento **x**, ho spostato l'etichetta **values** da un oggetto a un altro

- ❑ Stesso problema se i dati sono oggetti non modificabili...  
**errore logico!**

```
def clearVar(v) :  
    v = 0  
  
x = 5  
clearVar(x)  
print(x) # 5, non 0
```

Ragionare con  
"etichette" e  
"scatole"...

# Modificare liste con un ciclo `for`

```
x = [1, 3, 5, 4, 7]
for i in range(len(x)) :
    x[i] *= 2
print(x) # [2, 6, 10, 8, 14]
```

- ❑ Questi due cicli svolgono APPARENTEMENTE la stessa elaborazione, ma gli effetti prodotti sulla lista **x** sono ben diversi...

```
x = [1, 3, 5, 4, 7]
for e in x :
    e *= 2
print(x) # [1, 3, 5, 4, 7]
# la lista x non è stata
# modificata... perché ?
```

## ❑ Perché?

```
x = [1, 3, 5, 4, 7]
for i in range(len(x)) : # i=...
    print("DEBUG:", x[i], x)
    x[i] *= 2 # x[i] = x[i] * 2
    print("DEBUG:", x[i], x)
print(x) # [2, 6, 10, 8, 14]
```

```
x = [1, 3, 5, 4, 7]
for e in x : # e=...
    print("DEBUG:", e, x)
    e *= 2 # e = e * 2
    print("DEBUG:", e, x)
print(x) # [1, 3, 5, 4, 7]
```


## ❑ Ragionare con "etichette" e "scatole" aiuta...

- Bisogna ricordare le assegnazioni (**implicite**) che avvengono nel ciclo **for**

# Errori di arrotondamento nel calcolo in virgola mobile

# Errori di arrotondamento


❑ Per oggetti di tipo **float** Python usa lo standard

 **IEEE 754 a 64 bit** che describe come **codificare** un numero "reale" mediante una sequenza binaria di 64 bit

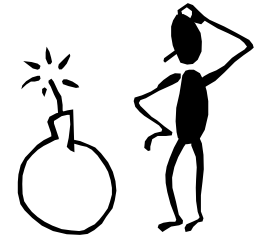
❑ Gli errori di arrotondamento sono un fenomeno **inevitabile** nel calcolo in virgola mobile eseguito con un numero *finito* di cifre significative, anche in base decimale

- calcolando  $1/3$  con due cifre significative, si ottiene 0.33
- moltiplicando 0.33 per 3, si ottiene 0.99 e non 1

❑ Siamo abituati a valutare questi errori pensando alla rappresentazione dei numeri in base *decimale*, ma i

 computer rappresentano i numeri in base *binaria* e a volte si ottengono dei risultati inattesi! (ma prevedibili...)

# Errori di arrotondamento



```
f = 4.35 # tipo float  
n = int(100 * f)  
print(n)
```



434



- ❑ Qui l'errore *inatteso* è dovuto al fatto che 4.35 non ha una *rappresentazione esatta* nel sistema binario usando (necessariamente...) un numero finito di cifre, proprio come 1/3 non ha una rappresentazione esatta nel sistema decimale
  - 4.35 viene rappresentato con un numero "un po' inferiore" a 4.35, che, quando viene moltiplicato per 100, fornisce un numero "un po' inferiore" a 435, quanto basta però per essere **troncato** a 434 dalla funzione **int**
- ❑ In generale, è meglio invocare **round**, eventualmente anche nella versione con due parametri (a meno che non si voglia effettivamente fare un troncamento)

# Confronto di numeri in virgola mobile

- ❑ I calcoli con numeri in virgola mobile possono introdurre errori di arrotondamento e/o troncamento
- ❑ Tali errori sono *inevitabili* e bisogna fare molta attenzione nella formulazione di verifiche che coinvolgono numeri in virgola mobile

```
r = 2**(1/2) # radice quadrata di 2
x = r**2      # "ovviamente" x vale 2
if x != 2 : print("Help") # Help
```

- ❑ Per fare in modo che gli errori di arrotondamento non influenzino la logica del programma, **i confronti per uguaglianza/diversità tra numeri di tipo float devono prevedere una tolleranza**
- ❑ Il modulo **math** in Python mette a disposizione una funzione `isclose(x, 2)` Restituisce **True** o **False** che gestisce proprio la necessaria tolleranza

# La libreria standard di Python



# La libreria standard di Python

- ❑ Oltre a **round**, **abs** e altre che abbiamo visto, ci sono moltissime funzioni matematiche utili, che **non** sono "predefinite" (*built-in*) nel linguaggio, ma fanno parte della **libreria standard** di Python
- ❑ Nella programmazione (in qualsiasi linguaggio), una **libreria** è una **raccolta di codice**, generalmente scritto da altri programmatori e pronto per essere utilizzato
  - In italiano forse diremmo meglio "biblioteca", che in effetti è la traduzione corretta del termine *library* usato in inglese, ma, ormai, in italiano è molto diffuso il termine libreria
- ❑ La **libreria standard** di uno specifico linguaggio di programmazione è una **libreria che viene considerata parte integrante del linguaggio stesso** e deve essere presente in qualsiasi pacchetto software che dichiari di essere un "ambiente di sviluppo per quel linguaggio"

# La libreria standard di Python

- ❑ La libreria standard di Python è organizzata in **moduli**
  - In altri linguaggi si parla spesso di "pacchetti"
- ❑ Ciascun modulo contiene la definizione di classi, funzioni, variabili e costanti che riguardano uno specifico campo applicativo
  - **math** – Modulo contenente funzioni e costanti matematiche
  - **string** – Modulo contenente funzioni che elaborano stringhe
  - **datetime** – Modulo contenente funzioni e costanti che aiutano a elaborare orari, date e intervalli temporali
  - ...
- ❑ Per poter utilizzare in un programma le risorse messe a disposizione da un modulo, bisogna *importare* tale modulo (o una sua parte)
  - La stessa sintassi si usa sia per i moduli della libreria standard sia per moduli che provengono da altre librerie "scaricate" a parte o per moduli progettati da noi (es. **ezgraphics**)

# Il modulo `math`

- ❑ Ad esempio, il modulo `math` contiene la definizione del valore di  $\pi$  con la massima precisione consentita dalle variabili `float` del linguaggio Python
  - In informatica non si può ottenere la "precisione infinita" tipica dell'algebra in campo reale, perché la memoria non può mai essere infinita

```
from math import pi  
print(pi) # visualizza 3.141592653589793
```

- ❑ Senza l'importazione si ha un errore

```
print(pi)
```



```
NameError: name 'pi' is not defined
```

# Il modulo **math**

- ❑ Il modulo **math** contiene molte risorse utili, ad esempio
  - Costanti
    - **pi**, valore di  $\pi$
    - **e**, valore della base dei logaritmi naturali
  - Funzioni trigonometriche dirette e inverse (**sin**, **cos**, **tan**, **asin**, **acos**, **atan**, ...)
  - Funzioni per numeri interi (**factorial**, **ceil**, **floor**, ...)
  - Funzioni esponenziali e logaritmiche (**log**, **log2**, **log10**, ...)
    - **sqrt** estrae la radice quadrata (*square root*), nome "storico", anche se in Python non serve (basta elevare a  $\frac{1}{2}$ )
    - **pow** eleva a potenza, altro nome "storico" che in Python non serve (riceve due argomenti: base ed esponente)
- ❑ Le informazioni complete si trovano nella **documentazione della libreria standard di Python**

# La libreria standard di Python

- ❑ Di solito gli enunciati di importazione si trovano all'inizio di un programma (ma è sufficiente che precedano il punto in cui le risorse vengono utilizzate)
- ❑ Un unico enunciato può importare **più risorse** (separate da virgole) **da uno stesso modulo**

```
from math import pi, sqrt, sin, cos
from string import capwords
from datetime import * # l'intero modulo
```

- ❑ Oppure si può importare un intero modulo, poi ai nomi delle singole risorse occorre aggiungere **il nome del modulo come prefisso**

```
import math
print(math.pi)
```

Alcuni preferiscono questo stile perché, per ogni risorsa, appare evidente quale sia il modulo che la definisce

# Funzioni predefinite in Python

- ❑ Le funzioni predefinite nel linguaggio Python (come **print**, **input**, **float**, **int**, ...) costituiscono in pratica un ulteriore modulo della libreria standard, ma non c'è bisogno di importarlo
  - Si chiama **builtins**
- ❑ Tutto funziona come se tale intero modulo fosse importato implicitamente all'inizio di qualsiasi programma `from builtins import *` # non serve
- ❑ Perché le funzioni non sono tutte predefinite?
  - All'aumentare delle funzioni note all'interprete Python, aumenta il tempo necessario alla traduzione del codice dei programmi (l'interprete deve sfogliare "dizionari" più voluminosi...)
    - Importare moduli e risorse non utilizzate non è un errore, ma è meglio non esagerare...
  - Inoltre, avere un unico modulo (cioè nessun modulo...) porrebbe problemi per lo "spazio dei nomi" (*namespace*)
    - non possono esserci due funzioni con lo stesso nome nello stesso modulo, ma possono esserci due funzioni con lo stesso nome in due moduli diversi (perché le si distingue con il nome del modulo)

**Lezione 26**  
**29/11/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Documentazione della libreria standard



# Documentazione della libreria standard

- ❑ La documentazione della libreria standard di Python versione **3** è consultabile all'indirizzo **`https://docs.python.org/3/`**
  - Seguire poi il collegamento "**Library Reference**"
  - È molto importante imparare ad analizzare tale documentazione, anche se all'inizio può sembrare un po' (oppure molto) criptica...
- ❑ Nella stessa pagina si trovano molte altre informazioni
  - Es. "Language Reference", utile per dissipare dubbi relativi alla sintassi e alla semantica del linguaggio
    - Es. quali caratteri possono far parte di un nome di variabile?
- ❑ Ovviamente è in inglese... 😊

La funzione help

# La funzione `help()`

- ❑ Usando la *shell* di Python (cioè l'interprete in modalità interattiva, anche all'interno di **idle**) può essere comoda la funzione **`help(...)`** che visualizza la documentazione **sintetica** di una funzione

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.

>>>
```

- ❑ Per ottenere informazioni su un modulo o su una sua funzione, bisogna prima importarlo

- ❑ Dopo aver importato un modulo, si possono anche avere informazioni sull'intero modulo (es. **`help(math)`**), ottenendo un elenco delle funzioni disponibili al suo interno. **Provate!**

```
>>> import math
>>> help(math.sqrt)
...
```

**Ancora stringhe**

# È un prefisso/suffisso...

- ❑ Vediamo altri **metodi** utili per stringhe
- ❑ Possiamo verificare se una stringa è una "sottostringa iniziale" (detta *prefisso*) di un'altra oppure se ne è una "sottostringa finale" (detta *suffisso*)

```
s = "topolino e minnie"  
if s.startswith("topo") : print("OK") # OK  
if s.endswith("ie")      : print("OK") # OK  
if s.endswith("e")       : print("OK") # OK
```

- Ovviamente se uno di questi due metodi restituisce **True**, altrettanto farebbe l'operatore **in** (ma non necessariamente viceversa)
- Codice equivalente senza questi metodi (molto meno chiaro):

```
s = "topolino e minnie"  
if s[:len("topo")] == "topo" : print("OK") # OK  
if s[-len("ie"):] == "ie"    : print("OK") # OK  
if s[-len("e"):] == "e"      : print("OK") # OK
```

# Il metodo `find`, alternativa a `index`

- Anche il metodo `find` può essere utile

```
s = "mototopo e autogatto"  
print(s.find("to")) # 2  
print(s.find("x")) # -1
```

- Se l'argomento **non** è una sottostringa della stringa con cui il metodo è stato invocato, restituisce **-1**
- Altrimenti restituisce il **minimo** indice (non negativo) in cui, nella stringa in cui si cerca, **inizia** una delle occorrenze della sottostringa cercata
- Ha lo stesso comportamento di `index`, ma restituisce **-1** quando `index` sollevarebbe l'eccezione **`IndexError`**
- Come `index`, ha anche la versione con due parametri, per specificare il punto di partenza della ricerca

# Altri metodi utili delle stringhe

- ❑ Le righe lette da un file di testo sono un po' "scomode", perché terminano sempre con un carattere `\n`
- ❑ Questo semplice programma dovrebbe **visualizzare il contenuto di un file**, ma non funziona come (distrattamente) previsto...

C:\python3 view.py names.txt

```
# view.py
from sys import argv
if len(argv) == 2 :
    inp = open(argv[1])
    for line in inp :
        print(line)
    inp.close()
```



```
1763452;Minnie
1122543;Mickey Mouse
1235436;Donald Duck
1324321;Daisy Duck
```

```
1763452;Minnie

1122543;Mickey Mouse

1235436;Donald Duck

1324321;Daisy Duck
```

- ❑ La funzione **print** aggiunge un carattere `\n`, ma già le stringhe terminano con `\n`: viene visualizzata una riga vuota dopo ogni riga di testo!

# Altri metodi utili delle stringhe

- ❑ Nell'esempio precedente possiamo ovviamente utilizzare la forma di **print** che NON aggiunge un carattere finale `\n` (cioè **end=""**) oppure possiamo fornire come argomento a **print** la sottostringa di **line** priva dell'ultimo carattere, cioè **line[:-1]**
- ❑ Invece, in generale, **vogliamo imparare a "ripulire" le stringhe da eventuali "caratteri di spaziatura" iniziali e/o finali**
  - I "caratteri di spaziatura" sono gli spazi, i caratteri `\n` e i caratteri di tabulazione (`\t`) [che non usiamo]
- ❑ Si possono usare i seguenti metodi
  - che, come al solito, restituiscono una nuova stringa, senza (ovviamente) modificare la stringa **s** che elaborano
  - **s.lstrip()** elimina spaziature iniziali (*left*)
  - **s.rstrip()** elimina spaziature finali (*right*)
  - **s.strip()** elimina spaziature iniziali e finali



# Altri metodi utili delle stringhe

- ❑ Si possono usare i seguenti metodi che restituiscono una nuova stringa
  - `s.lstrip()` elimina spaziature iniziali (*left*)
  - `s.rstrip()` elimina spaziature finali (*right*)
  - `s.strip()` elimina spaziature iniziali e finali
- ❑ Questi metodi possono anche ricevere una stringa come parametro tra parentesi: verranno eliminati tutti i caratteri iniziali e/o finali **uguali a quelli contenuti nella stringa ricevuta** come parametro tra parentesi
  - Es. `s.strip("ab")` restituisce una stringa uguale a `s` ma priva di eventuali caratteri `a` e `b` iniziali e/o finali, in qualsiasi ordine
    - `"aababccacabcaabbbaa".lstrip("ab")` restituisce  
`"ccacabcaabbbaa"`
    - `"aababccacabcaabbbaa".rstrip("ab")` restituisce  
`"aababccacabc"`
    - `"aababccacabcaabbbaa".strip("ab")` restituisce  
`"ccacabc"`
  - Quindi, la forma senza parametro equivale a usare il parametro `" \n\t"`
  - **Fare qualche esperimento!**  
In pratica, `lstrip` scandisce la stringa dall'inizio e, per ogni carattere, si chiede se appartenga alla stringa ricevuta come parametro esplicito, oppure no: se vi appartiene viene "eliminato", altrimenti il processo termina. Il metodo `rstrip` agisce a ritroso, dalla fine, e `strip` da entrambe le estremità.

# Altri metodi utili delle stringhe

- ❑ Un altro problema ricorrente è la scomposizione di una stringa in sottostringhe sulla base della presenza di caratteri di delimitazione

- ❑ Esempio: estrarre il numero di matricola in ciascuna riga

- ❑ **Ovviamente** 😊 non è difficile...

```
1763452;Minnie  
1122543;Mickey Mouse  
1235436;Donald Duck  
1324321;Daisy Duck
```

```
s = "1763452;Minnie"  
n = s[:s.index(";")] # n = "1763452"
```

- ❑ Ma se ciascuna riga ha molti dati, separati da ";", può essere noioso inserirli tutti in una lista...

- Es. ottenere la lista dei voti...
- Si fa (ovviamente) con un ciclo

```
1763452;30;28;30;26  
1122543;28;30;30;30  
1235436;24;18  
1324321;30;30;30;30
```

- ❑ Il metodo **split** è molto comodo!

```
s = "1763452;30;28;30;26"  
grades = s.split(";")[1:] # lista con i voti
```



# split

```
s = "ccabcccabzzzabxx"
ss = s.split("ab", 2)
# ss = ['cc', 'ccc', 'zzzabxx']
```

- ❑ Il metodo **split** **costruisce e restituisce una lista di stringhe** ottenuta "suddividendo" la stringa su cui opera in corrispondenza delle occorrenze della stringa ricevuta come parametro, detta **"separatore"**
  - **Può essere presente un numero intero  $n$  come secondo parametro:** verranno effettuate al massimo  $n$  suddivisioni, restituendo una lista di lunghezza massima  $n + 1$  il cui ultimo elemento contiene la porzione di stringa successiva all'ultima suddivisione (tale ultimo elemento, quindi, è l'unico della lista che può contenere il separatore al proprio interno)
  - **Se la stringa non contiene il separatore,** viene restituita una lista di lunghezza unitaria il cui unico elemento è la stringa intera originaria
  - **Se non viene fornito il separatore,** viene usato lo spazio, con un **comportamento aggiuntivo:** dalla lista sono escluse eventuali stringhe vuote (dovute alla presenza ripetuta del separatore, senza caratteri interposti)
- ❑ Il metodo **splitlines()** è una scorciatoia per **split("\n")**
  - Utile, ad esempio, per scomporre la stringa letta da file usando **read()**

```
s = "ab  c" # 2 spazi
ss = s.split(" ")
# ss = ['ab', '', 'c']
sss = s.split()
# sss = ['ab', 'c']
```

# Analisi di espressioni complesse

❑ A volte non è semplice comprendere espressioni complesse

```
s = "1763452;30;28;30\n"  
g = s[:-1].split(";")[1:]
```

❑ Può essere utile introdurre variabili temporanee

```
s = "1763452;30;28;30\n"  
a = s[:-1]           # a = ?  
b = a.split(";")     # b = ?  
g = b[1:]            # g = ?
```

❑ Al contrario, dopo aver progettato un'espressione che fa uso di variabili temporanee...

```
s = "1763452;30;28;30\n"  
a = s[:-1]  
b = a.split(";")  
g = b[1:]
```

❑ ... le possiamo eliminare, una dopo l'altra!

```
s = "1763452;30;28;30\n"  
g = s[:-1].split(";")[1:]
```

❑ **Senza esagerare...**

# Il metodo `join`

□ A volte ci troveremo di fronte al problema "inverso" di quello risolto dal metodo `split` delle stringhe

- Data una sequenza di stringhe, generare una stringa che ne sia la concatenazione, interponendo un separatore tra stringhe consecutive

- Ovviamente si può usare **un ciclo**...
- Oppure il metodo **`join`**!
  - attenzione a quale sia il suo parametro

```
seq = ('ab', 'xy', 'zzzz')
sep = "**"
s = seq[0]
i = 1
while i < len(seq) :
    s = s + sep + seq[i]
    i += 1
# s = "ab**xy**zzzz"

t = sep.join(seq)
# t = "ab**xy**zzzz"
```

# Il metodo `join`

```
seq = ['ab', 'xy', 'z']  
t = "".join(seq)  
# t = "abxyz"
```

- ❑ Se il separatore è **la stringa vuota**, il metodo `join` fa semplicemente la concatenazione delle stringhe presenti nella sequenza (**comunque più comodo di un ciclo...**)
- ❑ La sequenza può essere una lista/tupla di stringhe o... una stringa! In tal caso la stringa viene elaborata come sequenza di caratteri, che sono stringhe di lunghezza unitaria...

```
s = "abcd"  
t = "-".join(s)  
# equivale a t = "-".join(list(s))  
# t = "a-b-c-d"
```

- ❑ Se la sequenza ha un solo elemento, il metodo `join` restituisce una stringa uguale a tale unico elemento e il separatore non viene usato
- ❑ Se la sequenza è vuota, viene restituita la stringa vuota

# La funzione predefinita `sorted`

- ❑ Abbiamo visto che il metodo `sort` ordina la lista con cui viene invocato
- ❑ Più in generale, la funzione predefinita `sorted` è in grado di ordinare una sequenza di qualsiasi tipo
  - Per meglio dire, **riceve una sequenza (stringa, lista, tupla, range...)** e **restituisce una lista ordinata (sempre una lista) contenente gli stessi elementi**

```
s = sorted("acdb")  
# s = ['a', 'b', 'c', 'd']
```

- Quindi se voglio "ordinare una tupla"...

```
t = (4, 2, 3)  
t = tuple(sorted(t))  
# t = (2, 3, 4)
```

Ovviamente non ho ordinato una tupla, che è un oggetto immutabile... ho creato una nuova tupla (a partire da una lista ordinata) e l'ho assegnata alla stessa variabile!

# Progettazione di moduli: collaudo



# Codice di collaudo nei moduli

- ❑ Abbiamo visto che per progettare un modulo "importabile" è sufficiente scrivere le sue funzioni in un normale file sorgente, che non sarà eseguibile perché non contiene codice al di fuori delle funzioni che definisce
- ❑ Possiamo, invece, rendere il modulo "eseguibile" ?
- ❑ Sarebbe comodo, ad esempio, poterlo collaudare eseguendolo, senza dover necessariamente scrivere in un file separato un programma che lo collaudi
  - Un modulo che abbia il codice di collaudo al proprio interno è più facilmente "trasportabile" rispetto a una coppia di file...
  - Il problema è che **se il file contenente il modulo contiene anche codice eseguibile, questo viene eseguito ogni volta che il modulo viene importato**
    - **Importare un modulo** non è una funzione speciale dell'interprete: semplicemente, lo legge e lo esegue, quindi non deve contenere codice eseguibile, ma soltanto definizioni di funzioni/variabili/classi...

# Codice di collaudo nei moduli

- ❑ Vorremmo poter scrivere, in un modulo, codice che:
  - **viene eseguito** se il modulo viene messo in esecuzione da solo
  - **NON viene eseguito** se il modulo viene importato
- ❑ Ci viene in aiuto l'interprete: senza entrare nei dettagli, basta "proteggere" in questo modo il codice da eseguire

```
... # codice del modulo

if __name__ == "__main__" :

    # codice di collaudo del modulo
    ...

# __name__ è una variabile che viene
# definita dall'interprete
# PRIMA di leggere un file sorgente
# e vale "__main__" soltanto nel file
# sorgente nominato nella invocazione
# dell'interprete, non in quelli
# importati
```

**Convenzione  
(per noi un  
obbligo):**

anche se è  
sintatticamente  
consentito, non si  
può assegnare un  
valore a una  
variabile il cui  
nome inizia con  
"doppio  
underscore"

**Lezione 27**  
**03/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

Una cosa strana...

# Gli identificatori in Python

- ❑ Il fatto che ogni informazione sia un **oggetto** (un numero intero, una stringa, **la definizione di una funzione**) e che una variabile possa contenere l'indirizzo di un oggetto di qualsiasi tipo, ci consente di fare anche alcune cose "strane" che in altri linguaggi non sono possibili

- Possiamo creare un alias per il nome di una funzione

```
visualizza = print # pensare a "scatole/etichette"...  
visualizza("ciao") # come print("ciao")
```

- **Interessante:** possiamo **passare una funzione come parametro!**

```
def doWhatYouWant(x, func) :  
    print(func(x))  
  
doWhatYouWant(-2, abs)          # visualizza 2  
doWhatYouWant(2, float)        # visualizza 2.0  
import math  
doWhatYouWant(2, math.sqrt) # visualizza 1.4142135623730951
```

Il materiale necessario per il  
Laboratorio 08  
termina qui

Espressioni  
regolari o canoniche  
(*regular expression*)

# Una funzione `split` più potente...

- ❑ A volte si vogliono suddividere stringhe usando separatori più complessi rispetto a un'unica stringa separatrice...

- ❑ Esempio 

```
s = "http://python.org"
```

- ❑ Vorremmo ottenere queste tre componenti (le "parole"...)

```
lst = ['http', 'python', 'org']
```

- ❑ Come possiamo fare? Vorremmo scrivere qualcosa di questo tipo... 

```
lst = s.split("://" or ".") # NO!
```

- Cioè usare come separatore la stringa `://"` oppure la stringa `."` (ma la sintassi di questo esempio è **SBAGLIATA**)

- ❑ Si potrebbe immaginare una versione di `split` che accetti **una lista di separatori**, ma anche questo a volte non sarebbe sufficiente o, in generale, poco pratico... in realtà, in questo esempio, **vorremmo poter scrivere che il separatore è "qualsiasi sequenza di caratteri che non siano lettere"**



# Una funzione `split` più potente...

- ❑ Quello che ci serve è un linguaggio per descrivere i separatori!
  - Più in generale, ci serve  
**un linguaggio per descrivere "insiemi di stringhe",  
senza doverle elencare** esplicitamente
- ❑ È proprio quello che fanno le *espressioni regolari* o canoniche
  - *regular expression*, abbreviato *regexp* o *regex* o RE
- ❑ Nel modulo **re** di Python è definita una funzione **split** che accetta un'espressione regolare e una stringa, da suddividere usando qualunque separatore che sia descritto dall'espressione regolare

```
from re import split
s = "http://python.org"
regex = descrizione dell'insieme dei separatori
words = split(regex, s) # è una funzione...
# words = ['http', 'python', 'org']
```

# Espressioni regolari

- ❑ Il **linguaggio** usato per descrivere **"insiemi di stringhe"** è abbastanza complesso, vedremo solo alcune delle sue regole
  - Innanzitutto, dal punto di vista sintattico, in Python un'espressione regolare è **una stringa**, quindi non ci serve un nuovo tipo di dato per scrivere espressioni regolari: scriveremo stringhe
  - **Regola base:** un'espressione regolare costituita soltanto da **caratteri "non speciali"** descrive l'insieme contenente soltanto la stringa stessa (poi vedremo quali sono i caratteri speciali...)
    - Es. l'espressione regolare **"abcz"** descrive l'insieme di stringhe costituito dalla sola stringa **"abcz"**
    - Quindi, se alla **funzione `split`** forniamo come espressione regolare una stringa "normale", il suo comportamento è come quello del **metodo `split`**: c'è un solo separatore
  - Un carattere "speciale" preceduto dal carattere **\** perde le sue caratteristiche speciali e rappresenta se stesso
    - Regola "opposta" a quella usata nelle sequenze di escape
  - Ma **quali sono i caratteri speciali nelle espressioni regolari?**

## ❑ Caratteri "speciali" nelle espressioni regolari

- Il carattere `|` posto tra due sotto-espressioni regolari significa "oppure"
  - Es. `"ab | cd | z"` descrive l'insieme costituito dalle stringhe `"ab"`, `"cd"` e `"z"`
  - Es. `"ab \ | cd"` descrive l'insieme costituito dall'unica stringa `"ab | cd"` (il `\` rende "non speciale" il carattere `|`)
- Il carattere `.` significa **"un"** carattere qualsiasi tranne `\n`
  - Es. `"a."` descrive l'insieme costituito da tutte le stringhe di **due caratteri** che iniziano con il carattere `a`, tranne la stringa `"a\n"`
    - attenzione: due **caratteri**, non necessariamente due **lettere**... quindi, questa espressione regolare rappresenta un insieme contenente 65534 stringhe...
    - per comodità, non citerò più l'esclusione di `\n` dall'insieme di caratteri rappresentati da `.` (ma è importante ricordarlo)
  - Es. `".b. \."` descrive l'insieme costituito da tutte le stringhe di quattro caratteri che hanno `b` come secondo carattere e terminano con il carattere "punto", rappresentato, nell'espressione regolare, dalla sequenza `\.` che toglie al `.` il significato di "carattere speciale"

## □ Altri caratteri "speciali" nelle espressioni regolari

- Una coppia di parentesi quadre che racchiude caratteri significa "**un** carattere qualsiasi tra quelli indicati tra parentesi"
  - Es. "**a**[**bcd**]" descrive l'insieme costituito dalle stringhe "**ab**", "**ac**" e "**ad**" (cioè equivalente alla regex "**ab|ac|ad**")
- All'interno di una coppia di parentesi quadre, gli insiemi di caratteri possono anche essere espressi come "intervalli" di caratteri, indicando il primo e l'ultimo carattere dell'intervallo, separati da un trattino
  - Es. "**a**[**a-z**]" descrive l'insieme costituito dalle stringhe di due caratteri che iniziano con **a** e proseguono con una lettera minuscola
  - Es. "**a**[**a-zA-Z0-9;**]" descrive l'insieme costituito dalle stringhe di due caratteri che iniziano con **a** e hanno come secondo carattere una lettera (minuscola o maiuscola) oppure una cifra numerica o un "punto e virgola"
  - **Per identificare quali siano i caratteri interni all'intervallo si usa la codifica Unicode** ma gli intervalli più utilizzati sono **a-z**, **A-Z** e **0-9**
  - In ogni caso, una coppia di parentesi quadre rappresenta sempre **un solo** carattere (scelto tra quelli indicati al suo interno)

## ❑ Altri caratteri "speciali" nelle espressioni regolari

- Se il **primo carattere** all'interno di una coppia di parentesi quadre è ^ ("accento circonflesso", in inglese si chiama *caret*), la coppia di parentesi significa "**un** carattere qualsiasi che **NON** sia tra quelli indicati tra parentesi"

- Es. "**a** [ ^ **a-z** ] " descrive l'insieme costituito dalle stringhe di due caratteri che iniziano con **a** e proseguono con qualunque carattere che **non** sia una lettera minuscola
- Es. "**a** [ ^ **a-zA-Z** ] . " descrive l'insieme costituito dalle stringhe di tre caratteri che iniziano con **a**, terminano con un carattere qualsiasi (rappresentato dal . fuori dalle parentesi quadre) e hanno al centro un carattere qualsiasi che **non** sia una lettera minuscola né una lettera maiuscola

## ❑ Nota: **attenzione a non aggiungere spazi "qua e là"** in una regex perché uno spazio rappresenta... uno spazio!

- Es. "**a** [ ^ **a-z** ] " descrive l'insieme costituito dalle stringhe aventi **TRE** caratteri: tutte iniziano con la lettera **a** **seguita da uno spazio** e proseguono con qualunque carattere che **non** sia una lettera minuscola o uno spazio

□ Altri caratteri "speciali" nelle espressioni regolari, che agiscono come operatori **postfissi**

■ Se la **descrizione di un carattere** è **seguita** da:

- **+** significa "una o più" volte
  - Es. "**ab+**" descrive l'insieme (infinito) costituito dalle stringhe "**ab**", "**abb**", "**abbb**", "**abbbb**", ecc
  - Es. "**[0-9]+a**" descrive l'insieme (infinito) costituito dalle stringhe che terminano con **a** e iniziano con un numero qualsiasi di cifre (ma almeno una cifra), come "**504a**" e "**9a**" ma non "**a**"
- **\*** significa "zero o più" volte, cioè una **presenza opzionale**
  - Es. "**ab\***" descrive l'insieme (infinito) costituito dalle stringhe "**a**", "**ab**", "**abb**", "**abbb**", "**abbbb**", ecc
  - Es. "**a[0-9]\***" descrive l'insieme (infinito) costituito dalle stringhe che iniziano con **a** e proseguono con un numero qualsiasi di cifre (anche nessuna cifra)
- **?** significa "zero o una" volta, cioè una **presenza opzionale ma non ripetuta**
  - Es. "**a[0-3]?**" descrive l'insieme costituito dalle stringhe "**a**", "**a0**", "**a1**", "**a2**", "**a3**"
  - **Quale insieme di stringhe è descritto dalla seguente regex?**  
**"a[b-e]?[0-3]\*[xyz]+b"**



## □ Altri operatori postfissi nelle espressioni regolari

### ■ Se la descrizione di un carattere è seguita da:

- **{m}** , dove **m** è un numero intero, significa

"il carattere precedente esattamente **m** volte"

- Es. "**a** [0-9] {3}" descrive l'insieme costituito dalle stringhe di 4 caratteri, il primo dei quali è **a** e gli altri tre sono cifre qualsiasi (anche diverse tra loro), equivalente a "**a** [0-9] [0-9] [0-9]"

- **{m,n}** , dove **m** e **n** sono numeri interi, significa

"il carattere precedente almeno **m** e al massimo **n** volte"

- Es. "**a** [0-9] {2,4}" descrive l'insieme costituito dalle stringhe aventi **a** come primo carattere, seguito da 2, 3 o 4 cifre (anche diverse tra loro), equivalente a

"**a** [0-9] [0-9] | **a** [0-9] [0-9] [0-9] | **a** [0-9] [0-9] [0-9] [0-9]"

- Se manca **n** (ma c'è la virgola), si intende "n infinito"

Es. "**a** [0-9] {4,}" descrive l'insieme (infinito) costituito dalle stringhe aventi **a** come primo carattere, seguito da almeno 4 cifre (anche diverse tra loro), equivalente a

"**a** [0-9] [0-9] [0-9] [0-9] +"

- Quindi **+** **\*** **?** sono soltanto abbreviazioni

- **+** equivale a {1,}, **\*** equivale a {0,}, **?** equivale a {0,1}

# Espressioni regolari

Linguaggio molto  
più articolato

## □ Altri caratteri "speciali" nelle espressioni regolari

- Una coppia di **parentesi tonde** può essere usata per **raggruppare una sotto-espressione regolare**, aggiungendo il modificatore **? :** dopo la parentesi tonda aperta
- I "moltiplicatori" postfissi (**{m,n}** e simili) che **agiscono sul singolo carattere che li precede**, agiscono invece su un'intera sotto-espressione se questa è racchiusa tra parentesi tonde
  - "**a[0-9]{3}**" equivale a "**a[0-9][0-9][0-9]**", quindi descrive l'insieme costituito dalle stringhe di 4 caratteri, il primo dei quali è **a** e gli altri tre sono cifre qualsiasi
  - "**(?:a[0-9]){3}**" equivale invece a "**a[0-9]a[0-9]a[0-9]**", quindi descrive l'insieme costituito dalle stringhe di 6 caratteri, costituite da tre sottostringhe di due caratteri: in ciascuna coppia, il primo carattere è **a** e il secondo è una cifra qualsiasi. Ad esempio: **a0a2a1, a0a0a0, a4a3a7**



# Una funzione `split` più potente...

- ❑ Torniamo al nostro esempio, che ora possiamo completare
- ❑ Come descriviamo un insieme di separatori costituito da **"qualsiasi sequenza di caratteri che non siano lettere (sottinteso, né maiuscole né minuscole)"** ?

```
s = "http://python.org"
regex = "[^a-zA-Z]+"
from re import split
words = split(regex, s)
# words = ['http', 'python', 'org']
```

# La funzione `findall`

- ❑ La funzione `findall` del modulo `re` restituisce una lista con le sottostringhe che **corrispondono** all'espressione regolare
  - nell'ordine in cui le trova dall'inizio alla fine della stringa (lista vuota se non trova corrispondenze), **senza sovrapposizioni** (cioè inizia a cercare la seconda stringa a partire dal carattere successivo all'ultimo della prima stringa trovata, e così via...)

```
from re import findall
s = "http://python.org"
regex = "[a-zA-Z]+" # parole
words = findall(regex, s) # cerca le parole
# words = ['http', 'python', 'org']
regex = "[^a-zA-Z]+" # separatori
seps = findall(regex, s) # cerca i separatori
# seps = ['://', '.']
```

- A volte è più comodo descrivere i separatori, altre volte è più comodo descrivere ciò che si cerca (in questo caso è sostanzialmente equivalente)

# I moltiplicatori sono "golosi"

- Nelle espressioni regolari, con l'uso dei moltiplicatori con molteplicità indefinita (es. `+`, `*`, `{qualcosa, }`) si possono verificare ambiguità, ovviamente da **eliminare**

```
from re import findall
s = "cabbbx"
regex = "ab+" # a seguita da "almeno una b"
words = findall(regex, s)
# trova "ab" oppure "abb" oppure "abbb" ?
# sono tutte stringhe che corrispondono alla
# descrizione della regex...
```

- Nell'utilizzo delle espressioni regolari, si risolvono queste ambiguità usando un **approccio greedy** ("goloso"): **viene identificata la stringa più lunga** tra quelle che sono descritte dall'espressione regolare
  - Nel caso qui riportato, **findall** trova **"abbb"**
  - Lo stesso fenomeno c'era nell'esempio precedente di utilizzo di **split**, dove il primo separatore veniva identificato come **" : / /"** e non soltanto **" : "** oppure **" : /"**

**Lezione 28**  
**04/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Insieme in Python

# Insieme

- ❑ Un insieme è un contenitore di dati che memorizza **una raccolta di valori univoci**, cioè senza duplicati
  - Naturalmente un insieme può anche essere memorizzato in una lista/tupla, ma (come vedremo) gli insiemi di Python hanno il vantaggio che *impediscono* automaticamente l'inserimento di elementi duplicati
- ❑ Diversamente da quanto accade in una lista o in una tupla, **gli elementi di un insieme non vengono memorizzati in alcun ordine specifico e NON vi si può accedere tramite un indice associato alla loro posizione**
  - Per questa caratteristica, **gli insiemi possono essere (e sono) implementati in Python in modo più efficiente delle liste/tuple**, quindi tutte le volte che dobbiamo memorizzare una raccolta di dati (tutti diversi) dei quali non ci interessa la posizione reciproca, è meglio usare un insieme
    - Analogamente, quando sappiamo che una sequenza non verrà modificata, è preferibile utilizzare una tupla rispetto a una lista



# Creare un insieme

- ❑ Per creare un insieme contenente alcuni dati iniziali, si usa una coppia di parentesi **graffe** contenenti i dati separati da virgole

```
s = {"pippo", "pluto", "topolino"}
t = {1, 4, 2, 6}
x = {-2, "minnie", (3, 4)}
# uno degli elementi dell'insieme x è una tupla
# (non è "il terzo elemento", non ha senso...)
```

- ❑ **Non è necessario che gli insiemi siano raccolte omogenee**

- ❑ **Gli elementi di un insieme devono essere oggetti immutabili** (*hashable*, in "pythonese"...)
  - Quindi anche una tupla, ma deve contenere a sua volta oggetti immutabili... non una tupla di liste... (provare **x={ ([ ] ) }**...)

- Errore: *unhashable type*...

- Quindi non si può costruire un insieme di insiemi né un insieme di liste... invece si può (ovviamente?) costruire una lista/tupla di insiemi... perché si può costruire una lista/tupla di **qualsiasi** tipo di oggetti

# Creare un insieme

- ❑ La funzione **set** genera un insieme a partire da un contenitore qualsiasi (lista, tupla, stringa, range o altro insieme) di oggetti immutabili

```
lst = [1, 2, 4]
s = set(lst)
```

- ❑ Per generare un insieme vuoto **non** si può scrivere così



```
s = { }
```



Pericoloso perché NON è un errore di sintassi: come vedremo, crea un **dizionario** vuoto, una struttura molto diversa da un insieme

ma si può scrivere così

```
s = set()
```

- ❑ Anche se a volte sarebbe naturale, soprattutto in esempi sintetici come questi, non usiamo **list** come nome di una variabile che contiene una lista e non usiamo **set** come variabile che contiene un insieme... **già sappiamo perché...**



# Visualizzare un insieme

- Naturalmente si può chiedere alla funzione `print` di visualizzare un insieme, ma **l'ordine in cui i suoi elementi vengono visualizzati è praticamente "imprevedibile"**

- L'ordine non è "casuale" ma dipende da regole complesse che non vediamo perché non è utile saperlo (dipende dalla tecnica di *hashing*)



```
t = {"pippo", "pluto", "topolino"}  
print(t)  
# {'topolino', 'pluto', 'pippo'}
```

- **Se mi interessa sapere in che ordine vengono visualizzati gli elementi di un insieme... significa che un insieme non è il contenitore adatto!**  
Evidentemente mi serve una sequenza (lista o tupla)

# Scandire un insieme

- ❑ Per scandire gli elementi di un insieme non possiamo usare un ciclo con indice
  - **l'accesso agli elementi usando un indice tra parentesi quadre NON FUNZIONA con gli insiemi**, perché **gli insiemi non sono sequenze...**  
gli elementi di un insieme non hanno una "posizione"
- ❑ Dobbiamo usare un ciclo **for** che usi direttamente l'insieme come contenitore

```
t = {"pippo", "pluto", "topolino"}  
for name in t :  
    fai qualcosa con name ma  
    non si sa in quale ordine name assumerà  
    i diversi valori presenti nell'insieme
```

# Funzioni e operatori per insiemi

- ❑ Come per qualsiasi contenitore, si può conoscere la dimensione di un insieme (cioè la sua cardinalità) usando la funzione **len**

```
t = {"pippo", "pluto", "topolino"}  
print(len(t)) # 3
```

- ❑ Per verificare se un elemento è presente in un insieme si possono usare gli operatori **in** e **not in**

```
t = {"pippo", "pluto", "topolino"}  
if "pippo" in t :  
    print("OK") # OK  
if "paperino" not in t :  
    print("OK") # OK
```

# Insiemi immutabili

- ❑ La funzione **frozenset** crea un **insieme immutabile** a partire da qualsiasi contenitore (anche da un insieme)

- Gli insiemi immutabili sono più efficienti degli insiemi

```
t = {2, 4, 1}
s = frozenset(t)
```

- ❑ Con gli insiemi immutabili si possono fare le stesse cose che si fanno con gli insiemi, tranne quelle (che vedremo) che provocherebbero modifiche al loro contenuto

- Si può, quindi, costruire un insieme di insiemi immutabili...

- ❑ Non c'è una forma letterale di tipo "insieme immutabile": le parentesi graffe creano sempre un insieme, non un insieme immutabile

```
s = frozenset({1, 4, -1})
```

# Metodo add

□ Dato che gli insiemi sono oggetti, con essi possiamo invocare metodi

□ Per aggiungere un elemento a un insieme si usa il metodo **add**

```
t = {2, 5, 4}
print(len(t)) # 3
t.add(1)
print(len(t)) # 4
```

□ Se l'elemento è già presente nell'insieme, non viene aggiunto, la cardinalità dell'insieme non cambia e non si verifica alcun errore

```
t = {2, 5, 4}
t.add(2)
print(len(t)) # 3
```

□ La funzione **set** invoca ripetutamente **add** a partire da un insieme vuoto, quindi (ovviamente) crea un insieme privo di duplicati

```
s = set("pippo")
print(s)
# {'o', 'p', 'i'}
```

# Metodi **discard** e **remove**

- ❑ I metodi **discard** e **remove** eliminano dall'insieme l'elemento ricevuto come argomento (ovviamente non ricevono indici)
- ❑ Se l'elemento non è presente nell'insieme
  - **discard** "fallisce silenziosamente", cioè non fa niente e non modifica l'insieme
  - **remove** solleva l'eccezione **KeyError**
    - L'eccezione ha un nome un po' "strano", che dipende da come sono realizzati gli insiemi (cfr. *hashing*, non vediamo)
- ❑ Il metodo **clear** rende vuoto l'insieme su cui agisce ed è più rapido di uno svuotamento mediante ripetute eliminazioni

# Confronto tra insiemi

- ❑ Due insiemi (di tipo *set* e/o *frozenset*) sono uguali se contengono gli stessi elementi
  - per il confronto si usano gli operatori `==` e `!=`
  - **[incoerenza nel linguaggio]** anche un *set* e un *frozenset* possono essere uguali tra loro, se hanno lo stesso contenuto, mentre liste e tuple non sono mai uguali
  - l'ordine tra gli elementi è ovviamente ininfluente, non essendo definito
- ❑ Per verificare se un insieme è un sottoinsieme di un altro insieme si può usare il metodo **`issubset`**

Lo stesso risultato  
si ottiene  
confrontando gli  
insiemi con  
l'operatore `<=`

```
t = {"pippo", "pluto", "topolino"}
s = {"pippo", "topolino"}
if s.issubset(t) :
    print("OK")
if s <= t : # sottoinsieme?
    print("OK")
```

# Ancora metodi per insiemi

- ❑ Con gli insiemi si possono invocare metodi che vogliono come parametro un altro insieme e realizzano le ben note operazioni tra insiemi

- ❑ Restituisce l'insieme unione (che, ovviamente, non contiene duplicati)

```
s = {1, 2}
t = {2, 3, 0}
u = s.union(t)
# u = {0, 1, 2, 3}
```

- ❑ Restituisce l'insieme intersezione

```
s = {1, 2}
t = {2, 3, 0}
i = s.intersection(t)
# i = {2}
```

- ❑ Restituisce l'insieme differenza (ripassare la definizione... ☺), operazione NON commutativa

```
s = {1, 2}
t = {2, 3, 0}
d = s.difference(t)
# d = {1}
```

- ❑ Nessuno di questi metodi modifica gli insiemi su cui opera



# Liste o insiemi: prestazioni!

- ❑ Ribadiamo che tutto ciò che si può fare con un insieme si potrebbe fare con una lista/tupla che memorizzi gli elementi dell'insieme
- ❑ Perché in Python usiamo gli oggetti di tipo **set** ? **Per efficienza!**

- ❑ Problema: data una lista di parole, costruire la lista delle parole uniche (dove, quindi, ogni parola è presente una volta sola)

```
ws = [...] #stringhe (immutabili)
wList = list() # uso una lista
for w in ws :
    if w not in wList :
        wList.append(w)
wSet = set() # uso un insieme
for w in ws :
    wSet.add(w) # non verifico!!
wList2 = list(wSet)
# wList e wList2 contengono gli
# stessi elementi e, dopo averle
# ordinate, sono identiche
```

- ❑ Esperimento:  
parole nella  
Divina Commedia

- ❑ Tempo con insieme: 12ms. Tempo con lista: 2257ms.

**Circa 190 volte più lento**

(es. si passa da un minuto a più di 3 ore, da un'ora a 8 giorni...)

# Come si misura il tempo d'esecuzione?

- ❑ Il tempo di esecuzione di un algoritmo va misurato all'interno del programma
- ❑ Si può usare la funzione `time` del modulo `time` che, a ogni invocazione, restituisce un valore di tipo `float` che rappresenta...
  - il numero di secondi trascorsi da un evento di riferimento (*la mezzanotte del 1 gennaio 1970*) !!
- ❑ Ciò che interessa è la *differenza* tra due valori
  - si invoca la funzione *prima e dopo* l'esecuzione dell'algoritmo (escludendo le operazioni di input/output dei dati)

```
from time import time
# fase di input dei dati
...
beginTime = time()
Esecuzione del codice che voglio misurare
elapsedTime = time() - beginTime
print("%f secondi" % elapsedTime)
# fase di output dei dati
...
```

# Come sono fatti gli insiemi?



❑ Per ottenere prestazioni così eccellenti, gli insiemi sono realizzati con una struttura di memorizzazione dei dati chiamata **tabella hash** (una "lista di liste" organizzate in modo speciale...)

❑ Perché non usiamo sempre gli insiemi?

- Semplicemente perché memorizzano informazioni diverse da quelle memorizzate dalle liste!
- **Gli insiemi non memorizzano una sequenza**: i dati non appartengono a una relazione precedente/successivo
- Gli insiemi non memorizzano la molteplicità (in una sequenza, invece, il fatto che uno stesso dato sia presente più volte può essere rilevante, es. "**rossa**" vs. "**rosa**")
- Gli insiemi memorizzano soltanto oggetti immutabili (è un limite della tecnica di *hashing* con cui sono implementati)

❑ Però, ovviamente, **in tutti quei casi in cui gli insiemi sono "sufficienti", vanno utilizzati!**

**L'oggetto None**

# Funzioni e None

- ❑ Quando una funzione termina eseguendo **return** senza che venga fornita un'espressione di cui restituire il valore (oppure termina eseguendo un **return** implicito), cosa restituisce?
- ❑ Abbiamo detto che "non restituisce niente", in realtà **restituisce sempre "qualcosa"**. Cosa?
  - Restituisce un oggetto rappresentato dal letterale **None** (*niente*), di un tipo così speciale che... non rappresenta alcun dato
- ❑ Cosa si può fare con l'oggetto **None**?
  - **Lo si può soltanto confrontare con se stesso!**
  - Oppure passare alla funzione **print**... che visualizzerà... provate!
  - Si può progettare una funzione che, **quando** non ha un valore valido da restituire, restituisce **None** (basta che esegua **return** senza espressione)
    - L'invocante, sapendolo, controlla il valore restituito e si comporta di conseguenza (attenzione, l'oggetto **None**, non la stringa "**None**")
    - Alternativa al sollevamento di un'eccezione, in caso di errore meno grave
- ❑ **Forse non useremo questa caratteristica in funzioni progettate da noi, ma è bene conoscere l'oggetto **None** perché potrebbe comparire "per errore" durante l'attività di debugging**

```
x = func()
if x == None :
    ...
```

Restituire liste/tuple di  
lunghezza prefissata

# Ancora liste/tuple come valori restituiti

```
def minmax(ss) :  
    # trova nella lista di stringhe ss  
    # la stringa più corta (x) e la stringa più lunga (y)  
    ...  
    return (x, y) # tupla di lunghezza 2
```

```
s = minmax(...)  
a = s[0]  
b = s[1]  
# non userò più s...
```

- ❑ A volte una funzione restituisce una lista/tupla **con lunghezza prefissata**
  - In pratica, è un "trucco" per restituire più di un valore
- ❑ L'invocante, anziché assegnare il riferimento ricevuto a una variabile (che farebbe poi riferimento alla tupla/lista), può assegnare la tupla/lista ricevuta **a una tupla/lista di variabili**

```
(a, b) = minmax(...)  
# anche senza parentesi...  
a, b = minmax(...)
```

Sintassi strana

- ❑ È comodo tutte le volte in cui la tupla/lista viene restituita come "trucco" per restituire più valori, ma, nel codice invocante, non si ha intenzione di elaborare tale struttura come se fosse una sequenza

# Dizionari in Python



# Dizionario (o mappa)



- Un dizionario (o mappa) è un contenitore di dati che memorizza **una raccolta di coppie di tipo chiave/valore**, cioè ogni dato è, in realtà, una coppia di due elementi che hanno ruoli diversi
  - La **chiave** (*key*) è l'elemento che identifica **univocamente** la coppia all'interno del dizionario (cioè **non ci possono essere due coppie con la stessa chiave**) e tecnicamente **deve essere un oggetto immutabile** (numero, stringa, *frozenset*, tupla di oggetti immutabili...)
  - Il **valore** (*value*) è un dato di qualsiasi tipo, **anche modificabile**, associato alla chiave all'interno di una coppia (possono esistere più coppie aventi lo stesso valore, purché abbiano chiavi diverse)
- Es. le chiavi sono numeri di matricola di studenti, i valori sono liste con i voti ottenuti da ciascuno (*sintassi in seguito*)



$d = \{ 123432 : [30, 28, 22] , \quad 112232 : [21, 27] \}$

**Lezione 29**  
**06/12/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Dizionario

- ❑ Tutto quello che si può fare con un dizionario si potrebbe fare
  - con una lista di coppie (cioè una lista di liste/tuple di dimensione due), ciascuna contenente una chiave e un valore
  -  ■ con due liste: una lista di chiavi e una lista di valori, in posizioni corrispondenti (una tecnica detta "liste parallele")
  - in altri modi...
- ❑ Ma **con il dizionario l'elaborazione è più efficiente!**
  -  ■ Come gli insiemi, i dizionari in Python sono realizzati usando una tabella hash
- ❑ Come un insieme, anche **un dizionario NON è una sequenza**: non preserva alcuna relazione tra le coppie o tra le chiavi (non c'è la "prima" coppia...)

# Creare un dizionario

- ❑ Per creare un dizionario contenente alcune coppie iniziali, si usa una coppia di **parentesi graffe**, come per creare un insieme
- ❑ Le coppie sono separate da virgole (come in ogni contenitore) e ciascuna coppia ha il formato ***chiave : valore***

```
d = {123432:"Mickey Mouse",  
     112232:"Daisy Duck"}
```

```
e = {"Mickey Mouse": [20, 30, 18],  
     "Daisy Duck": [30, 30, 28, 24],  
     "Goofy": [18, 20, 21]}
```

**Si capisce che  
non è un insieme  
perché ci sono i  
"due punti"**

- ❑ Come già detto, **le chiavi di un dizionario devono essere oggetti immutabili** (mentre sui valori non ci sono vincoli)

# Creare un dizionario vuoto

- ❑ La funzione `dict()` (*dictionary*) crea un dizionario vuoto

```
d = dict()
```

- ❑ Per creare un dizionario vuoto si potrebbe anche scrivere così

```
d = { }
```



ma evitiamo di farlo perché (per un lettore distratto) non è ben chiaro se venga creato un insieme o un dizionario (sappiamo che viene creato un dizionario, ma ricordarlo non è scontato...)

- ❑ La funzione `dict` crea anche dizionari a partire da altri contenitori, ma la sintassi è un po' più complicata del solito...

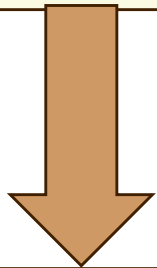
- Comoda, invece,  
per **clonare** un dizionario

```
d = { .. : .. , .. : .. }  
e = dict(d)  
# cosa fa e = d ?
```

# Visualizzare un dizionario

- Naturalmente si può chiedere alla funzione **print** di visualizzare un dizionario, ma l'ordine in cui le sue coppie vengono visualizzate è praticamente imprevedibile
  - non è "casuale" ma dipende da regole complesse che non vediamo (come negli insiemi...)

```
d = {123432: "Mickey Mouse", \
      112232: "Daisy Duck"}
print(d) # invoca str(d)
```



Come al solito, **str** genera una stringa avente il formato che sarebbe adatto a inizializzare quell'oggetto

```
{123432: 'Mickey Mouse', 112232: 'Daisy Duck'}
```

# len, in e not in per dizionari

- ❑ Come per qualsiasi contenitore, si può conoscere la dimensione di un dizionario usando la funzione **len**

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}  
print(len(d)) # 2
```

- ❑ La lunghezza di un dizionario è il numero di coppie che contiene
- ❑ Per verificare se in un dizionario esiste una coppia avente una determinata chiave si possono usare gli operatori **in** e **not in**

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}  
if 112232 in d : print("OK") # OK
```

- Cerco chiavi, non valori o coppie
- ❑ Da questo esempio deduciamo che un dizionario, usato come contenitore (ad esempio alla destra degli operatori **in** e **not in**), si comporta come contenitore delle sue sole **chiavi**
  - Vedremo la stessa proprietà nei cicli **for**

# Dizionari e operatore [ ]

- ❑ Con i dizionari si può usare **l'operatore "parentesi quadre"**, ma al suo interno non si mette un indice (dato che, come ricordiamo, **il dizionario non è una sequenza**) bensì **una chiave**

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
name = d[112232] # name = "Daisy Duck"
e = {"Mickey Mouse": [20, 30, 18],
     "Daisy Duck": [30, 30, 28, 24],
     "Goofy": [18, 20, 21]}
grades = e["Goofy"] # grades = [18, 20, 21]
```

e si ottiene il **valore** corrispondente alla chiave

- ❑ Se nel dizionario **non** è presente una coppia con la chiave indicata, viene sollevata l'eccezione **KeyError**
  - Spesso si condiziona l'accesso usando una verifica con operatore **in**

```
if 112232 in d :
    name = d[112232]
```



# Metodo get

- ❑ Problema che vogliamo risolvere: "se nel dizionario c'è una coppia con chiave **key**, allora la variabile **name** conterrà il valore associato alla chiave, altrimenti conterrà un valore predefinito"

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
key = int(input("Matricola da ispezionare: "))
if key in d :
    name = d[key]
else :
    name = "Sconosciuto"
print(name)
```

- ❑ Per questo scopo si può usare il metodo **get**, che **riceve una chiave da cercare e un valore da restituire se la chiave è assente** (altrimenti restituisce il valore associato alla chiave)

```
name = d.get(key, "Sconosciuto")
```

- ❑ Se il metodo **get** riceve solo la chiave e tale chiave è assente, restituisce l'oggetto speciale **None** (ma non solleva eccezione)

# Ancora sull'operatore [ ]

- ❑ L'operatore "parentesi quadre" può essere usato, come nelle liste, anche per **modificare il valore** associato

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}  
d[123432] = "Mr. Mickey Mouse"  
# modifica il valore presente nella coppia  
# avente la chiave specificata
```

- ❑ Se il dizionario **non** contiene una coppia con la chiave specificata, **NON è un errore**
  - Viene **aggiunta** una nuova coppia con quella chiave e quel valore!

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}  
d[121212] = "Goofy"  
# ora il dizionario ha dimensione 3
```

# Contenitori e operatore [ ]

## ❑ **Bisogna fare attenzione**

- L'operatore [ ] ha **una semantica completamente diversa** quando viene applicato a un dizionario rispetto a quando viene applicato a una sequenza (stringa, lista, tupla)
    - Con gli insiemi non si può usare
  - Usato con i dizionari contiene **una chiave**, negli altri casi contiene **un indice**
  - **Si rischia di fare confusione soprattutto quando un dizionario usa numeri interi come chiavi**, perché sembrano indici (ma non lo sono)
- ❑ L'operatore *slicing* (cioè parentesi quadre con **due** indici separati da : ) **non si può usare con i dizionari**, perché non sono sequenze e, quindi, non avrebbe senso
- (ovviamente) non si può usare nemmeno con gli insiemi

# Metodo pop

- ❑ Per eliminare una coppia da un dizionario si invoca il metodo **pop**, fornendo la chiave come argomento

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
key = int(input("Matricola da eliminare: "))
if key in d :
    name = d.pop(key)
    # la dimensione del dizionario diminuisce
```

- ❑ Il metodo **pop**
  - restituisce il valore presente nella coppia eliminata (ovviamente tale valore può essere ignorato dall'invocante)
  - solleva l'eccezione **KeyError** se non esiste una coppia con la chiave specificata

# Funzione `sorted`

- ❑ Come sappiamo, la funzione `sorted` restituisce una lista ordinata avente lo stesso contenuto del contenitore che riceve come argomento
- ❑ Il suo funzionamento è intuitivo con stringhe, liste, tuple e insiemi
- ❑ Cosa restituisce ricevendo un dizionario?
  - Forse una lista di tuple (le coppie), ordinate in base alla chiave?
  - No, **una lista ordinata contenente soltanto le chiavi**

```
d = {123432: "Mickey Mouse", 112232: "Daisy Duck"}  
print(sorted(d)) # [112232, 123432]
```

- ❑ Ovviamente non modifica in alcun modo il dizionario ricevuto

# Scandire un dizionario

- ❑ Per scandire gli elementi di un dizionario non possiamo usare un ciclo con indice, perché **l'accesso agli elementi usando un indice tra parentesi quadre non funziona con i dizionari**: dobbiamo usare un ciclo **for**
  - In un ciclo **for**, un dizionario si comporta come un **contenitore delle sue chiavi**

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
# visualizza i nomi degli studenti
for key in d :
    print(d[key])
# visualizza matricole e nomi
for key in d :
    print(key, d[key])
# l'ordine in cui le chiavi vengono assegnate,
# una dopo l'altra, alla variabile key non è
# definito (ma è adatto a rendere efficienti le
# operazioni sul dizionario)
```

# Scandire un dizionario

- ❑ Applicando la funzione **sorted** a un dizionario si ottiene **una lista ordinata delle chiavi**, che può essere usata come contenitore in un ciclo **for**

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
# visualizza i nomi degli studenti
# in ordine di matricola crescente
for key in sorted(d) :
    print(d[key])
# visualizza matricole e nomi
# in ordine di matricola crescente
for key in sorted(d) :
    print(key, d[key])
```

- ❑ Ricordiamo sempre che la funzione **sorted** **non modifica il contenitore che riceve**, ma restituisce una lista con il suo contenuto ordinato (se il contenitore è un dizionario, restituisce una lista con le chiavi ordinate)

# Scandire un dizionario

- ❑ Più tecnicamente, **quando si usa un dizionario in una posizione in cui è richiesto un contenitore** (come operando destro di `in` e `not in`, in un ciclo `for`, come argomento delle funzioni `sorted` o `list/tuple/set`, ecc.) il contenitore che viene effettivamente utilizzato è **"il contenitore delle chiavi del dizionario"** che viene ottenuto dall'interprete invocando **implicitamente** il metodo **`keys()`** del dizionario
  - Possiamo anche usare il metodo **`keys()`** in modo esplicito

```
d = {123432: "Mickey Mouse", 112232: "Daisy Duck"}  
for key in d.keys() : # come for key in d :  
    ...
```

- ❑ Di che tipo è il contenitore restituito da **`keys()`** ?
  - È **simile a una lista**, è di tipo **`dict_keys`**, ma non ci interessa: ci basta sapere che **è una sequenza**
    - Es. Passandolo a **`list`**, lo si "trasforma" in una vera lista



# Scandire un dizionario

- ❑ E se volessimo visualizzare il contenuto di un dizionario seguendo l'ordine dei **valori**?
  - In questo esempio, i nomi degli studenti in ordine lessicografico
  - Usiamo in modo analogo il contenitore restituito dal metodo **values()** (che è di tipo **dict\_values**, ma ci basta sapere che è **una sequenza**)
  - Il suo contenuto non è ordinato, ma è **ordinabile**...

```
d = {123432: "Mickey Mouse", 112232: "Daisy Duck"}  
for val in sorted(d.values()) :  
    print(val)
```



```
Daisy Duck  
Mickey Mouse
```

# Scandire un dizionario

- ❑ C'è un altro contenitore che può essere richiesto a un dizionario
  - Il contenitore delle coppie!
    - Viene restituito dal metodo **items()** ed è di tipo **dict\_items**, ma ci basta sapere che è **una sequenza**
    - I singoli elementi di tale contenitore (cioè le coppie) sono **tuple di dimensione 2** dove il primo elemento è la chiave
    - Il suo contenuto non è ordinato, ma è **ordinabile**...  
come si ordina una lista di tuple?
    - Viene usato il "confronto tra tuple", con l'operatore **<** che, **ricordiamo**, restituisce **True** se il primo elemento della tupla di sinistra è minore del primo elemento della tupla di destra
      - in caso di **primi elementi uguali** (che, **in questo caso, non può succedere** perché le chiavi sono tutte diverse), si guarderebbero i secondi elementi delle tuple

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}

# righe in ordine di matricola crescente
for item in sorted(d.items()) :
    # in ogni tupla presente nella sequenza
    # restituita da items(),
    # il primo elemento è la chiave
    print(item[0], item[1])

# un altro modo per fare la stessa cosa...
# ricordiamo l'assegnamento di una tupla a
# una tupla di variabili...
# (var1, var2) = tupla restituita da funzione...
# in questo caso la tupla di destra è, via via,
# una di quelle appartenenti alla sequenza
# restituita da items()
for (key, val) in sorted(d.items()) :
    print(key, val)
# oppure (come già visto...)
for key in sorted(d) :
    print(key, d[key])
```

# Una lista è un dizionario speciale...

- ❑ Che tipo di dizionario otteniamo se **usiamo come chiavi numeri interi consecutivi crescenti a partire da zero**?
  - Solitamente numeri di questo tipo vengono chiamati *indici*
- ❑ Le sue coppie sono del tipo (*numero*, *valore*), con *numero* che sarebbe un indice valido in una lista avente la stessa dimensione del dizionario
- ❑ In effetti **una lista mette in relazione valori con indici**:  
dal punto di vista del contenuto informativo, una lista, come un dizionario, memorizza coppie di tipo (*indice*, *valore*), con indici univoci (cioè tutti diversi)
  - Anche se **tecnicamente** una lista è ben diversa da un dizionario
- ❑ In altri linguaggi i dizionari vengono chiamati "**array associativi**", perché in pratica estendono il concetto di array (un sinonimo di lista) in modo che usi dati qualsiasi come indici, memorizzando *associazioni* tra chiavi (cioè "indici generalizzati") e valori

# Un insieme è un dizionario speciale...

- ❑ Che tipo di dizionario otteniamo se **usiamo None come valore in tutte le coppie?**
  - Il contenuto informativo di una coppia equivale a quello della sua chiave: i valori sono inutili
- ❑ Il contenuto informativo di un tale dizionario equivale a quello di un insieme contenente, come elementi, le chiavi delle coppie del dizionario
  - Inoltre, i vincoli a cui devono sottostare le chiavi di un dizionario (cioè il fatto di essere univoche e immutabili) coincidono con i vincoli a cui devono sottostare gli elementi di un insieme
  - Anche se **tecnicamente** un insieme è diverso da un dizionario, quindi non facciamo confusione tra i due tipi di dati

# Alcune precisazioni...

- ❑ Non esiste un **metodo** per inserire una nuova coppia in un dizionario, si usa la sintassi (già vista)  
**d[chiave] = valore**
- ❑ Gli oggetti restituiti dai metodi **keys**, **values** e **items** non sono delle vere e proprie liste... ma possono essere "trasformati" in liste passandoli alla funzione **list(...)** oppure in altri contenitori (tuple, insiemi, ...) passandoli alle opportune funzioni
  - Senza "trasformazione", con tali oggetti si può soltanto
    - Conoscerne la lunghezza, con la funzione **len**
    - Farne una scansione, usandoli in un ciclo **for**
    - Verificare la presenza/assenza di un elemento, con gli operatori **in** e **not in**
    - Ottenere la corrispondente lista ordinata, usando **sorted**

# Alcune precisazioni...

- ❑ Vediamo esplicitamente due rappresentazioni alternative delle informazioni di un dizionario usando soltanto liste

```
diz = {  
    "Mickey Mouse": [20, 30, 18],  
    "Daisy Duck": [30, 30, 28, 24],  
    "Goofy": [18, 20, 21]  
}
```

- ❑ **Una lista di coppie** (cioè di liste di dimensione due), ciascuna contenente una chiave e un valore

```
lst = [  
    ["Mickey Mouse", [20, 30, 18]],  
    ["Daisy Duck", [30, 30, 28, 24]],  
    ["Goofy", [18, 20, 21]]  
]
```

# Alcune precisazioni...

- ❑ Vediamo esplicitamente due rappresentazioni alternative delle informazioni di un dizionario usando soltanto liste

```
diz = {  
    "Mickey Mouse": [20, 30, 18],  
    "Daisy Duck": [30, 30, 28, 24],  
    "Goofy": [18, 20, 21]  
}
```

- ❑ **Due "liste parallele"**, una con le chiavi e una con i valori, in posizioni corrispondenti

```
keylist = ["Mickey Mouse", "Daisy Duck",  
           "Goofy"]  
valuelist = [[20, 30, 18],  
             [30, 30, 28, 24],  
             [18, 20, 21]  
            ]
```



# Tracciamento dell'esecuzione di un programma

# Debugging: tracciamento dell'esecuzione

- ❑ Il collaudo di un programma consente, a volte, di scoprire che è presente un errore (**rilevazione d'errore**): questo accade tutte le volte in cui il programma non fa quello che dovrebbe fare
  - Ricordare sempre:  
**se il collaudo va a buon fine, non è detto che non ci siano errori!**
- ❑ Dopo aver rilevato un errore, la prima attività da compiere è la **DIAGNOSI** dell'errore stesso
  - Innanzitutto bisogna capire/scoprire **DOVE** si trova l'errore
  - Successivamente bisogna capire **QUALE** è l'errore
  - Infine, possiamo sperare di **CORREGGERE** l'errore: se non sappiamo dov'è e qual è l'errore, non possiamo certamente correggerlo
- ❑ Per la prima fase, che possiamo chiamare **localizzazione** dell'errore (la prima parte della diagnosi) è molto utile (se non indispensabile) utilizzare il **tracciamento dell'esecuzione**

# Debugging: tracciamento dell'esecuzione

- ❑ Come sappiamo, il tracciamento dell'esecuzione di un programma richiede l'inserimento di enunciati **print(...)** e l'osservazione di quanto viene effettivamente visualizzato da tali enunciati durante l'esecuzione
- ❑ Se il collaudo evidenzia la presenza di un ciclo infinito nell'esecuzione di questo programma, all'interno di quale funzione sarà l'errore?
- ❑ Eseguo di nuovo il programma (con gli stessi dati di input) e osservo l'output
- ❑ Esempio

```
func1 (...)  
func2 (...)  
func3 (...)
```

```
func1 (...)  
print("func1 eseguita")  
func2 (...)  
print("func2 eseguita")  
func3 (...)  
print("func3 eseguita")
```

```
func1 eseguita  
func2 eseguita
```



Deduco che il ciclo infinito è all'interno della funzione **func3**

# Debugging: tracciamento dell'esecuzione

- ❑ Dopo aver identificato la funzione che contiene l'errore, si inseriscono messaggi di tracciamento al suo interno, per isolare ancora meglio l'errore

```
def func3(...) :  
    print(...) # i parametri ricevuti  
    ... # prima parte  
    print("func3: prima parte eseguita")  
    ... # seconda parte  
    print("func3: seconda parte eseguita")  
    toReturn = ... # calcolo valore da restituire  
    print("func3 restituisce", toReturn)  
    return toReturn
```

- ❑ Eseguiamo di nuovo e immaginiamo di scoprire che il ciclo infinito sia nella prima parte della funzione **func3**: dobbiamo tracciare l'esecuzione del ciclo

# Debugging: tracciamento dell'esecuzione

- ❑ All'interno del corpo del ciclo bisogna visualizzare i valori di tutte le variabili coinvolte nel ciclo stesso (sia quelle che secondo noi cambiano valore tra un'iterazione e l'altra, sia quelle che, magari sbagliando, immaginiamo che siano costanti)

```
def func3(...) :  
    ...  
    i = 0  
    while condizione con i :  
        print("i=", i, ...)   
        ...  
        if ... :  
            i += 1  
    ... # seconda parte
```

## Esempio

```
i= 0  
i= 1  
i= 2  
i= 2  
i= 2  
i= 2  
i= 2  
all'infinito
```

- ❑ Scopriamo che, se la condizione dell'**if** è falsa, la variabile **i** non viene incrementata: proprio questo è l'errore
- ❑ Qual è la correzione da fare? DIPENDE DALL'ALGORITMO

# Debugging: tracciamento dell'esecuzione

- ❑ Terminata la diagnosi e corretto l'errore, è bene procedere in questo modo
  - Eseguire nuovamente il programma, osservando ancora l'output del tracciamento, per verificare che il problema non sia più presente
  - Eseguire nuovamente il collaudo del programma, verificando che ora il collaudo venga superato
    - La soluzione del problema potrà, però, evidenziare nuovi problemi...
  - Eliminare i messaggi di tracciamento dal programma
    - È molto più efficiente **commentare** tali enunciati **print**: potranno tornare utili in seguito o dopo una fase di modifica delle funzionalità del programma, se il collaudo dovesse fallire di nuovo
- ❑ Può essere utile una strategia di "programmazione pessimista"
  - **Mentre scrivo il codice di un ciclo, inserisco già gli enunciati di tracciamento, commentandoli**
  - **Capita spesso che, ragionando per decidere quali informazioni di tracciamento sia utile visualizzare, si scopra già un errore...**  
così da poterlo correggere ancor prima di iniziare il collaudo!

**Lezione 30**  
**10/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Esempi di utilizzo di insiemi e dizionari



# Dati strutturati: *record* di dati

- ❑ Spesso capita di elaborare entità aventi molteplici attributi
  - Es. uno studente ha un numero di matricola, un nome, un cognome, una lista di voti degli esami superati...
  - Un tipo di informazione che nell'ambito della "gestione delle basi di dati" (*database management*) si chiama *record* ("voce in un archivio")
- ❑ Può essere comodo memorizzare tutte queste caratteristiche all'interno di **un unico oggetto** che rappresenti lo studente
  - Ad esempio, **una funzione che deve "elaborare uno studente"** potrà **ricevere un unico parametro** (da cui estrarre i singoli attributi) anziché molteplici parametri che diano un valore ai singoli attributi
  - Cambiando il numero di attributi, la firma della funzione non cambierà e riceverà sempre un unico oggetto: **molto comodo per non dover modificare TUTTE le invocazioni della funzione!**
- ❑ Il dizionario è perfetto!
  - Dobbiamo solo **dare un "nome" a ciascun attributo**: sarà la chiave di una coppia che descrive l'attributo

```
stud = {"id": 112233,
        "name": "Mickey",
        "surname": "Mouse",
        "grades": [18, 20, 19]}
}
```

- ❑ Anche in questo caso **potremmo usare una lista** per memorizzare il *record* di uno studente, ma è molto scomodo: **dovremmo ricordarci la posizione di ciascun singolo attributo**

```
stud = [112233, "Mickey", "Mouse", [18, 20, 19]]  
# visualizza il nome  
print(stud[1]) # il nome è in posizione 1
```

```
stud = {"id": 112233, "name": "Mickey",  
        "surname": "Mouse", "grades": [18, 20, 19]}  
# visualizza il nome  
print(stud["name"]) # molto più intuitivo
```

- ❑ Se, poi, decidessimo di eliminare un attributo, es. la matricola, bisognerebbe modificare tutto il codice che usa il record!

```
stud = ["Mickey", "Mouse ", [18, 20, 19]]  
# visualizza il nome  
print(stud[0]) # 1 diventa 0
```

```
stud = {"name": "Mickey", # ho tolto id  
        "surname": "Mouse", "grades": [18, 20, 19]}  
# visualizza il nome  
print(stud["name"]) # qui non cambia niente!
```

# Intestazione di un file CSV

- ❑ Spesso i file CSV (con campi separati da "punto e virgola") hanno una prima riga di "intestazione" (*header*), dove vengono dati

**nomi alle colonne:**

possiamo sfruttarli come chiavi in un dizionario

- ❑ Creiamo **una lista di dizionari**, uno per ciascuno studente

```
ID;Name;Surname;Enrollment
```

```
1122543;Mickey;Mouse;Law
```

```
1235436;Donald;Duck;Maths
```

```
1324321;Daisy;Duck;Surgery
```

```
1122110;Scrooge;McDuck;Economics
```

```
f = open("csv.txt")
lines = f.readlines() # leggo tutto il file
f.close()
keys = lines[0][:-1].split(";") # chiavi
studs = list() # sarà lista di dizionari
for r in range(1, len(lines)): # altre righe...
    stud = dict() # dizionario di uno studente
    fields = lines[r].strip("\n").split(";")
    for k in range(len(keys)) : # ogni chiave
        stud[keys[k]] = fields[k] # colonna k
    studs.append(stud) # lista di dizionari...
```

# Lettura di un file CSV

- ❑ Ora, ad esempio, possiamo cercare gli studenti iscritti a **giurisprudenza**

```
ID;Name;Surname;Enrollment
1122543;Mickey;Mouse;Law
1235436;Donald;Duck;Maths
1324321;Daisy;Duck;Surgery
1122110;Scrooge;McDuck;Economics
```

```
... # il codice precedente che
    # riempie studs, una lista di dizionari
for x in studs : # x è un dizionario
    if x["Enrollment"] == "Law" :
        print(x["ID"], x["Name"], x["Surname"])
```

Avrei potuto usare una lista di liste, ma sarebbe più scomodo (posizioni anziché chiavi...) e meno efficiente



```
1122543 Mickey Mouse
```

# Scrittura di un file CSV

- ❑ Dopo aver eventualmente modificato i dati all'interno del programma, posso riscriverli in formato CSV

```
... # il codice precedente seguito da eventuali
    # modifiche ai dati (es. elimino alcuni
    # studenti, cioè i loro dizionari da studs)
w = open("new.csv", "w") # nuovo file
# scrivo l'intestazione (prima riga)
w.write(";".join(keys) + "\n")
for x in studs : # x è un dizionario
    line = "" # creo la prossima riga
    for key in keys : # così sono in ordine
                        # rispetto alle colonne
        line += x[key] + ";"
    line = line[:-1] # tolgo ";" finale
    # senza fare un ciclo, avrei potuto scrivere
    # line = ";".join(list(x.values())) ???
    # no, perché... Com'è la lista x.values()?
    w.write(line + "\n")
w.close()
```

# Dizionario di dizionari...

- Quando le righe contengono (almeno) **un identificatore univoco**, cosa che spesso accade, si può usare una memorizzazione diversa

- **Un dizionario di dizionari**: il dizionario usa ID come chiave, mentre il valore sarà

| ID      | Name    | Surname | Enrollment |
|---------|---------|---------|------------|
| 1122543 | Mickey  | Mouse   | Law        |
| 1235436 | Donald  | Duck    | Maths      |
| 1324321 | Daisy   | Duck    | Surgery    |
| 1122110 | Scrooge | McDuck  | Economics  |

un dizionario contenente tutti i dati di quello studente

```
f = open("csv.txt")
lines = f.readlines()
f.close()
keys = lines[0][:-1].split(";")
studs = dict() # sarà dizionario di dizionari
for r in range(1, len(lines)): #altre righe...
    stud = dict() # dizionario di uno studente
    fields = lines[r][:-1].split(";")
    for k in range(len(keys)) :
        stud[keys[k]] = fields[k] #come prima...
    studs[stud["ID"]] = stud
# confrontare con l'idea precedente
```

# Dizionario di dizionari...

- ❑ Un dizionario di dizionari:
  - il dizionario "principale" usa ID come chiave, mentre ciascun valore sarà un dizionario contenente tutti i dati dello studente avente tale ID
  - Nel dizionario di ciascuno studente si può scegliere se ripetere anche ID (come nell'esempio precedente) oppure no
    - Può essere comodo ripeterlo per avere **tutti** i dati all'interno del singolo dizionario dello studente
- ❑ Con una struttura di memorizzazione come questa, è molto facile, ad esempio, scrivere un file CSV contenente le righe in ordine di numero di matricola

# Scrivere studenti in ordine di matricola

- Avendo i dati in un dizionario di dizionari, scrivo un file CSV con le **righe ordinate in base alla matricola**

```
... # il codice della slide precedente
    # seguito da eventuali modifiche ai dati
w = open("newSorted.csv", "w")
# scrivo l'intestazione (come prima)
w.write(";".join(keys) + "\n")
# studs è diz di dizionari, IDs come chiavi
for key in sorted(studs.keys()) : # keys()...
    # key è matricola in ordine crescente
    d = studs[key] # dizionario di uno studente
    # poi, tutto come prima...
    line = ""
    for k in keys :
        line += d[k] + ";" # ; finale da togliere
    w.write(line[:-1] + "\n")
w.close()
```



# Ordinamento "al contrario"

- ❑ Se volessimo ordinare le righe in base al numero di matricola DECRESCENTE, come potremmo fare?
- ❑ Possiamo semplicemente **invertire il contenuto della lista di chiavi ordinate**

```
... # il codice della slide precedente
    # seguito da eventuali modifiche ai dati
w = open("newSorted.csv", "w")
w.write(";".join(keys) + "\n")
lst = sorted(studs.keys()) # ID crescente
for i in range(len(lst)//2) :
    # ciclo analizzato nella prossima slide
    temp = lst[i]
    lst[i] = lst[-i-1]
    lst[-i-1] = temp
for key in lst : # lst è stata invertita
    d = studs[key] # dizionario di uno studente
    ... # come prima per generare una riga da d
w.close()
```

# Inversione di una lista

- ❑ Ricordiamo l'algoritmo generale per invertire il contenuto di una lista (**lst**)

```
for i in range(len(lst)//2) :  
    temp = lst[i]  
    lst[i] = lst[-i-1]  
    lst[-i-1] = temp
```

- ❑ Verifichiamo che funzioni correttamente con
  - Lista vuota (nessuna iterazione: ok)
  - Lista di lunghezza unitaria (nessuna iterazione: ok)
  - Lista di lunghezza 2: **len(lst)//2** vale 1, quindi fa una sola iterazione, con **i = 0**, scambiando **lst[0]** con **lst[-1]**, ok
  - Lista di lunghezza 3: **len(lst)//2** vale 1, quindi fa una sola iterazione, con **i = 0**, scambiando **lst[0]** con **lst[-1]**, ok
  - Lista di lunghezza 4: **len(lst)//2** vale 2, quindi fa due iterazioni, con **i = 0** e **i = 1**, scambiando **lst[0]** con **lst[-1]**, poi **lst[1]** con **lst[-2]**, ok
  - Eccetera...
- ❑ Lo stesso risultato si può ottenere con **lst.reverse()** 😊

# Ordinamento "al contrario"

- ❑ Oltre al metodo **reverse** per liste, c'è anche un'altra soluzione, che funziona anche per altri contenitori
  - La funzione **sorted** (come anche il metodo **sort** delle liste) accetta un "parametro con nome"  
[stessa sintassi vista in **print(..., end="")**]

```
... # il codice della slide precedente
    # seguito da eventuali modifiche ai dati
w = open("newSorted.csv", "w")
# scrivo l'intestazione (prima riga)
w.write(";".join(keys) + "\n")
for key in sorted(studs.keys(), reverse=True) :
    d = studs[key] # dizionario di uno studente
    # qui solito codice per generare e scrivere
    # una riga a partire dal dizionario d
    ...
w.close()
```

# Ordinamento "per cognome"

- ❑ Riusciamo a visualizzare le righe (o a scriverle in un file CSV) ordinate in base al cognome degli studenti?
- ❑ Potrei pensare di memorizzare i record (che sono dizionari) in **un dizionario principale che utilizzi il cognome come chiave** (invece del numero di matricola), dopodiché eseguo (come prima) un ciclo che scandisca le chiavi (cioè i cognomi) in ordine e recupero i record (cioè i dizionari) nell'ordine desiderato
  - Non è, però, una soluzione generale, perché **ci possono essere studenti che hanno lo stesso cognome**, quindi il cognome non è accettabile come chiave, **che deve essere univoca**
- ❑ Potrei usare come chiave **una tupla (cognome, nome)**: come verrebbero ordinate queste chiavi?
  - Le tuple si ordinano sulla base del confronto del loro primo elemento e, in caso di parità, si ordinano sulla base del secondo elemento: perfetto!
  - Ma, di nuovo, queste tuple non garantiscono l'univocità: possono esserci studenti omonimi, con lo stesso nome e cognome

# Ordinamento "per cognome"

- ❑ Riusciamo a visualizzare le righe (o a scriverle in un file CSV) ordinate in base al cognome degli studenti?
- ❑ Uso come chiave **una tupla (cognome, nome, matricola)**, in questo modo l'ordinamento delle chiavi segue il criterio che desidero e le chiavi sono univoche (perché lo sono i numeri di matricola)

```
...
newStuds = dict() #sarà dizionario di dizionari
# i "valori" delle coppie in studs sono
# i dizionari dei singoli studenti (i record)
for d in studs.values() : # d è un dizionario
    # genero la tupla che farà da chiave per d
    t = (d["Surname"], d["Name"], d["ID"])
    newStuds[t] = d
w = open("newSorted.csv", "w")
w.write(";".join(keys) + "\n")
for key in sorted(newStuds) : # le tuple...
    d = newStuds[key] # con cognome crescente
    # come prima, genero e scrivo la riga di d
    ...
w.close()
```

# Ordinamento "per cognome"

- ❑ Invece di una tupla (cognome, nome, matricola), potremmo usare una **stringa** ottenuta concatenando gli stessi elementi

```
... # il codice della slide precedente
    # seguito da eventuali modifiche ai dati
newStuds = dict()
for d in studs.values() : # per ogni record
    # genero la stringa che farà da chiave per d
    t = d["Surname"] + d["Name"] + str(d["ID"])
    newStuds[t] = d
# poi non cambia niente...
```

- ❑ Si tratta di una soluzione meno "generale", può fallire in casi particolari, ad esempio quando la parte iniziale del nome di uno studente è uguale alla parte finale del cognome di un altro...
- ❑ Ovviamente potremmo usare dei separatori nella concatenazione, ma, in generale, la tupla presenta meno problemi ed è comunque un oggetto non modificabile (condizione essenziale per svolgere il ruolo di chiave in un dizionario)

# Conteggio di occorrenze

- ❑ Problema: data una lista di parole, generare una lista delle parole uniche, **contando anche quante volte ciascuna parola ricorre nella lista originale**
  - Vogliamo generare, quindi, un contenitore di coppie: ciascuna coppia contiene una parola e il suo conteggio, e le parole sono tutte diverse
- ❑ Ovviamente potremmo
  - Generare l'insieme delle parole uniche (problema già visto come applicazione degli insiemi)
  - Per ogni parola unica, cercarne le occorrenze nella lista originaria (con il metodo **count**), memorizzando il risultato in una lista di tuple costruite come (*parola*, *num\_di\_occorrenze*)
- ❑ **Oppure usiamo un dizionario!**
  - Usiamo un dizionario dove le parole sono le chiavi e i relativi conteggi sono i valori
    - gestirà "automaticamente" l'unicità delle parole

# Conteggio di occorrenze

```
ws = [...] # lista di parole
s = set() # creo un insieme di parole
for word in ws :
    s.add(word) # trovo le parole uniche
counts = list() # lista di (parola, conteggio)
for word in s : # conto le occorrenze
    counts.append((word, ws.count(word))) #tuple
```

```
ws = [...] # lista di parole
d = dict() # insieme di parole uniche non serve
           # perché saranno le chiavi in d
for word in ws :
    d[word] = 1 + d.get(word, 0) # analizzare...
# finito 😊
```

- ❑ Divina Commedia: circa 600 mila caratteri,  
circa 105 mila parole, circa 14 mila parole uniche
- ❑ Con il dizionario: **0.028 s**
- ❑ Con la lista di tuple: **14.5 s (circa 500 volte più lento)**



# Conteggio di occorrenze

**Stesso  
codice,  
diverso  
libro  
elaborato**

```
ws = [...] # lista di parole
s = set()
for word in ws :
    s.add(word)
counts = list()
for word in s :
    counts.append( (word, ws.count(word)) )
```

```
ws = [...] # lista di parole
d = dict()
for word in ws :
    d[word] = 1 + d.get(word, 0)
# finito 😊
```

- ❑ La Bibbia (versione inglese di King James): circa 4.5 milioni di caratteri, circa 800 mila parole, circa 14 mila parole uniche (stranamente come nella Divina Commedia 😊)
- ❑ Con il dizionario: **0.173 s**
- ❑ Con la lista di tuple: **106 s (circa 600 volte più lento)**

# Parole più frequenti

```
ws = [...] # lista di parole
d = dict()
for word in ws :
    d[word] = 1 + d.get(word, 0)
```

- ❑ Dopo aver ottenuto il numero di occorrenze di ciascuna parola di un testo (come coppie parola/conteggio), possiamo, ad esempio, visualizzare le dieci parole più frequenti (e le relative occorrenze)
- ❑ **In pratica, si tratta di ordinare le coppie di un dizionario sulla base dei valori**, anziché ordinare le chiavi, come farebbe la funzione **sorted()**
  - Ordinando i valori, però, questi devono rimanere associati alle proprie chiavi (altrimenti perdo le informazioni che mi servono...) quindi devo ordinare delle coppie chiave/valore in base al valore, **non posso semplicemente ordinare i valori**
  - Se voglio sfruttare **sorted()** per ordinare un contenitore di coppie, devo generare delle **coppie valore/chiave**, perché **sorted()** ordina le tuple agendo prima sul primo componente

# Parole più frequenti

```
ws = [...] # lista di parole
d = dict()
for word in ws :
    d[word] = 1 + d.get(word, 0)
```

- ❑ Se voglio sfruttare **sorted()** per ordinare un contenitore di coppie, devo generare delle **coppie valore/chiave**, perché **sorted()** ordina le tuple agendo prima sul primo componente
- ❑ Ho un dizionario di coppie **chiave/valore**, posso generare **un dizionario** di coppie **valore/chiave**, avente lo stesso contenuto?
  - **In generale no, perché i valori non sono univoci!**
- ❑ Devo "trasferire" il dizionario in **una lista** di tuple valore/chiave, poi ordinerò tale lista con la funzione **sorted()** o con il metodo **sort()**

# Parole più frequenti

```
ws = [...] # lista di parole
d = dict()
for word in ws :
    d[word] = 1 + d.get(word, 0)
lst = list()
for (key, val) in d.items() :
    lst.append((val, key)) #tuple
lst.sort()
# la lista è ordinata per valori
# (cioè conteggi) crescenti
# visualizzo le ultime 10 coppie
for i in range(-1, -11, -1) :
    print(lst[i][1], lst[i][0])
```

Sappiamo che ci sono circa 800 mila parole, il 7.7% sono **"the"**

Dovremmo anche decidere cosa fare con **le maiuscole...**

La Bibbia  
(versione di  
King James)

**the** 62265  
**and** 38915  
of 34588  
to 13474  
**And** 12846  
that 12589  
in 12387  
shall 9762  
he 9668  
unto 8942

# Parole più frequenti

```
ws = [...] # lista di parole
d = dict()
for word in ws :
    # ignoro le maiuscole
    word = word.lower()
    d[word] = 1 + d.get(word, 0)
# da qui tutto come prima
lst = list()
for (key, val) in d.items() :
    lst.append((val, key))
lst.sort()
for i in range(-1, -11, -1) :
    print(lst[i][1], lst[i][0])
```

La Bibbia  
(versione di  
King James)

the 64204  
and 51761  
of 34791  
to 13660  
that 12927  
in 12725  
he 10422  
shall 9840  
for 8998  
unto 8997

# Ancora analisi di testi...

- ❑ Problema: dato un testo, visualizzare in ordine alfabetico l'elenco delle parole uniche utilizzate e, per ogni parola, indicare i numeri di riga in cui compare (se compare più volte su una stessa riga, quel numero di riga va indicato una volta sola)
  - È una specie di "indice analitico", solo che invece dei numeri di pagina uso i numeri di riga
- ❑ Innanzitutto, per memorizzare un elenco di parole uniche possiamo utilizzare **un dizionario o un insieme**
  - Dato che a ciascuna parola dobbiamo associare delle informazioni, è più naturale utilizzare **un dizionario, con le parole come chiavi**
  - Cosa saranno i valori associati alle parole?
    - Devono essere elenchi di numeri di riga **univoci**, possiamo usare un insieme
  - Usiamo **un dizionario di insiemi**, con stringhe come chiavi
  - Alla fine, per la visualizzazione, faremo una scansione del dizionario per chiavi ordinate (ordinando anche gli insiemi)

# Un dizionario di insiemi

- ❑ Problema: dato un testo, visualizzare in ordine alfabetico l'elenco delle parole uniche utilizzate e, per ogni parola, indicare i numeri di riga in cui compare (se compare più volte su una stessa riga, quel numero di riga va indicato una volta sola)

```
import re
text = ... # una stringa con più "righe"
lines = text.splitlines()
d = dict()
i = 1 # numero di riga
while i <= len(lines) :
    words = re.split("[^a-zA-Z]+", lines[i-1])
    for word in words :
        if word not in d : # parola nuova...
            d[word] = set() # insieme vuoto
            d[word].add(i) # "aggiungo" il numero
                           # all'insieme d[word]
    i += 1 # prossima riga
for word in sorted(d) :
    # d[word] è un insieme di numeri (di riga)
    print(word, sorted(d[word]))
```

Poche righe di codice risolvono  
un problema piuttosto complesso

Il materiale necessario per il  
Laboratorio 09  
termina qui



**Lezione 31**  
**11/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Progettazione di oggetti e di classi

# Progettazione di oggetti

- ❑ Abbiamo visto che può far comodo **memorizzare in un unico oggetto composto tutti gli attributi o proprietà di una entità** elaborata da un programma
  - Nei nostri esempi, i dati di ciascuno studente erano memorizzati in un dizionario (ma avremmo potuto usare una lista)
- ❑ In questo modo è molto comodo:
  - "passare uno studente" a una funzione che lo elabori
    - altrimenti dovrei definire un parametro per ogni attributo
  - progettare una funzione che "restituisca uno studente", ad esempio una funzione che legga una riga di un file CSV e restituisca il *record* corrispondente sottoforma di dizionario
    - impossibile senza oggetti composti...
  - simulare il comportamento della "segreteria studenti", che avrà bisogno di una raccolta di studenti, che diventa una lista di contenitori (o un dizionario di contenitori)
- ❑ **Abbiamo rappresentato un "dato composto" (detto *record*) mediante un oggetto (es. un dizionario o una lista)**

# Progettazione di oggetti

- ❑ Un *record* contiene tutti gli attributi di un'entità, ma è soltanto una struttura di memorizzazione, un componente software "passivo"
- ❑ Cosa significa renderlo "attivo"? Come può "agire" uno studente?
- ❑ Immaginiamo una rappresentazione semplificata di studente: un numero di matricola e una lista di voti

```
stud = { "ID": 123, "grades": [18, 24, 30] }
```

```
def addGradeToStud(x, g) :  
    x["grades"].append(g)
```

- ❑ Quando lo studente **stud** supera un esame e si deve aggiornare la sua carriera, la segreteria potrebbe invocare la funzione

```
addGradeToStud addGradeToStud(stud, newGrade)
```

- ❑ Ma quando lo studente diventerà un "**oggetto attivo**", potremo invocarne un **metodo**!

```
# ancora no...  
mystud.addGrade(newGrade)
```

- ❑ Vedremo alcuni vantaggi

# Progettazione di oggetti

□ Un primo grande vantaggio

- **Il partizionamento dello spazio dei nomi di funzioni**

□ Immaginiamo un programma che simuli il lavoro svolto dalla segreteria studenti. Avrà numerose funzioni:

- Funzioni per elaborare studenti (immatricola, iscriviti, modifica piano di studi, registra pagamenti, registra esame, cambia corso di studi, ecc.)
- Funzioni per elaborare insegnamenti (assegna anno/semestre nel manifesto, assegna crediti, assegna docente, assegna orario, assegna date appelli, ecc.)
- Funzioni per elaborare docenti...
- Funzioni per elaborare corsi di studio...
- Funzioni per elaborare...

□ **Tutte queste funzioni dovranno avere nomi diversi**, dai quali, per comodità, deve anche risultare **evidente la categoria di dati a cui si applicano** (e che restituiscono o vogliono ricevere come argomenti)

- Es. **addGradeToStud**, **displayStud**, **displayCourse**, **displayTeacher**

# Progettazione di oggetti

- ❑ Invece sappiamo che, **quando usiamo oggetti della libreria** di Python o di altri moduli (es. **ezgraphics**), **invochiamo metodi specifici per quegli oggetti**: non c'è bisogno che il nome del metodo faccia riferimento al tipo di oggetto con cui viene invocato, perché questo risulta evidente dal codice (es. metodo **strip**, non **stripFromString**; metodo **split**, non **splitString**)
- ❑ **Tali metodi si trovano in uno spazio di nomi distinto da quello delle funzioni**
- ❑ Non c'è bisogno che i nomi dei metodi facciano riferimento al tipo di dato su cui operano, perché possono essere invocati soltanto con oggetti di quel tipo
  - Il codice risulta decisamente più "snello"
- ❑ Il fatto che i metodi applicabili a un determinato tipo di oggetti siano effettivamente specifici per quella categoria lascia intendere che **dati e metodi** siano molto correlati tra loro e sarebbe meglio poterli **definire "insieme"**, anche dal punto di vista sintattico

# Progettazione di oggetti

- ❑ Tutte queste considerazioni (e **molte altre** che non vediamo) si traducono nella **definizione di una classe**, un'azione sintattica che **caratterizza** la **programmazione orientata agli oggetti** (**OOP**, *object-oriented programming*)
- ❑ Una classe costituisce **lo schema progettuale per la costruzione degli oggetti di un determinato tipo** e **definisce un nuovo tipo di dato**
- ❑ In tale schema progettuale ci sono
  - La descrizione degli **attributi di ogni oggetto di quel tipo**, cioè le **proprietà o caratteristiche** di tali oggetti
  - La descrizione della **procedura di "costruzione"** di un **oggetto** che sia un **esemplare (o istanza)** di tale classe, cioè che ne segua lo schema progettuale
  - Le definizioni dei **metodi** che sono invocabili con tali oggetti e che li elaborano in vario modo, determinandone l'evoluzione all'interno del programma che li usa

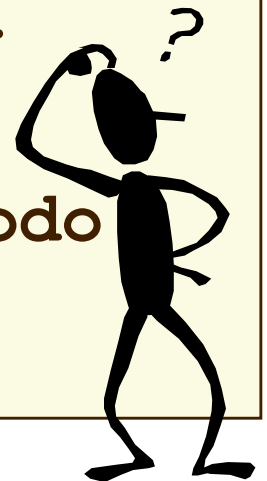
# Utilizzo di oggetti

- Come si useranno gli oggetti che **progetteremo**? Esattamente

come quelli che abbiamo usato finora: per memorizzare un record useremo un oggetto di tipo **Student** definito nel modulo **student** anziché un dizionario

| ID      | Name    | Surname | Enrollment |
|---------|---------|---------|------------|
| 1122543 | Mickey  | Mouse   | Law        |
| 1235436 | Donald  | Duck    | Maths      |
| 1324321 | Daisy   | Duck    | Surgery    |
| 1122110 | Scrooge | McDuck  | Economics  |

```
from student import Student
f = open("csv.txt")
lines = f.readlines(); f.close() # novità...
studs = dict() # dizionario di oggetti Student
for r in range(1, len(lines)) :
    fields = lines[r][: -1].split(";")
    stud = Student(fields[0], fields[1],
                   fields[2], fields[3])
    studs[fields[0]] = stud # matricola...
# registriamo esami di uno studente
x = studs["1235436"] # Paperino
x.addGrade(28) # Student avrà questo metodo
x.addGrade(30)
print(x.getAvgGrade()) # e questo...
```





# Definizione di una classe

- ❑ Il progetto di un nuovo tipo di oggetti si concretizza sintatticamente nella **definizione di una classe**
- ❑ Innanzitutto occorre decidere il nome di tale nuovo tipo di dato, nel nostro caso **Student**
  - Solitamente i nomi delle classi (cioè dei tipi di dati) sono sostantivi (singolari) con iniziale maiuscola, per convenzione stilistica
    - Per noi, una regola... che NON vale nella libreria standard!
  - Spesso la definizione di una classe viene scritta in un file sorgente che diventa **un modulo**, anche se solitamente contiene codice dedicato al collaudo della classe stessa (ed eseguibile condizionatamente usando `__name__`)
    - Scriveremo il modulo **student.py**
    - I programmi che useranno oggetti di tipo **Student** avranno

`from student import Student`

# Definizione di una classe

- ❑ Il progetto di un nuovo tipo di oggetti si concretizza sintatticamente nella **definizione di una classe**

```
# student.py (modulo)

## Uno studente ha un numero di matricola
# (univoco) e una lista di voti.
#
class Student :
    ... # codice che definisce le
        # caratteristiche e le funzionalità
        # degli oggetti di tipo Student
```

- ❑ **class** è una nuova parola riservata del linguaggio
- ❑ Per convenzione, i nomi delle classi definite **al di fuori** della libreria standard hanno l'iniziale maiuscola
- ❑ **Ricordiamo che nel nostro corso le "convenzioni stilistiche" sono REGOLE...**

# Definizione di una classe

- ❑ Nel progetto di una classe, la prima cosa da decidere è la sua *interfaccia pubblica*
  - L'insieme dei metodi invocabili con oggetti che siano esemplari di tale classe
- ❑ L'interfaccia pubblica di una classe definisce le funzionalità dei suoi esemplari
  - Viene anche chiamata API (*Application Programming Interface*)
- ❑ È costituita dai suoi metodi e, per ciascun metodo:
  - Nome
  - Elenco delle eventuali variabili parametro e del loro significato
  - Eventuali eccezioni sollevate, specificandone il nome e le condizioni che ne provocano il sollevamento
  - Eventuale valore restituito e suo significato
  - Descrizione della funzionalità svolta (non necessariamente il codice del metodo)

# Interfaccia pubblica

Osserviamo che i metodi sono definiti **all'interno** della classe (sono indentati verso destra)

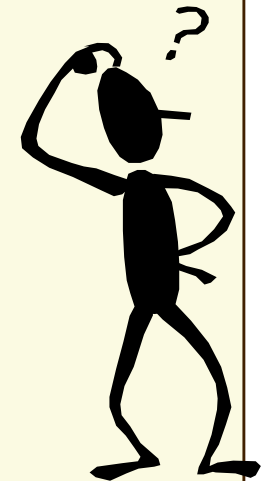
```
# student.py (modulo)

## Uno studente ha un numero di matricola
# (univoco) e una lista di voti.
#
```

Stessa sintassi usata per le funzioni

```
class Student :
    ## Aggiunge un voto allo studente.
    # @param grade il voto da aggiungere
    #
    def addGrade(self, grade) :
        ... # codice del metodo

    ## Ispeziona la media dei voti.
    # @return la media dei voti
    #
    def getAvgGrade(self) :
        ... # codice del metodo
```



❑ Cosa sarà il parametro **self**, oltretutto non documentato?

# Parametro `self`

- ❑ Tutti i metodi definiti all'interno di una classe **devono** avere, come **primo parametro**, la variabile **`self`**
  - È un **nome predefinito**, con un **significato speciale**
- ❑ **A cosa fa riferimento questa variabile?**
- ❑ L'elaborazione svolta da un metodo può coinvolgere
  - i dati forniti come argomenti nell'invocazione (all'interno delle parentesi tonde che seguono il nome)
  - **l'oggetto con cui si invoca il metodo**, cioè quello che figura alla sinistra del nome del metodo, separato da un punto
- ❑ Come fa il (codice del) metodo ad agire sull'oggetto con cui viene invocato?

**`stud.addGrade(24)`**

  - Tale oggetto, passato come gli altri parametri, viene memorizzato automaticamente dall'interprete nella variabile **`self`** (che è un *alias* per l'oggetto usato nell'invocazione) e viene anche detto "parametro implicito" (gli altri "espliciti")

# Parametro `self`

❑ **ATTENZIONE: purtroppo**, se dimentichiamo la variabile parametro **`self`** nella definizione di un metodo, **non viene segnalato un errore di sintassi**



- Si verifica un errore (solitamente **`TypeError`**) non appena proviamo a **invocare tale metodo** (con un oggetto che sia esemplare di quella classe)

❑ La variabile **`self`**, **all'interno di un metodo**, fa sempre (ovviamente?) riferimento a un oggetto che sia esemplare della classe in cui è definito il metodo stesso

- Questo perché un metodo della classe **`Student`** può essere invocato soltanto usando un oggetto di tipo **`Student`**

```
stud = Student(...)  
stud.addGrade(24)
```

# Definizione dello stato degli oggetti

- ❑ Dopo aver definito l'interfaccia pubblica di una classe (cioè le funzionalità dei suoi esemplari), mediante l'elenco dei suoi metodi, occorre dedicarsi alla definizione dello **stato** degli oggetti
  - Per meglio dire, delle **informazioni di stato**
- ❑ Cosa intendiamo per "stato di un oggetto" ?
  - È l'insieme delle **informazioni che ne riassumono il passato e ne determinano il futuro**, in funzione dei metodi che verranno invocati con quell'oggetto
- ❑ In pratica, dobbiamo chiederci
  - **Tra due invocazioni di metodi qualsiasi effettuate con uno stesso oggetto, cosa si deve ricordare tale oggetto per poter assolvere ai propri compiti futuri, sulla base del suo passato?**

# Definizione dello stato degli oggetti

- ❑ I nostri oggetti di tipo **Student** hanno un'interfaccia pubblica piuttosto limitata
  - **addGrade** e **getAvgGrade**
- ❑ Il metodo **addGrade** non ha "effetti visibili", perché non restituisce alcun valore, né visualizza alcunché, quindi potremmo anche immaginare che non faccia niente e che, quindi, non abbia bisogno di informazioni sul "passato" dell'oggetto, né che debba preservare informazioni per il futuro
- ❑ Il metodo **getAvgGrade**, invece, deve restituire il valore medio dei voti che sono stati aggiunti allo studente mediante invocazioni del metodo **addGrade**, quindi ha bisogno di informazioni sul passato
  - Ad esempio, ha bisogno di una lista contenente i voti aggiunti
  - Oppure della somma dei voti aggiunti e del loro numero
  - Oppure della media dei voti aggiunti
  - Oppure di una stringa contenente la concatenazione dei voti aggiunti, con separatori che consentano di distinguerli individualmente
  - Oppure... **tutte informazioni che vanno aggiornate da addGrade!**



# Definizione dello stato degli oggetti

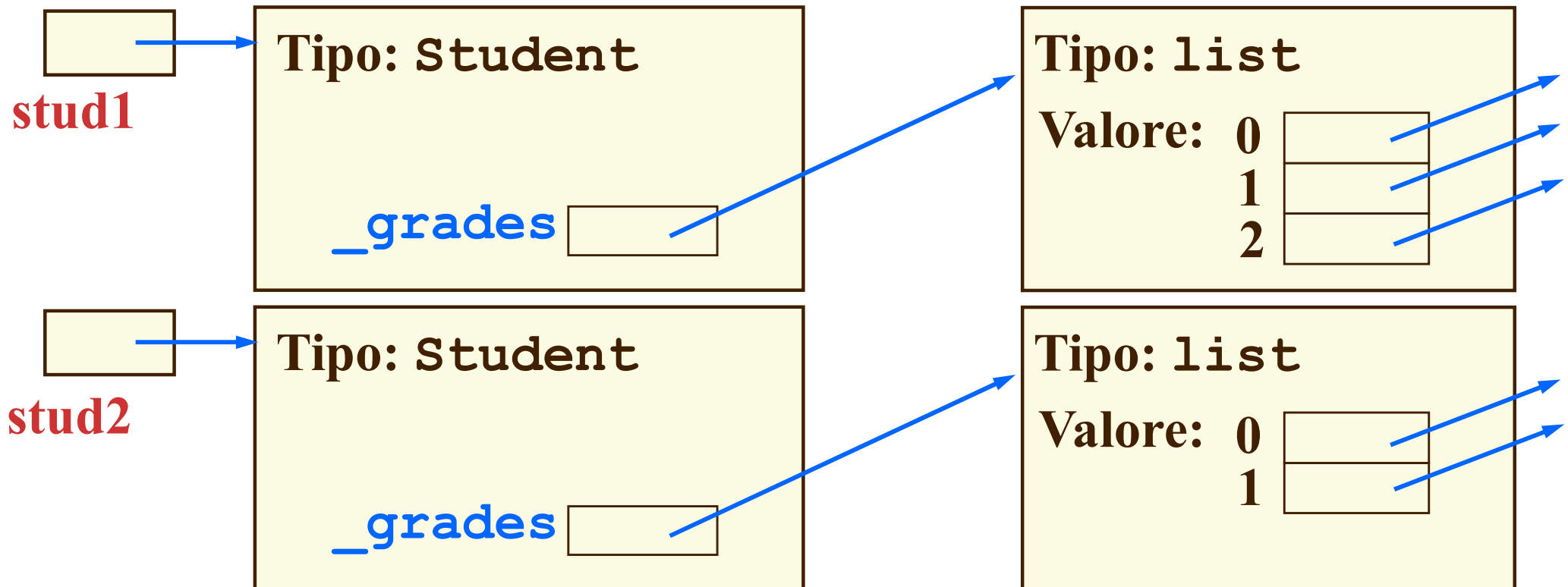
- ❑ Dopo aver individuato quali informazioni devono essere memorizzate all'interno di ciascun oggetto per consentirne il funzionamento (cioè il funzionamento dei metodi dell'interfaccia pubblica della classe di cui è esemplare), dobbiamo tradurre tali informazioni in codice Python
- ❑ Gli elementi sintattici che consentono la **memorizzazione dello stato di un oggetto al suo interno** fanno parte della definizione della classe di cui l'oggetto stesso è esemplare
  - Si chiamano **variabili di esemplare** (o di stato) e, per **convenzione che seguiremo, hanno nomi che iniziano con \_**
  - Se decidiamo, ad esempio, di rappresentare lo stato di uno studente con una lista che ne memorizzi i voti, **ci basta un'unica variabile di esemplare: una lista**
    - **Il numero di matricola non serve perché nell'interfaccia pubblica non ci sono metodi che ne hanno bisogno, quindi per il momento non lo inseriamo nello stato di uno studente**
    - Magari in una successiva evoluzione della classe...

# Definizione dello stato degli oggetti

- In generale, in un programma si potranno creare **più esemplari di una stessa classe** (la segreteria gestisce molti studenti...)

**Molto importante:** **ciascun esemplare di una classe ha le proprie informazioni di stato!**

- All'interno della zona di memoria dedicata a ciascun singolo oggetto trovano posto le **sue** variabili di esemplare [qui rappresentazione ALLEGORICA]
- Le variabili di esemplare di un oggetto non hanno **NESSUNA** relazione con le variabili di esemplare di un altro oggetto dello stesso tipo, anche se hanno gli stessi nomi



# Accesso alle variabili di esemplare

- ❑ Se un oggetto contiene una variabile di esemplare che si chiama **\_grades**, per accedervi si usa il **nome di una variabile che faccia riferimento all'oggetto**, seguita da un punto e dal nome della variabile di esemplare

- Sintassi **simile** all'invocazione di un metodo



```
stud = Student(...)  
v = len(stud._grades) # questo accesso è OK  
# ma vedremo che in realtà è sconsigliato
```

- Quindi, se invoco un metodo della classe **Student** (usando ovviamente un oggetto di tipo **Student**), nel codice di quel metodo, tramite la variabile **self**, potrò accedere alle variabili di esemplare dello specifico oggetto con cui il metodo è stato invocato
- Nel codice dei metodi **addGrade** e **getAvgGrade**, usando la sintassi **self.\_grades** accedo alla lista dei voti dello studente utilizzato **per quella specifica invocazione** del metodo

- ❑ Decisa l'interfaccia pubblica e determinate le informazioni di stato e la loro modalità di memorizzazione, possiamo procedere alla definizione del codice dei metodi della classe

```
# student.py (modulo)

## Uno studente ha (soltanto) una lista di voti.
# **** CLASSE ANCORA INCOMPLETA ****
class Student :
    ## Aggiunge un voto allo studente.
    # @param grade il voto da aggiungere
    #
    def addGrade(self, grade) :
        self._grades.append(grade)

    ## Ispeziona la media dei voti.
    # @return la media dei voti
    #
    def getAvgGrade(self) :
        if len(self._grades) == 0 : return 0
        return sum(self._grades)/len(self._grades)
```

**Lezione 32**  
**13/12/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Definizione dello stato degli oggetti

- ❑ A volte **non è semplice definire completamente l'elenco delle variabili di esemplare** di una classe
- ❑ **Si può procedere per "miglioramenti successivi"**
  - Individuo una parte delle informazioni di stato (magari penso di averle individuate tutte...)
    - Inutile sforzarsi troppo, ragionando in astratto...
  - Inizio a scrivere il codice dei metodi basandomi su tali variabili di esemplare
  - Mi accorgo che alcuni metodi non sono realizzabili: mi servono nuove variabili di esemplare
    - Dopo aver analizzato a fondo il codice e le caratteristiche degli esemplari della classe, **se effettivamente sono necessarie nuove variabili di esemplare**, le aggiungo
  - Proseguo nella realizzazione dei metodi: se scopro altre informazioni mancanti... torno ai punti precedenti!
  - Se, **al termine** della realizzazione dei metodi scopro che alcune delle variabili di esemplare che avevo inizialmente previsto NON sono state utilizzate nel codice dei metodi, dopo aver controllato a fondo, le cancello perché evidentemente sono inutili (occuperebbero inutilmente spazio in memoria in ciascun esemplare della classe)

# Stato iniziale di un oggetto

- ❑ Abbiamo quasi finito...
- ❑ Dobbiamo prendere un'ultima decisione importante (di solito abbastanza semplice)
  - Qual è lo **stato iniziale** di un esemplare della classe?
- ❑ Se l'evoluzione dello stato di un oggetto dipende dai metodi che si invocano, (è evidente che...) ciò dipenderà anche dallo stato iniziale dell'oggetto stesso
  - La media dei voti di uno studente è ben diversa se, dopo la **prima** esecuzione del metodo **stud.addGrade(18)**, la lista dei voti dello studente **stud** contiene soltanto quell'unico 18 oppure tre 30 seguiti da un 18 !
  - È ragionevole che, subito dopo essere stato creato, lo stato iniziale di un oggetto di tipo **Student** abbia una lista di voti vuota: **come esprimiamo questa fase del progetto?**

# Stato iniziale di un oggetto

- ❑ Lo **stato iniziale** degli esemplari di una classe viene definito mediante un metodo "speciale" detto **costruttore**
- ❑ Il costruttore, in una classe, è un metodo che si chiama SEMPRE **`__init__`** # doppio underscore prima e dopo
- ❑ Di norma, il suo compito è quello di assegnare un valore a **tutte** le variabili di esemplare dell'oggetto appena creato
- ❑ Viene invocato **automaticamente** dall'interprete dopo aver creato un oggetto: non va **mai** invocato esplicitamente (anche se non sarebbe un errore di sintassi)

```
class Student :
```

```
    def __init__(self) :  
        self._grades = list()  
  
    ... # il codice dei metodi
```

```
# crea un nuovo oggetto e  
# ne invoca il costruttore  
stud = Student()  
print(stud.getAvgGrade()) # 0
```

Vedremo che, come tutti i metodi, può ricevere parametri, **oltre a self**



# Stato iniziale di un oggetto

- ❑ L'invocazione e l'esecuzione del costruttore è conseguenza della creazione di un oggetto, che si effettua scrivendo il nome della classe di cui l'oggetto sarà esemplare, seguito da una coppia di parentesi tonde

```
# crea un nuovo oggetto e  
# ne invoca il costruttore  
stud = Student()
```

- ❑ A cosa fa riferimento il parametro **self** definito nel costruttore?

- ❑ In effetti, non invochiamo il costruttore usando un oggetto, come invece facciamo con i metodi "normali", quindi?

```
class Student :  
  
    def __init__(self) :  
        self._grades = list()
```

- ❑ Nel costruttore, il parametro **self** fa riferimento all'**oggetto che è in corso di costruzione e che verrà restituito dal costruttore!** È tutto gestito dall'interprete, che costruisce l'oggetto assegnandogli uno spazio in memoria e, poi, ne esegue immediatamente il costruttore.

# Stato iniziale di un oggetto

- ❑ Al termine dell'esecuzione del costruttore, durante il processo di creazione di un nuovo oggetto, esemplare di una classe, l'oggetto deve essere **"pronto a funzionare"**, cioè con tale oggetto deve essere possibile invocare QUALSIASI metodo della sua classe, ottenendo il funzionamento previsto
- ❑ Proprio per questo motivo, **il costruttore ha il compito di assegnare un valore iniziale a tutte le variabili di esemplare dell'oggetto**

```
# crea un nuovo oggetto e
# ne invoca il costruttore
stud = Student()
# ora l'oggetto stud è pronto a funzionare
print(stud.getAvgGrade()) # 0
stud.addGrade(18)
stud.addGrade(20)
print(stud.getAvgGrade()) # 19
```

# Classe completata

```
# student.py (modulo)

## Uno studente ha (soltanto) una lista di voti.
#
class Student :
    def __init__(self) :
        self._grades = list()

    ## Aggiunge un voto allo studente.
    # @param grade il voto da aggiungere
    #
    def addGrade(self, grade) :
        self._grades.append(grade)

    ## Ispeziona la media dei voti dello studente.
    # @return la media dei voti dello studente
    #
    def getAvgGrade(self) :
        if len(self._grades) == 0 : return 0
        return sum(self._grades)/len(self._grades)
```

# Progetto alternativo

- ❑ Usiamo **una diversa rappresentazione delle informazioni di stato**

```
class Student : # MANCANO I COMMENTI... che non cambiano
    def __init__(self) :
        self._sum = 0
        self._num = 0
    def addGrade(self, grade) :
        self._sum += grade
        self._num += 1
    def getAvgGrade(self) :
        if self._num == 0 : return 0
        return self._sum / self._num
```

- ❑ Chi utilizzava esemplari della versione precedente della classe non deve modificare **NULLA** nel proprio codice, perché **l'interfaccia pubblica non è cambiata!**
- ❑ Grande vantaggio della programmazione a oggetti
  - **Posso migliorare le prestazioni della classe senza obbligare gli utilizzatori a modificare il proprio codice**
  - **È quello che spesso succede con le nuove versioni del linguaggio Python:** le classi della libreria standard diventano "migliori" (sotto diversi punti di vista) senza che chi le utilizza debba modificare il proprio codice (se non, eventualmente, per utilizzare metodi che siano stati aggiunti)

# Costruttore

- ❑ Il nostro costruttore non accetta/richiede argomenti... ma come avevamo pensato di costruire un oggetto di tipo **Student**?

```
...  
for r in range(1, len(lines)) :  
    fields = lines[r][: -1].split(";")  
    stud = Student(fields[0], fields[1],  
                   fields[2], fields[3])  
...
```

- ❑ Con l'interfaccia pubblica che abbiamo progettato, i nostri studenti non hanno bisogno di informazioni aggiuntive, ma possiamo immaginare che il progetto si evolva: adeguiamo il costruttore

```
class Student :  
  
    def __init__(self, id, name, surname, enroll) :  
        self._grades = list()  
        self._id = id  
        self._name = name  
        self._surname = surname  
        self._enroll = enroll
```

Non è necessario che variabili di  
esemplare e parametri del  
costruttore abbiano gli stessi nomi  
(a parte l'underscore iniziale)  
ma spesso è così

# Costruttore

- ❑ Possiamo anche decidere di acquisire i valori in modo diverso...

```
class Student :  
  
    def __init__(self, csvline) :  
        self._grades = list()  
        fields = csvline[:-1].split(";") # \n finale...  
        self._id = fields[0]  
        self._name = fields[1]  
        self._surname = fields[2]  
        self._enroll = fields[3]
```

- ❑ E, di conseguenza, nel solito esempio:

```
...  
for r in range(1, len(lines)) :  
    stud = Student(lines[r])  
...
```

- ❑ Da questo progetto potremmo far evolvere il nostro studente...  
invece lo faremo ripartendo dalla versione che ha soltanto la lista  
dei voti

# Definizioni dei metodi

- ❑ Come abbiamo visto, **la definizione di un metodo è molto simile alla definizione di una funzione**
- ❑ **Riassunto delle differenze:**
  - I metodi sono **definiti all'interno di una classe**, nel suo *corpo*
  - I metodi devono avere almeno una variabile parametro e la prima variabile parametro deve chiamarsi **self**
- ❑ Poi, solitamente i metodi si distinguono (non sintatticamente, ma semanticamente) in
  - **Metodi di solo accesso:** consentono di ispezionare alcune proprietà degli oggetti con cui vengono invocati
  - **Metodi modificatori:** consentono di modificare lo stato degli oggetti con cui vengono invocati
  - Ci sono anche metodi con comportamento misto

# Errore MOLTO frequente

- ❑ Uno degli errori più frequenti, nella scrittura del codice di un metodo/costruttore, è il **mancato utilizzo di `self`** nell'accesso a una variabile di esemplare

```
class Student :  
    def __init__(self) :  
        _grades = list()  
    def addGrade(self, grade) :  
        self._grades.append(grade)  
    ...
```

- ❑ Questo, PURTROPPO, non è un errore di sintassi
  - Nel costruttore viene definita la **variabile locale `_grades`** (a cui viene assegnata una lista vuota) e **NON la variabile di esemplare `self._grades`**: perfettamente legittimo, anche se errato dal punto di vista logico
  - Il problema si manifesta (e viene segnalato) durante la prima esecuzione di **`addGrade`**, dove la variabile **`self._grades`** viene utilizzata anche se non esiste all'interno dell'oggetto  
**AttributeError: 'Student' object has no attribute '\_grades'**



# Costanti di classe

# Costanti di classe

- ❑ A volte sorge la necessità di **condividere informazioni tra tutti gli esemplari di una stessa classe**
- ❑ Un primo esempio è la condivisione di informazioni **COSTANTI**, che è inutile ripetere in ciascun oggetto

Come sappiamo,  
è bene non usare valori  
letterali nel codice,  
i cosiddetti  
**"numeri magici"**

```
class Student :  
    ...  
    def addGrade(self, grade) :  
        if grade < 18 :  
            raise ValueError  
        self._grades.append(grade)
```

```
class Student :  
    ...  
    PASSING_GRADE = 18 # nella classe ma fuori  
                        # da tutti i metodi  
    # nome maiuscolo perché è una costante...  
    def addGrade(self, grade) :  
        if grade < Student.PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)
```

# Costanti di classe

- ❑ Una variabile definita **all'interno di una classe** ma **al di fuori dei suoi metodi** si chiama **variabile di classe**
  - In altri linguaggi, "variabile statica"
  - Ne esiste **un'unica copia**, in una zona di memoria dedicata ALLA CLASSE, non ai suoi singoli esemplari
  - Ogni esemplare della classe contiene **concettualmente** un riferimento a tale zona di "memoria di classe", **condivisa**
    - anche se tecnicamente le cose sono un po' diverse...
- ❑ Dall'esterno della classe, si accede alle variabili di classe usando **il nome della classe** (seguito dal punto)

```
print(Student.PASSING_GRADE) # 18
```
- ❑ Dall'interno della classe... è uguale ☺

```
if grade < Student.PASSING_GRADE :
```
- ❑ Non si usa **self** ! Perché non è una variabile di esemplare

# Costanti di classe

```
class Student :  
    ...  
    PASSING_GRADE = 18 # costante di classe  
    def addGrade(self, grade) :  
        if grade < Student.PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)
```

- ❑ In alternativa, potrei usare una costante **locale**, all'interno del metodo **addGrade**, ma la costante di classe rende disponibile tale valore **in tutti i metodi** della classe (e anche all'esterno di essa), cosa che può essere utile!

```
class Student :  
    ...  
    def addGrade(self, grade) :  
        PASSING_GRADE = 18 # costante locale  
        if grade < PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)
```

**Collaudo di una classe**

# Collaudo di una classe

- ❑ È buona norma inserire il codice di collaudo della classe all'interno del modulo stesso, con la nota esecuzione condizionata

```
... # codice del modulo

if __name__ == "__main__" :

    # codice di collaudo del modulo:
    # crea esemplari della classe
    # e li "mette all'opera" in vari modi
    ...
```

- ❑ In questo modo si può collaudare il funzionamento degli esemplari della classe ancor prima di scrivere un vero programma che li utilizzi
- ❑ È una strategia che prende il nome di **collaudo di unità** (*unit testing*), nel senso che si collauda un'unità operativa (una classe) separatamente da tutto il resto

**Lezione 33**  
**17/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

**Metodi che invocano altri  
metodi della stessa classe**



# Metodi che invocano altri metodi

- ❑ Modifichiamo ancora l'interfaccia pubblica della classe **Student**
- ❑ Il metodo **addGrade**, dopo aver aggiunto il voto allo studente con cui è stato invocato, restituisce il suo voto medio (come farebbe **getAvgGrade**)
  - Valore, che, ovviamente, può essere ignorato dall'invocante

```
class Student :  
    ...  
    def addGrade(self, grade) :  
        if grade < Student.PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)  
        return sum(self._grades)/len(self._grades)
```

- ❑ Osserviamo che l'elaborazione svolta dalla parte finale del metodo **addGrade** coincide con quella svolta dal metodo **getAvgGrade**

# Metodi che invocano altri metodi

- ❑ In generale, sappiamo che **è bene evitare codice duplicato**, per agevolare la manutenzione e ridurre la possibilità di errori
  - Ad esempio, in futuro potremmo decidere che nel calcolo del voto medio vada ignorato il voto più basso e/o il voto più alto... dovremmo così **modificare il codice di entrambi i metodi**
  - L'ideale sarebbe che il metodo **addGrade**, dopo aver compiuto la sua elaborazione specifica (l'aggiunta del nuovo voto allo studente), potesse invocare il metodo **getAvgGrade** per lo stesso studente
  - In effetti può farlo... ma quale oggetto deve usare per invocare tale metodo?
  - Ovviamente deve usare l'oggetto (di tipo **Student**) che sta elaborando! Cioè **self**...

Situazione  
frequente

```
class Student :  
    ...  
    def addGrade(self, grade) :  
        if grade < Student.PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)  
        return self.getAvgGrade()
```

- Così non si duplica codice

Funzione `print( )`  
e nostri oggetti

# print( ) stampa qualsiasi cosa...

- ❑ Proviamo a invocare **print** passando un oggetto di tipo **Student** come argomento

```
stud = Student()  
print(stud)
```



```
<__main__.Student object at 0x000002199684C7F0>
```

- ❑ Non esaltante... ma ragionevole

- Come avrebbe potuto il progettista della funzione **str(...)**, che sappiamo essere invocata da **print(...)** sapere "come è fatto uno studente" ? Il tipo di dato **Student** non esisteva ancora...
- È responsabilità del progettista di una classe decidere come i suoi esemplari debbano essere visualizzati o "trasformati in stringa"

# print() stampa qualsiasi cosa...

## ❑ Come fa `print()` a sapere come visualizzare oggetti di tipi diversi??

- Ricordiamo che converte in stringhe tutti gli oggetti che riceve, passandoli uno ad uno alla funzione predefinita `str()`
- Abbiamo solo spostato il problema...
- Come fa `str()` a sapere come convertire in stringa oggetti di tipi diversi?
- È semplice... **delega** il compito al progettista di ciascun oggetto!  
Cioè usa metodi della classe di cui l'oggetto è esemplare...
  - In particolare, se nella classe è definito il metodo `__str__`, lo invoca e usa la stringa restituita da tale metodo
  - Altrimenti genera la stringa che abbiamo visto... che in pratica contiene informazioni (note all'interprete) sul tipo di oggetto e sul suo indirizzo in memoria (in esadecimale)

```
<__main__.Student object at 0x000002199684C7F0>
```

- ❑ Quindi, perché gli esemplari di una classe forniscano informazioni significative quando vengono passati alla funzione **print()** o **str()**, è necessario (e sufficiente) definire nella classe il metodo **\_\_str\_\_**, nel quale possiamo decidere quali informazioni visualizzare e con quale formato

```
class Student :  
    ...  
    def __str__(self) :  
        return str(self._grades) + \  
            " Avg. = " + str(self.getAvgGrade())
```

- ❑ La definizione del metodo **\_\_str\_\_** rende possibile ottenere informazioni significative da un oggetto di tipo **Student**
  - quando si passano esemplari di **Student** alla funzione **str**
  - di conseguenza, quando si passano esemplari di **Student** alla funzione **print**
- ❑ **MOLTO UTILE PER IL DEBUGGING...**

**Metodi di una classe che  
elaborano due esemplari della  
classe stessa**

# Metodi che elaborano due oggetti

- ❑ Capita spesso di voler scrivere **un metodo che elabori due oggetti dello stesso tipo**: quello con cui viene invocato (cioè il suo parametro "implicito", **self**) e un altro ricevuto come parametro esplicito ma del medesimo tipo

- ❑ Esempio: un oggetto è "minore" di un altro, dello stesso tipo?

```
stud1 = Student()  
stud2 = Student()  
if stud1.isLessThan(stud2) :  
    ...
```

- Ovviamente **bisogna definire una relazione d'ordine** per gli oggetti di quel tipo, altrimenti "minore" non ha senso
- Nel nostro esempio di "studente" estremamente semplificato, non abbiamo molte alternative
  - Lo studente minore è quello che ha il voto medio minore (o viceversa...)
- Usando studenti più "completi", potremmo, ad esempio, definire "minore" lo studente che ha il numero di matricola minore, oppure chi ha il cognome "minore" in senso lessicografico, oppure...



# Metodi che elaborano due oggetti

- ❑ Abbiamo deciso che lo studente minore è quello che ha il voto medio minore, quindi:

```
class Student :  
    ...  
    def isLessThan(self, other) :  
        return self.getAvgGrade() < other.getAvgGrade()
```

- ❑ Attenzione all'ordine tra i parametri... per rispettare il significato del metodo che stiamo progettando
- ❑ Vediamo (ancora) un metodo (**isLessThan**) che invoca un altro metodo della stessa classe (per non ripetere codice...): **getAvgGrade**
  - Anzi, lo invoca due volte, con due oggetti diversi...
  - Come abbiamo visto in precedenza, è perfettamente lecito

# Operatori di confronto tra oggetti

- ❑ In realtà ci vediamo "costretti" a scrivere così...

```
stud1 = Student()  
stud2 = Student()  
if stud1.isLessThan(stud2) :  
    ...
```

- ❑ ... ma preferiremmo poter scrivere così!

```
stud1 = Student()  
stud2 = Student()  
if stud1 < stud2 :  
    ...
```

- ❑ Sappiamo che gli **operatori di confronto** funzionano con molti oggetti di libreria (stringhe, liste, ...)
  - Come facciamo a convincere l'interprete a usarli anche con "nostri" oggetti?

# Operatori di confronto tra oggetti

- ❑ Come fa l'interprete a eseguire **l'operatore <** ?
  - Già sappiamo che tale operatore **ha un comportamento molto diverso quando i due operandi sono numeri rispetto a quando sono stringhe...**
- ❑ Ispeziona l'oggetto che rappresenta il primo operando dell'operatore < e cerca, nella classe di cui è esemplare, la definizione del metodo `__lt__` (*less than*)
  - Se tale metodo non è definito, **non ne esiste una definizione "standard"** : viene sollevata l'eccezione **TypeError** : non si può usare l'operatore <

```
class Student :  
    ...  
    def __lt__(self, other) :  
        return self.getAvgGrade() < other.getAvgGrade()
```

- ❑ Definire in una classe il metodo `__lt__` è condizione necessaria e sufficiente perché oggetti di quel tipo siano "confrontabili" usando l'operatore <
  - Questo **meccanismo di "delega"** è molto interessante: è giusto/utile che sia responsabilità del progettista della classe **Student** decidere come si effettua il confronto tra esemplari di studente! Come abbiamo visto per `__str__`

# Operatori di confronto tra oggetti

❑ Come fa l'interprete a eseguire **l'operatore <** ?

- In pratica, quando scriviamo

```
if something < somethingelse :  
    ...
```

è come se scrivessimo

```
if something.__lt__(somethingelse) :  
    ...
```

- Se nella classe di cui **something** è esemplare non è definito il metodo **\_\_lt\_\_**, viene sollevata l'eccezione **TypeError**

❑ E per la **verifica di uguaglianza/diversità**?

Ci sarà un meccanismo simile?

- Ovviamente sì... 😊

# Operatori di confronto tra oggetti

- ❑ Come fa l'interprete a eseguire **l'operatore ==** ?
  - Già sappiamo che tale operatore **ha un comportamento molto diverso quando i due operandi sono numeri rispetto a quando sono stringhe...**
- ❑ Ispeziona l'oggetto che rappresenta il primo operando dell'operatore == e cerca, nella classe di cui è esemplare, la definizione del metodo `__eq__` (*equal*)

```
class Student :
```

```
...
```

```
def __eq__(self, other) :
```

```
    return self.getAvgGrade() == other.getAvgGrade()
```

Esempio che non ha molto senso ma non abbiamo altre proprietà dell'oggetto, è un oggetto troppo semplice...

- Se tale metodo non è definito, **ne esiste una definizione "standard"** che restituisce **True** se e solo se i due riferimenti puntano **allo stesso oggetto**, cioè sono alias allo stesso oggetto
  - Se una classe definisce il metodo `__eq__`, per i suoi esemplari non viene più eseguito il metodo "standard"
- ❑ Di nuovo il **meccanismo di "delega"**: è il progettista della classe **Student** che decide come si effettua il confronto per uguaglianza tra esemplari di studente!

- ❑ Se l'elaborazione richiesta dal metodo `__eq__` è molto onerosa, può essere utile verificare per prima cosa che i due riferimenti confrontati non siano in realtà **alias al medesimo oggetto**
  - In tal caso, ovviamente, il metodo `__eq__` può restituire **True** senza fare ulteriori indagini
- ❑ Come si fa a capire se due variabili (nel nostro caso, le variabili parametro **self** e **other**) fanno riferimento al medesimo oggetto?
  - Si usa l'operatore **is**, il cui valore è **True** se e solo se i suoi due operandi sono riferimenti al medesimo oggetto (cioè, tecnicamente, solo lo stesso indirizzo)
  - Esiste anche l'analogo operatore **is not**

```
class Student :  
    ...  
    def __eq__(self, other) :  
        if self is other: # non ==, perché ?  
            return True # SOLO per risparmiare tempo...  
        return self.getAvgGrade() == other.getAvgGrade()
```

# Operatori di confronto tra oggetti

- ❑ Gli altri quattro metodi di confronto disponibili si possono sempre definire in modo che invochino `__eq__` e/o `__lt__`

```
class Student :  
    ...  
    def __ne__(self, other) : # not equal  
        return not (self == other) # invoca __eq__  
    def __le__(self, other) : # less than or equal  
        return self == other or self < other  
    def __gt__(self, other) : # greater than  
        return not (self <= other) # invoca __le__  
    def __ge__(self, other) : # greater than or equal  
        return not (self < other) # invoca __lt__  
    # attenzione a non specificare definizioni  
    # "circolari"... prima o poi si deve arrivare a  
    # __eq__ e/o __lt__
```

- ❑ Solitamente questi sei metodi si definiscono in tutte le classi per le quali le operazioni di confronto abbiano senso

Il materiale necessario per il  
Laboratorio 10  
termina qui



# Funzioni con numero variabile di argomenti e/o con argomenti predefiniti

Per semplicità, nelle slide si parla  
solo di funzioni, ma tutto vale  
anche per metodi e costruttori

# Funzioni con numero variabile di argomenti

- ❑ Come si fa a progettare una funzione, come **print**, che riceve un numero di argomenti *qualsiasi*, eventualmente anche zero?
- ❑ Sappiamo già progettare funzioni che ricevono liste/tuple, ma dal punto di vista dell'invocante è una cosa un po' diversa (anche se non molto...)

```
func1([1, 4, 3, 6]) # passo una lista  
func2((3, 2, 4))    # passo una tupla  
func3(3, 2, 4)      # vorrei scrivere così
```

- ❑ Nel terzo esempio, si potrebbe pensare: una sequenza di valori separati da virgole è uno dei modi per scrivere una tupla, quindi la funzione **func3** riceve, come unico parametro, una tupla
  - In realtà non è così, la sintassi "sequenza di valori separati da virgole" crea "quasi sempre" una tupla, ma non in questo caso (e ricordiamoci che noi **non** usiamo questa opportunità...)
  - Altrimenti, non sarebbe possibile invocare una funzione con più argomenti, diventerebbero sempre una tupla! E sarebbe un po' scomodo, poi, scrivere il codice delle funzioni...

# Funzioni con numero variabile di argomenti

- ❑ Per definire una funzione che accetta un numero variabile di argomenti, **bisogna agire sulla sua firma, premettendo un asterisco** al nome della variabile parametro che, al momento dell'invocazione, riceverà un riferimento a **una tupla contenente tutti i valori** (eventualmente nessuno) forniti come argomenti
- ❑ Nella scrittura del codice della funzione bisogna tenere presente che tale variabile parametro farà sicuramente riferimento a una tupla (eventualmente vuota)
- ❑ Nell'invocazione della funzione, bisogna soltanto sapere che accetta un numero di argomenti qualsiasi (anche zero)
  - Ovviamente ciascun singolo argomento può anche essere una lista/tupla/stringa... e gli argomenti possono essere di tipi diversi, perché una tupla può essere disomogenea...

```
def mysum( *values ) :  
    sum = 0  
    for val in values :  
        sum += val  
    return sum  
print(mysum(2,5,1,7)) # 15
```

# Funzioni con numero variabile di argomenti

- ❑ Si può anche decidere che la funzione debba avere un numero **minimo** di argomenti, ancorché variabile

```
def func(firstArg, secondArg, *values ) :  
    print(firstArg, secondArg)  
    for val in values :  
        print(val)
```

- se la funzione viene invocata con meno di due argomenti, l'interprete segnala errore
  - se la funzione viene invocata con due argomenti, la tupla assegnata a **values** sarà vuota
  - se la funzione viene invocata con più di due argomenti, la tupla assegnata a **values** non sarà vuota e conterrà gli argomenti successivi ai primi due
- ❑ Ovviamente **una funzione può avere un unico argomento "con l'asterisco" e deve essere l'ultimo**

# Funzioni con parametro predefinito

- A volte fa comodo definire una funzione che possa ricevere, ad esempio, un parametro obbligatorio e uno facoltativo, con un **valore predefinito** per il parametro facoltativo **mancante**

```
def func(paramObbligatorio, paramFacoltativo=22)
```

- Se la funzione viene invocata con due argomenti, il primo viene assegnato a **paramObbligatorio**, mentre il secondo viene assegnato a **paramFacoltativo** (ignorando il suo valore predefinito, 22)
  - Se, invece, la funzione viene invocata con un solo argomento, questo viene assegnato a **paramObbligatorio**, mentre **paramFacoltativo** assume il valore 22
  - In questo esempio, la funzione non può essere invocata con un numero di argomenti diverso da uno o due (molto diverso dall'utilizzo di un parametro "con asterisco"...)
- **La variabile parametro con valore predefinito deve essere l'ultima** nell'intestazione della funzione
  - Non è necessario che ci siano parametri obbligatori, una funzione può anche avere soltanto un parametro facoltativo

# Funzioni con più parametri predefiniti

- ❑ Si possono anche definire più parametri con valori predefiniti

```
def func(p1, p2=22, p3="xx")
```

- ❑ Fornendo due soli valori, **p3** avrà il valore predefinito

```
func(3, 7) # si ottiene p1=3, p2=7, p3="xx"
```

- ❑ Si possono anche fornire tre valori

```
func(3, 7, 2) # si ottiene p1=3, p2=7, p3=2
```

- ❑ Se si vuole fornire un valore per **p3** ma non per **p2**, bisogna necessariamente usare la tecnica del "parametro con nome" vista in **print(..., end="")**

```
func(3, p3="yy")
```

- ❑ Se si usano solo i parametri con nome, l'ordine nell'invocazione non è più rilevante

```
func(3, p3="yy", p2=7)
```

- ❑ Le diverse tecniche si possono anche combinare: parametri obbligatori, parametri con valori predefiniti, parametri in numero variabile che costruiscono una tupla...

```
def func(x, y, p2=22, p3="xx", *tp)
```

ma non esageriamo...

**Lezione 34**  
**18/12/2024**  
**ore 10.30-12.30**  
**aula Ve**

# Incapsulamento



# Incapsulamento

- ❑ Come abbiamo visto, dal punto di vista sintattico possiamo certamente **ispezionare e anche modificare il contenuto di una variabile di esemplare** di un oggetto usando semplicemente la "sintassi punto" con l'oggetto stesso, **senza passare attraverso la "mediazione" di uno dei suoi metodi**

```
# crea un nuovo oggetto e
# ne invoca il costruttore
stud = Student()
print(stud.getAvgGrade()) # 0
stud._grades.append(20) # NON FARE QUESTO!!!!
print(stud.getAvgGrade()) # 20
```

- ❑ **Questo è un PESSIMO stile di programmazione** e, in pratica, rischia di vanificare molti dei vantaggi (che ancora non abbiamo esplorato) derivanti dalla definizione di classi
  - **Dobbiamo capire fin da subito che è una cosa da evitare!**

# Incapsulamento

- ❑ **Primo problema:** i metodi dell'interfaccia pubblica della classe potrebbero (legittimamente) limitare le modifiche possibili allo stato degli oggetti

```
class Student :  
    def addGrade(self, grade) :  
        if grade < 18 :  
            raise ValueError("Voto insufficiente")  
        self._grades.append(grade)
```

- ❑ **Usando direttamente le variabili di esemplare si evitano tali filtri, "corrompendo" lo stato dell'oggetto**

```
stud = Student()  
stud.addGrade(15)           # solleva eccezione  
stud._grades.append(15)   # 15 registrato!
```

- ❑ **Se il codice di altri metodi della classe si basa sull'ipotesi che la lista dei voti NON contenga valori minori di 18, potrebbe non funzionare più**

# Incapsulamento

- ❑ **Secondo problema:** se il progettista della classe **Student** decide di **modificare la modalità di rappresentazione dello stato** (pur senza **modificare l'interfaccia pubblica**), il codice dei programmi potrebbe smettere di funzionare
- ❑ Ad esempio, come abbiamo visto, si potrebbe riprogettare **Student** usando le variabili di esemplare **\_sum** e **\_num**
  - Ovviamente bisogna riprogettare i metodi della classe
  - **I programmi scritti facendo accessi diretti alla lista \_grades non funzioneranno più**
  - **I programmi (correttamente) scritti usando soltanto l'interfaccia pubblica di **Student** continueranno a funzionare senza alcun problema**, sfruttando le nuove definizioni dei metodi, senza che sia necessaria alcuna modifica

# Incapsulamento

- ❑ **Terzo problema:** per il debugging degli oggetti, può essere molto utile visualizzare messaggi **ogni volta** che le informazioni di stato di un oggetto vengono modificate
- ❑ Questo è **molto** semplice se l'accesso agli oggetti avviene **soltanto** attraverso la loro interfaccia pubblica
  - Basta inserire il **"messaggio di tracciamento"** all'interno del metodo

```
class Student :  
    def addGrade(self, grade) :  
        print("Aggiunto voto:", grade)  
        self._grades.append(grade)
```

```
stud = Student()  
stud.addGrade(22) # visualizza messaggio  
stud._grades.append(23) # nessun messaggio!
```

# Incapsulamento

- ❑ L'utilizzo di oggetti della libreria standard di Python, che abbiamo fatto finora, si basa tutto su questo principio
  - **Usare oggetti tramite la loro interfaccia pubblica senza fare NESSUNA ipotesi sul loro funzionamento interno**, cioè sulla rappresentazione dello stato degli oggetti
  - Pensiamo insiemi e dizionari... addirittura non abbiamo le basi teoriche per capire il loro funzionamento!  
Ma li usiamo senza problemi... o quasi 😊
  - **Lo stesso principio va usato con i tipi di dati progettati da noi!**  
Quando li usiamo, dobbiamo **"far finta di non sapere" quali sono le loro variabili di esemplare**
- ❑ È un principio che sta alla base della programmazione orientata agli oggetti e si chiama **incapsulamento** (o *information hiding*)

# Incapsulamento

- ❑ Alcuni linguaggi di programmazione orientati agli oggetti spingono i programmatori a usare l'incapsulamento in modo rigido
  - Es. il linguaggio Java consente di definire *private* le variabili di esemplare, impedendone **sintatticamente** l'accesso da codice scritto al di fuori dei metodi della classe che le definisce
- ❑ Python "aiuta" i programmatori a usare l'incapsulamento, ma è meno rigido
  - L'aiuto deriva dalla convenzione di **usare, per le variabili di esemplare, nomi che iniziano con \_**
  - **Tali nomi vengono ignorati dai software che generano automaticamente la documentazione delle classi**
  - Ovviamente rimane la possibilità di ispezionare direttamente i file sorgente delle classi, ma evitiamo!!!
  - **Rispettiamo rigidamente l'incapsulamento**

# Ereditarietà

# L'ereditarietà

- ❑ L'*ereditarietà* è uno dei principi basilari della programmazione orientata agli oggetti
  - insieme all'*incapsulamento* e al *polimorfismo*
- ❑ L'ereditarietà è il paradigma che consente, tra le altre cose, il *riutilizzo del codice*
  - si usa quando si deve realizzare una classe ed è già disponibile un'altra classe che rappresenta *un concetto più generale, meno specifico*
  - oppure quando *due o più classi condividono un comportamento comune*





# L'ereditarietà

- ❑ Immaginiamo di aver progettato una classe **Student**
  - Variabili di esemplare: nome e cognome, codice fiscale, numero di matricola, lista dei voti conseguiti
  - Costruttore che riceve: **nome e cognome, codice fiscale**, numero di matricola
  - Metodi che ispezionano le singole variabili di esemplare
  - Metodo che aggiunge un voto alla lista
  - Metodo che ottiene il voto medio
- ❑ Immaginiamo, ora, di voler progettare una classe **Teacher**, per rappresentare un docente universitario
  - Gli esemplari di **Teacher** avranno alcune **caratteristiche in comune** con esemplari di **Student**
    - **Nome e cognome, codice fiscale**
  - Il costruttore sarà molto simile, così come i metodi che ispezionano le singole variabili di esemplare
  - Un docente avrà, poi, una lista di corsi e i metodi che servono a gestirla... mentre non avrà una lista di voti, né una matricola

# L'ereditarietà



- ❑ Per progettare la classe **Teacher** potremmo partire da **una copia della classe Student**, apportando le modifiche necessarie
- ❑ **Non è una strategia molto efficace...**
  - Se in seguito si decide, ad esempio, di introdurre nel costruttore una verifica del codice fiscale, bisognerà introdurre modifiche identiche nei costruttori delle due classi
  - E se si sono progettate anche le classi "addetto amministrativo", "direttore di dipartimento" (che è un docente con caratteristiche aggiuntive...) bisogna fare la stessa cosa anche in quelle...
  - Sarebbe molto utile poter **scrivere UNA VOLTA SOLA il codice condiviso**
- ❑ Occorre per prima cosa **identificare le caratteristiche comuni** e definire, con quelle, **un'entità più astratta** delle due/tre/quattro entità in esame, ad esempio **una "persona", che ha un nome, un cognome e un codice fiscale**
  - studenti e docenti saranno **persone con peculiarità diverse**

```
class Person :  
  
    def __init__(self, name, stateID) :  
        self._name = name # stringa unica per nome e cognome  
        # qui ci potrebbe essere il codice che verifica la  
        # correttezza del codice fiscale, stateID;  
        # facciamo una verifica semplice  
        if len(stateID) != 16 : raise ValueError("Cod.Fiscale")  
        self._stateID = stateID  
  
    # definiamo metodi di ispezione per evitare che si acceda  
    # direttamente alle variabili di esemplare: pratica comune  
    def getName(self) :  
        return self._name  
  
    def getStateID(self) :  
        return self._stateID  
  
# non ci sono metodi che modificano le variabili di esemplare  
# perché in questo esempio non vogliamo che si possa fare...  
#  
# una "persona" più realistica avrebbe probabilmente anche  
# data e luogo di nascita, cittadinanza (modificabile), ecc.
```

Gli oggetti di tipo **Person** sono abbastanza inutili... non hanno metodi di elaborazione... servono soltanto come "base" per costruire "persone specializzate"

# L'ereditarietà

- ❑ È evidente che **dal punto di vista logico studenti e docenti sono "persone"** ... ma come facciamo a "spiegarlo" all'interprete Python?
- ❑ Vorremmo poter fare così...
  - Definisco la classe **Student**, i cui esemplari sono esemplari di **Person** con **alcune caratteristiche aggiuntive** (un numero di matricola, una lista di voti...) e **alcune funzionalità aggiuntive** (ispezione del numero di matricola e del voto medio, aggiunta di un voto)
  - Definisco la classe **Teacher**, i cui esemplari sono esemplari di **Person** con **alcune caratteristiche aggiuntive** (una lista di corsi...) e **alcune funzionalità aggiuntive** (ispezione della lista dei corsi, aggiunta/eliminazione di un corso)
  - **Ovviamente vorremmo evitare di dover ripetere il codice già scritto nella classe Person, non tanto per la copiatura in sé, ma per la difficoltà di manutenzione che ne sarebbe conseguenza**

```
class Student(Person) :   
  
    def __init__(self, name, stateID, univID) :  
         super().__init__(name, stateID)  
        self._univID = univID # matricola  
        self._grades = list() # lista di voti (vuota)  
  
    def getUnivID(self) :  
        return self._univID  
  
    def getAvgGrade(self) :  
        if len(self._grades) == 0 : return 0  
        return sum(self._grades) / len(self._grades)  
  
    def addGrade(self, grade) :  
        self._grades.append(grade)
```

- ❑ Ora **vedremo i dettagli**, ma osserviamo subito che il codice di **Person** NON è stato ricopiato
- ❑ Come vedremo, con un oggetto di tipo **Student** possiamo invocare (oltre ai metodi specifici di **Student**) anche i metodi **getName** e **getStateID**, **come se fosse un oggetto di tipo Person**

# L'ereditarietà

```
class Student (Person) :  
  
    ...
```


□ Prima novità sintattica

□ Dopo il nome della classe che stiamo definendo, indichiamo **tra parentesi tonde** il nome della classe di cui vogliamo **ereditare il comportamento**

- Si usano i termini **sottoclasse** (in questo caso, **Student**) e **superclasse** (in questo caso, **Person**)
  - Gli esemplari di una **sottoclasse** sono un **sottoinsieme** degli esemplari della superclasse (facile da ricordare...)
  - Si dice anche che **la sottoclasse deriva o è derivata dalla superclasse**
- Attenzione: **in Python esiste anche l'ereditarietà multipla**, cioè posso dichiarare che una sottoclasse ha due o più superclassi, in modo che erediti l'unione dei loro comportamenti
  - In alcuni linguaggi, come Java, l'ereditarietà multipla non esiste, ma esiste, ad esempio, in C++
  - **È una caratteristica di difficile utilizzo, che EVITEREMO**
- Possiamo usare, invece, **l'ereditarietà su più livelli**: una classe B che sia sottoclasse di A può diventare la superclasse di un'altra classe, C, la quale erediterà quindi il comportamento di B (e, di conseguenza, anche il comportamento di A, in quanto superclasse di B)

# L'ereditarietà

```
class Student (Person) :  
  
    ...
```


- ❑ Con questa definizione, **tutto funziona come se tutti i metodi presenti nella superclasse fossero definiti esattamente nello stesso modo nella sottoclasse**
  - C'è qualche differenza soltanto nel caso del costruttore 
- ❑ **Un esemplare di una sottoclasse funziona anche come un esemplare della sua superclasse (*molto comodo...*), anche se ha "qualcosa in più"...**
  - Quindi, se una funzione si aspetta di ricevere ed elaborare un esemplare di **Person**, accetta ed elabora correttamente anche un esemplare di **Student** o di **Teacher** (quando avremo definito la classe **Teacher**)
  - È ragionevole: se è stata progettata per elaborare persone, non avrà bisogno delle caratteristiche peculiari di studenti o docenti e utilizzerà soltanto le loro caratteristiche comuni (quelle di una persona)

Si chiama  
principio di  
sostituzione



# L'ereditarietà

```
class Student(Person) :  
    ...
```

- ❑ Con il costruttore c'è un piccolo problema...
- ❑ Il costruttore di una classe **deve** chiamarsi `__init__`
  - La classe **Person** ha il suo costruttore, che si chiama `__init__`
  - La classe **Student** ha il suo costruttore, che si chiama `__init__`
- ❑ Tutti i metodi della superclasse si comportano come se fossero definiti allo stesso modo nella sottoclasse.  
Quando viene costruito un esemplare di una sottoclasse,  
**quale metodo `__init__` viene eseguito, dato che ne ha due?**
- ❑ Una nuova regola sintattica
  - **Quando una sottoclasse definisce un metodo che ha lo stesso nome di un metodo della superclasse, quello della sottoclasse PREVALE**
  - Come vedremo, la regola non vale soltanto per i costruttori 
- ❑ Quindi, costruendo un esemplare di una sottoclasse, viene invocato ed eseguito **soltanto** il metodo `__init__` della sottoclasse stessa



- ❑ Il compito del costruttore è quello di assegnare un valore iniziale alle variabili di esemplare dell'oggetto in fase di costruzione
- ❑ Ma un oggetto che sia esemplare di una sottoclasse eredita (cioè "ha anche") le variabili di esemplare della superclasse (oltre ai metodi) !
  - Oltre ad aggiungere, eventualmente, variabili proprie
- ❑ Ovviamente potremmo scrivere così

```
class Student(Person) :  
    def __init__(self, name, stateID, univID) :  
        # c'era super().__init__(name, stateID)  
        self._name = name  
        # qui magari ci dimentichiamo il codice che verifica  
        # la correttezza di stateID...  
        self._stateID = stateID  
        self._univID = univID  
        self._grades = list() # lista di voti (vuota)  
        ...
```

- ❑ Ma perderemmo molti dei vantaggi derivanti dal fatto di non aver ricopiato codice dalla superclasse!

Perché **ne abbiamo comunque ricopiato una parte...**

- Se il progettista della superclasse decidesse di modificare la rappresentazione delle informazioni di stato della superclasse (es. il codice fiscale come lista, scomposto nelle sue parti), la sottoclasse non funzionerebbe più...

❑ Il costruttore della sottoclasse deve **DELEGARE** al costruttore della superclasse il compito di costruire quella "porzione di oggetto della sottoclasse" che costituisce un esemplare della superclasse

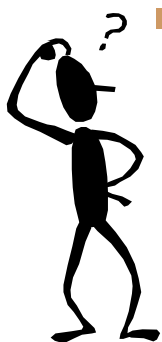
❑ In pratica, **il costruttore della sottoclasse deve invocare il costruttore della superclasse**, passando i parametri opportuni per quel costruttore "parziale"

❑ Quando, in un metodo di una sottoclasse, voglio invocare un metodo della superclasse, mi comporto come se fosse un metodo della sottoclasse stessa, cioè scrivo

```
def metodoDellaSottoClasse(self, ...) :  
    self.metodoDellaSuperClasse(...)  
    ...
```

❑ Ma se i due metodi hanno lo stesso nome, questo innesca una **ricorsione** (non voluta) ! Avverrebbe con **`__init__`**

■ Si avrebbe una ricorsione (?) perché il metodo della sottoclasse prevale su quello della superclasse avente lo stesso nome e **il metodo invocherebbe se stesso all'infinito**: apparentemente non c'è modo di invocare un metodo della superclasse che è stato "sovrascritto" nella sottoclasse...



❑ C'è una sintassi speciale per invocare un metodo della superclasse, che si usa quando nella sottoclasse c'è un metodo con lo stesso nome (in particolare, il costruttore!)

❑ Anziché usare il riferimento **self**, si usa il riferimento restituito dall'invocazione della funzione predefinita **super()** [che non vuole parametri]

```
class Student(Person) :  
    def __init__(self, name, stateID, univID) :  
        super().__init__(name, stateID)  
    ...
```

❑ Cosa restituisce **super()** ?

- Quando la funzione predefinita **super()** viene invocata all'interno di un metodo/costruttore di una sottoclasse, restituisce il riferimento alla "**porzione** dell'oggetto a cui fa riferimento **self** che costituisce un esemplare della superclasse"

• Tecnicamente non è proprio così (perché **non esiste un vero e proprio esemplare della superclasse all'interno di un esemplare di una sottoclasse**), ma "tutto funziona come se" fosse così



## ❑ Ora, ecco un possibile progetto della classe **Teacher**

```
class Teacher(Person) :  
  
    def __init__(self, name, stateID) :  
        super().__init__(name, stateID)  
        self._courses = list() # lista di corsi (vuota)  
  
    def addCourse(self, course) :  
        self._courses.append(course)  
  
    def removeCourse(self, course) :  
        self._courses.remove(course)  
  
    def getCourses(self) :  
        return tuple(self._courses)  
        # non restituisco self._courses, altrimenti dall'esterno  
        # si potrebbe modificare la lista dei corsi del docente,  
        # mentre voglio che questo avvenga soltanto usando i  
        # metodi addCourse e removeCourse, quindi clono la lista  
        # e restituisco una tupla con lo stesso contenuto  
        # (potrei anche restituire una lista, l'importante è  
        # clonare, ma restituendo una tupla rendo più chiaro che  
        # non ha senso modificarla, perché non avrebbe effetto  
        # sul docente...) COMMENTO IMPORTANTE
```

```
class Teacher(Person) :  
    ...  
    def addCourse(self, course) :  
        self._courses.append(course)  
    def removeCourse(self, course) :  
        self._courses.remove(course)
```

- ❑ Perché è così utile che le modifiche alla lista dei corsi siano possibili soltanto attraverso i metodi **addCourse** e **removeCourse**? Ricordiamo l'incapsulamento...
- ❑ Ad esempio, quando viene aggiunto o rimosso un corso, potrebbe essere utile scrivere un messaggio in un file di *log*, che tenga traccia di tutte le modifiche, oppure inviare un messaggio di posta
- ❑ Con questo stile di progetto, è sufficiente aggiungere il codice necessario ai due metodi
- ❑ Se, invece, fosse possibile modificare le liste dei corsi dei docenti in qualsiasi punto del codice di gestione dell'ateneo, dovrei cercare tutti i punti in cui questo avviene, e apportare le modifiche opportune
  - Una "tragedia" dal punto di vista dell'ingegneria del software
  - **La progettazione a oggetti, usata in modo opportuno, è uno strumento formidabile per l'ingegneria del software**

**Lezione 35**  
**20/12/2024**  
**ore 16.30-18.30**  
**aula Ve**

# Progettazione di eccezioni

# Progettazione di eccezioni

- ❑ Come facciamo a sollevare **un'eccezione con un nome di nostra scelta**, anziché doverne scegliere uno di quelli presenti nella libreria?
- ❑ **L'enunciato `raise` vuole "un oggetto che rappresenti un'eccezione"**
  - Se è un oggetto, sarà esemplare di una classe... anche se è abbastanza ovvio che non potrà essere una classe "qualsiasi"...
  - Se proviamo a eseguire **`raise`** con una lista o una stringa o un numero intero... otteniamo il messaggio seguente

```
>>> raise 2
TypeError: exceptions must derive from BaseException
```

- Scopriamo che l'oggetto deve essere un esemplare di **`BaseException`** o di una sua sottoclasse



# Progettazione di eccezioni

❑ L'enunciato **raise** vuole "un oggetto che rappresenti un'eccezione"

- L'oggetto deve essere un esemplare di **BaseException** o di una sua sottoclasse
- Quindi, dobbiamo progettare una sottoclasse di **BaseException** con il nome che vogliamo, poi solleveremo un suo esemplare!

```
class MyException(BaseException) :  
    ...  
  
raise MyException # esempio
```

- Che variabili di esemplare deve avere? Che metodi deve avere?
  - Non serve aggiungere niente al comportamento di **BaseException**! **Definiamo una classe soltanto per ottenere un diverso nome di tipo!**  
Non ci serve un comportamento diverso... né proprietà diverse...
  - Forse ci deve essere almeno il costruttore?
    - No, non avendo variabili di esemplare oltre a quelle ereditate, **è sufficiente il costruttore ereditato**, che viene invocato perché è come se fosse definito nella sottoclasse (come tutti i metodi della superclasse) e non va in conflitto con il costruttore della sottoclasse perché... non c'è!

# Progettazione di eccezioni

- ❑ Posso lasciare la classe "vuota" ?

```
class MyException(BaseException) :  
    # lascio vuoto il corpo della classe  
  
raise MyException # esempio
```

- No, viene segnalato un errore di sintassi

```
IndentationError: expected an indented block
```

- ❑ Sappiamo già cosa usare  
come "**riempitivo**"...

```
class MyException(BaseException) :  
    pass
```

- ❑ Quindi, ecco il progetto **completo** di una **eccezione specifica**,  
con un nome scelto da noi

# Variabili di classe

# Variabili di classe

- ❑ Torniamo al **nostro progetto originario** di studente e aggiungiamo, come ulteriore attributo, il numero di matricola
- ❑ Aggiungiamo all'interfaccia pubblica di **Student** il metodo **getID()**
  - Solitamente i nomi di metodi di "ispezione" di attributi iniziano con **get**
  - Ovviamente ci sarà una nuova variabile di esemplare, **\_ID**, che si rende necessaria per il funzionamento del metodo **getID**
    - Meglio usare le maiuscole perché, **per ciascuno studente**, il numero di matricola sarà una costante
- ❑ Decidiamo che il numero di matricola debba essere **assegnato automaticamente agli studenti**, nel momento in cui gli oggetti che li rappresentano vengono costruiti (anche se non molto realistico... serve per costruire un esempio...)
  - Sarà compito del costruttore
  - Ogni studente avrà un numero di matricola superiore di un'unità rispetto a quello dell'ultimo studente creato in precedenza

# Variabili di classe

- ❑ Ogni studente avrà un numero di matricola superiore di un'unità rispetto a quello dell'ultimo studente creato in precedenza

```
class Student :  
  
    def __init__(self) :  
        self._grades = list()  
        self._ID = ... # cosa scriviamo qui ???  
  
    def getID(self) : # questo è semplice...  
        return self._ID  
  
    ... # il codice degli altri metodi
```

- ❑ Come facciamo?

# Variabili di classe

- ❑ Ogni studente avrà un numero di matricola superiore di un'unità rispetto a quello dell'ultimo studente creato in precedenza
- ❑ **Come facciamo?**
- ❑ Ogni volta che viene creato uno studente, dobbiamo ricordarci il numero di matricola che gli viene assegnato
- ❑ **Non è sufficiente** memorizzare questa informazione **all'interno dello studente stesso**: quando creiamo lo studente successivo, come facciamo a recuperare il numero di matricola assegnato al precedente?
  - **Dobbiamo memorizzarlo in uno spazio condiviso tra tutti gli studenti!**
    - Concetto analogo alle costanti di classe...
  - **Ci serve una variabile di classe**

# Variabili di classe

- ❑ Ogni studente avrà un numero di matricola superiore di un'unità rispetto a quello dell'ultimo studente creato in precedenza

```
class Student :  
  
    _lastAssignedID = 0 # non è una costante...  
  
    def __init__(self) :  
        self._grades = list()  
        Student._lastAssignedID += 1  
        self._ID = Student._lastAssignedID  
  
    def getID(self) :  
        return self._ID  
  
    ... # il codice degli altri metodi
```

- ❑ Di solito è bene che le variabili di classe siano "private" come le variabili di esemplare: lo vediamo dal nome che inizia con \_

# Costanti di classe

```
class Student :  
  
    PASSING_GRADE = 18  
  
    def addGrade(self, grade) :  
        if grade < Student.PASSING_GRADE :  
            raise ValueError  
        self._grades.append(grade)
```

- ❑ Quando una variabile di classe è **costante**, non è necessario renderla "privata", possiamo darle un nome che NON inizi con \_
- ❑ Usiamo, però, un **nome composto da lettere maiuscole**, come per tutte le costanti
  - I programmatori che accedono dall'esterno a tale costante di classe devono rispettare il fatto che sia costante!
- ❑ Se, però, il progettista della classe ritiene che non sia di nessuna utilità "esporre" all'esterno il valore di una costante di classe, le può comunque dare un nome "privato", che inizi con \_



# Ancora sul costruttore

- ❑ Anche se non è strettamente necessario, nelle classi definite in questo corso **usiamo sempre il costruttore per assegnare un valore a TUTTE le variabili di esemplare** della classe
  - Evitiamo di studiare soluzioni alternative!
- ❑ Le **variabili (e costanti) di classe**, invece, vengono inizializzate all'esterno dei metodi
  - **L'interprete garantisce che la loro inizializzazione avvenga prima della creazione di qualsiasi esemplare della classe, in modo che possano essere utilizzate all'interno di metodi e costruttore, avendo già un valore**

# Classe Student completa

```
# student.py (modulo)

class Student : # mancano i commenti!

    PASSING_GRADE = 18
    _lastAssignedID = 0

    def __init__(self) :
        Student._lastAssignedID += 1
        self._ID = Student._lastAssignedID
        self._grades = list()

    def getID(self) :
        return self._ID

    def addGrade(self, grade) :
        if grade < Student.PASSING_GRADE :
            raise ValueError
        self._grades.append(grade)

    def getAvgGrade(self) :
        if len(self._grades) == 0 : return 0
        return sum(self._grades)/len(self._grades)
```

# Metodi di classe

# Metodi di classe?

- ❑ Abbiamo visto **variabili di esemplare** e **variabili di classe**
- ❑ I metodi che abbiamo visto finora, che hanno il parametro predefinito **self**, si dicono anche *metodi di esemplare*
- ❑ Quindi esistono anche *metodi di classe* ? **Sì**
- ❑ Che significato può avere un metodo di classe?
  - **In pratica è una funzione**, nel senso che **non ha parametro implicito** (quindi **non** viene invocato usando un esemplare della classe, ma usando **il nome della classe**), e viene inserito nella definizione della classe perché **svolge un'elaborazione "attinente" ai compiti attribuiti agli esemplari della classe**
    - Es. nella classe **Student**, un metodo di classe che traduca un voto dal sistema anglosassone al sistema italiano
  - Inoltre, **ha il "permesso" di accedere alle variabili di classe "private" (oltre che "pubbliche"), potendole anche modificare**
  - **[caso poco frequente]** Ha anche il "permesso" di accedere alle variabili di esemplare, potendole anche modificare, **se riceve esemplari della classe come parametri espliciti**



# Metodi di classe

- Spesso i metodi di classe svolgono elaborazioni che riguardano le variabili di classe (ma non solo)
  - Esempio: "saltare" alcuni numeri di matricola, che non verranno assegnati (ad esempio, per riservarli ad altri scopi)

```
class Student :  
  
    _lastAssignedID = 0  
  
    def stepLastAssignedID(x) :  
        Student._lastAssignedID += x  
  
    ...
```

- Si invocano usando il nome della classe

```
stud1 = Student() # matricola 1  
stud2 = Student() # matricola 2  
Student.stepLastAssignedID(10)  
stud3 = Student() # matricola 13
```

# Ereditarietà: Metodi sovrascritti

- ❑ Ora vogliamo progettare una classe che descriva un docente in pensione (**RetiredTeacher**), che può avere **un solo corso**
- ❑ Se definiamo **RetiredTeacher** come sottoclasse di **Teacher**, dobbiamo chiederci
  - Le informazioni di stato di un oggetto di tipo **Teacher** sono adeguate per un oggetto di tipo **RetiredTeacher**?
    - Sì
  - È un'eredità "di secondo livello", perché eredita anche da **Person**
  - Il comportamento (cioè i metodi) di un oggetto di tipo **Teacher** è adeguato per un oggetto di tipo **RetiredTeacher**?
    - In parte... però vogliamo che il comportamento del metodo **addCourse** sia un po' diverso: deve impedire l'aggiunta di un corso a un docente pensionato che ne abbia già uno
  - Finora abbiamo visto che una classe eredita i metodi della superclasse e può definire ulteriori metodi
  - Ma **possiamo ereditare un metodo modificandone il comportamento?**
    - **Sì, basta sovrascriverlo** (o, si dice anche, *ridefinirlo*), come abbiamo fatto con il costruttore

```
class RetiredTeacher(Teacher) : # quindi anche da Person...
                                # ma "non ho bisogno di saperlo"
    # il costruttore non serve: quello ereditato è adeguato,
    # perché non ci sono variabili di esemplare aggiuntive

    # sovrascrivo il metodo addCourse ereditato
    def addCourse(self, course) :
        if len(self._courses) > 0 :
            raise ValueError("Too many courses for retired teacher")
        self._courses.append(course)
```

- ❑ Quando il metodo **addCourse** verrà invocato con un oggetto di tipo **RetiredTeacher**, verrà eseguito il metodo definito nella classe **RetiredTeacher** e non quello ereditato
- ❑ Quando si invoca il metodo **addCourse** con un oggetto di tipo **Teacher** ovviamente non cambia nulla, viene eseguito il metodo **addCourse** definito nella classe **Teacher**
  - **La progettazione di una sottoclasse non può mai alterare il comportamento di esemplari della sua superclasse**



```
class RetiredTeacher(Teacher) :  
  
    # il costruttore non serve: quello ereditato è adeguato,  
    # perché non ci sono variabili di esemplare aggiuntive  
  
    # sovrascrivo il metodo addCourse ereditato  
    def addCourse(self, course) :  
        if len(self._courses) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self._courses.append(course)
```

❑ Questa soluzione, però, non rispetta le regole dell'incapsulamento... che vale anche tra sottoclasse e superclasse!

- **Accede direttamente alle variabili di esemplare della superclasse**
- Se il progettista della superclasse modifica la rappresentazione delle informazioni di stato, che è **un dettaglio interno della classe**, pur senza modificarne l'interfaccia pubblica (ad esempio, la lista dei nomi di corsi diventa un insieme), questa sottoclasse non funziona più, perché ipotizza che **self.\_courses** sia una lista

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self._courses) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self._courses.append(course)
```

- ❑ Questa soluzione non rispetta le regole...
- ❑ L'approccio corretto è quello già seguito con i costruttori
  - La gestione delle informazioni di stato relative alla superclasse deve essere delegata alla superclasse, tramite i suoi metodi (ereditati)
  - Vediamo come fare una prima modifica, quella più semplice

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self.getCourses()) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self._courses.append(course)
```

- Questo è corretto: per sapere quanti corsi sono stati assegnati al docente (pensionato), invoco il metodo **getCourses** (ereditato dalla superclasse), che restituisce la tupla dei corsi, della quale vado poi a calcolare la lunghezza

## ❑ Proviamo a sistemare anche l'altro "errore"

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self.getCourses()) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self.getCourses().append(course) # non funziona
```

❑ Questo non funziona! Perché il metodo **getCourses** non restituisce la lista dei corsi, ma una tupla (quindi, non modificabile) avente lo stesso contenuto

- Scelta progettuale fatta proprio per proteggere la lista dei corsi, che non si vuole sia modificabile se non passando per l'invocazione del metodo **addCourse**
- Quindi per aggiungere un corso alla lista dei corsi dobbiamo necessariamente invocare **addCourse della superclasse!**

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self.getCourses()) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self.addCourse(course) # attenzione: non funziona
```

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self.getCourses()) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        self.addCourse(course) # attenzione: non funziona
```

- ❑ Qui l'intenzione era quella di invocare il metodo **addCourse** ereditato dalla superclasse, invece (usando **self**) viene invocato il metodo **addCourse** della sottoclasse (che volutamente sta sovrascrivendo, cioè "nascondendo", quello ereditato...)



- **Si innesca una ricorsione infinita!**
- ❑ Come facciamo a invocare esplicitamente il metodo **addCourse** della superclasse?
  - Usiamo la funzione **super()**, come nei costruttori

```
class RetiredTeacher(Teacher) :  
    def addCourse(self, course) :  
        if len(self.getCourses()) > 0 :  
            raise ValueError("Too many courses for retired teacher")  
        super().addCourse(course) # ora funziona
```

- ❑ Quando i metodi NON sono sovrascritti il problema non si pone
  - Per **metodi ereditati e NON sovrascritti**, come **getCourses**, **self.metodo** è equivalente a **super().metodo**

La funzione `isinstance`

# La funzione `isinstance`

- ❑ Quando una funzione o un metodo/costruttore dichiara di ricevere un parametro, solitamente richiede che il valore effettivamente ricevuto come argomento in ciascuna invocazione sia di un determinato tipo (es. un numero intero, una stringa, una lista...)
- ❑ Questa verifica si può fare confrontando il valore restituito dalla funzione `type` (a cui forniamo l'argomento ricevuto) con **il nome del tipo di dato** che vogliamo

```
def func(x) : # x deve essere un numero intero
    if type(x) != int :
        raise ValueError(str(x) + "non è int")
    ... # codice della funzione,
        # qui x è CERTAMENTE un numero intero
```

- ❑ Usando l'ereditarietà, la situazione si complica un po'
  - Se la classe Y deriva dalla classe X (es. **Student** deriva da **Person**...), allora una funzione che richiede un parametro di tipo X deve ragionevolmente accettare un esemplare di Y, **anche senza sapere che esiste la classe Y !**

# La funzione `isinstance`

- ❑ Se la classe `Y` deriva dalla classe `X` (es. `Student` deriva da `Person...`), allora una funzione che richiede un parametro di tipo `X` deve ragionevolmente accettare un esemplare di `Y`, **anche senza sapere che esiste la classe `Y` !**

```
class Person :  
    ...  
class Student(Person) : # sottoclasse di Person  
    ...  
def func(p) : # p deve essere di tipo Person  
    if type(p) != Person : raise ValueError  
    ... # qui p è sicuramente un esemplare di Person  
s = Student(...) # un esemplare di Student  
func(s) # per il "principio di sostituzione"  
        # dovrebbe funzionare, invece solleva eccezione
```

- ❑ Invece della funzione `type`, si usa la funzione predefinita `isinstance`, che **riceve un oggetto e un nome di tipo** e restituisce `True` se e solo se l'oggetto è un esemplare **di quel tipo o di un suo sottotipo!**

```
def func(p) :  
    if not isinstance(p, Person): raise ValueError  
    ... # qui p è sicuramente un esemplare di Person  
        # O DI UNA CLASSE DERIVATA da Person  
s = Student(...)  
func(s) # OK
```

# La funzione `isinstance`

- ❑ Quando scriviamo una funzione che vuole ricevere un parametro di un determinato tipo, non sappiamo (né vogliamo sapere...) se esistono classi derivate da quel tipo
- ❑ **In pratica, quindi, dovremmo sempre usare `isinstance` invece di `type`**
  - La funzione **`type`** si usa (soltanto) per ottenere informazioni più specifiche in fase di debugging
    - Ad esempio, se voglio sapere se la funzione ha ricevuto specificatamente un esemplare di una classe o di una sua sottoclasse
  - L'uso di **`type`** ha comunque senso nella verifica di tipi di dati "fondamentali" presenti nella libreria standard (es. **`int`**, **`str`**, **`list`**...), perché **solitamente non si definiscono sottoclassi di tali classi**
    - Quindi, per quei tipi di dati, **`type`** è sostanzialmente equivalente a **`isinstance`** (anche se tecnicamente non lo è)
  - **D'ora in avanti usiamo sempre `isinstance`**



Il materiale necessario per il  
**Laboratorio 11**  
termina qui

Il Laboratorio 11 verrà  
pubblicato durante le vacanze  
(vi avviserò quando sarà pronto)

**Lezione 36**  
**07/01/2025**  
**ore 10.30-12.30**  
**aula Ve**

Ricerca per bisezione  
(o ricerca "binaria")

# Ricerca in una lista/tupla ordinata

- ❑ Il problema di verificare, all'interno di una lista (o tupla), la presenza di un elemento che contenga uno specifico valore può essere affrontato in modo *molto più efficiente se la lista è ordinata* secondo qualche criterio di ordinamento
- ❑ Diciamo che una lista è **ordinata in senso crescente** se il **valore dei suoi elementi aumenta all'aumentare dell'indice**

|   |   |    |    |    |    |
|---|---|----|----|----|----|
| 5 | 9 | 11 | 12 | 17 | 21 |
|---|---|----|----|----|----|

*a*[0]   *a*[1]   *a*[2]   *a*[3]   *a*[4]   *a*[5]

- Se ci sono elementi replicati (ovviamente consecutivi), diciamo che la lista è ordinata in senso "non decrescente" o "crescente in senso lato"
- Ovviamente una lista può anche essere ordinata in senso decrescente (o "non crescente"), cioè con valori decrescenti (o "non crescenti") al crescere dell'indice

# Ricerca in una lista ordinata

- Il problema di verificare, all'interno di una lista, la presenza di un elemento che contenga uno specifico valore può essere affrontato in modo *molto più efficiente se la lista è ordinata*

|      |      |      |      |      |      |
|------|------|------|------|------|------|
| 5    | 9    | 11   | 12   | 17   | 21   |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] |

- Cerchiamo il valore **12**
- Confrontiamo **12** con il valore che si trova **(circa) al centro della lista**, ad esempio **a[2]**, che è **11**
- Il valore che cerchiamo, **12**, è maggiore di **11**
  - Se è presente nella lista, sarà a destra di 11*
  - Certamente le posizioni a sinistra di 11 non contengono l'elemento cercato*

# Ricerca in una lista ordinata

La porzione  
scura non verrà  
più esaminata

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| 5      | 9      | 11     | 12     | 17     | 21     |
| $a[0]$ | $a[1]$ | $a[2]$ | $a[3]$ | $a[4]$ | $a[5]$ |

- ❑ A questo punto dobbiamo cercare il valore **12** nella sola sotto-lista che si trova a destra di  $a[2]$ 
  - La sotto-lista che contiene le celle in posizione 0, 1 e 2 non verrà più presa in esame, perché **sicuramente NON** contiene il valore cercato
- ❑ Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro della porzione da esaminare,  $a[4]$ , che è **17**
- ❑ Il valore che cerchiamo è minore di **17**
  - *se è presente nella lista, sarà a sinistra di 17*

# Ricerca in una lista ordinata

Le porzioni scure  
non verranno più  
esaminate

|        |        |        |        |        |        |
|--------|--------|--------|--------|--------|--------|
| 5      | 9      | 11     | 12     | 17     | 21     |
| $a[0]$ | $a[1]$ | $a[2]$ | $a[3]$ | $a[4]$ | $a[5]$ |

- ❑ Anche la sotto-lista che contiene le celle in posizione 4 e 5 non verrà più presa in esame
- ❑ A questo punto dobbiamo cercare il valore **12** nella sotto-lista composta dal solo elemento  $a[3]$
- ❑ Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro,  $a[3]$ , che è **12**
- ❑ Il valore che cerchiamo è uguale a **12**
  - *Il valore che cerchiamo è presente nella lista (e si trova in posizione 3)*

# Ricerca in una lista ordinata

|   |   |    |    |    |    |
|---|---|----|----|----|----|
| 5 | 9 | 11 | 12 | 17 | 21 |
|---|---|----|----|----|----|

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$   $a[5]$

- ❑ Se il valore da cercare fosse stato 13, dopo aver eseguito gli stessi passi, il confronto con l'elemento  $a[3]$  avrebbe dato esito negativo e avremmo cercato nella sotto-lista *di lunghezza zero* a destra della posizione 3, concludendo che **il valore cercato non è presente nella lista**
- ❑ Questo algoritmo si chiama *ricerca per bisezione* (o ricerca binaria), perché a ogni passo divide la lista rimasta *in due parti aventi (circa) le stesse dimensioni*
- ❑ ***Funziona soltanto se la lista è ordinata***
  - *Se la lista non è ordinata, l'algoritmo fornisce, in generale, una risposta errata (si parla di "falsi negativi")*



# Ricerca in una lista ordinata

- ❑ Questo algoritmo si chiama *ricerca per bisezione* (o ricerca binaria), perché a ogni passo divide la lista rimasta *in due parti aventi circa le stesse dimensioni* e *funziona soltanto se la lista è ordinata*
  - Si dimostra che **la ricerca per bisezione in una lista ordinata è MOLTO più efficiente**, in termini di tempo di esecuzione, della ricerca sequenziale (che, ovviamente, funziona anche se la lista è ordinata)
- ❑ **Quindi, se in una raccolta di dati memorizzata in una lista si devono fare MOLTE ricerche, può essere utile disporre i dati nella lista in modo che siano ordinati**
  - Se le ricerche da fare sono poche, bisogna valutare se valga la pena ordinare la sequenza, perché anche l'azione di ordinamento ha un costo in termini di tempo di elaborazione

# Ricerca in una lista ordinata

## □ Esempio di ricerca in una lista avente **un milione di elementi**

- **Se la lista NON è ordinata**, posso fare soltanto una ricerca sequenziale: devo ispezionare mediamente 500mila elementi per trovare quello cercato e, **nel caso peggiore, devo ispezionare e confrontare un milione di elementi** (in particolare, questo accade ogni volta in cui l'elemento cercato **non** è presente nella lista)
- **Se la lista è ordinata**, al primo passo della ricerca per bisezione riduco la dimensione della zona di ricerca da un milione a (circa) 500mila! Al passo successivo scendo a (circa) 250mila... **il numero di valori che vengono ispezionati è, nel caso peggiore, uguale al più piccolo numero intero che sia maggiore o uguale al logaritmo in base due della lunghezza della lista** (che è il numero di divisioni per due a cui posso sottoporre tale lunghezza, prima che diventi un numero minore di uno)
  - **Nel caso peggiore dovrò ispezionare (e confrontare con l'elemento cercato) VENTI valori invece di un milione!**

# Ricerca per bisezione nella libreria?

- ❑ Il metodo **index**, l'operatore **in** e l'operatore **not in**, che conosciamo, per fare ricerche in liste/tuple **NON usano la ricerca per bisezione**, bensì **la ricerca sequenziale**
  - **Perché non sanno se la lista/tupla sia ordinata oppure no!**
    - **Verificare se una sequenza è ordinata richiederebbe lo stesso tempo di una ricerca sequenziale**
    - Come si verifica se una lista è ordinata?
      - Bisogna confrontare ciascuna coppia di elementi consecutivi, verificando che il primo sia minore (o, meglio, non maggiore) del secondo
      - Quindi, per verificare se una lista è ordinata, bisogna ispezionare tutti i suoi elementi:  
tanto vale confrontarli per uguaglianza con l'elemento cercato!

# Ricerca per bisezione nella libreria?

- ❑ Il modulo `bisect` contiene la funzione `bisect` che consente di fare ricerche per bisezione in liste/tuple ordinate (utilizzo poco intuitivo...)
  - **ATTENZIONE:** se la lista non è ordinata, si ottengono risposte, in generale, errate! In particolare, si possono avere "falsi negativi"

```
from bisect import bisect
target = ... # elemento cercato
lst = [...] # lista di elementi ordinabili
lst.sort() # lista ordinata in senso crescente
i = bisect(lst, target)
# i è l'indice (non negativo) della posizione in cui
# andrebbe inserito target in lst per lasciarla in ordine
# cioè l'indice che si potrebbe fornire al metodo insert
# per inserire target in lst, preservandone l'ordinamento;
# se in lst sono già presenti valori uguali a target
# (ovviamente tra loro consecutivi), la funzione bisect
# restituisce il valore massimo tra quelli che
# soddisfano il requisito specificato
if i != 0 and lst[i-1] == target : # analizzare la
    ... # target trovato           # condizione
else :
    ... # target non trovato
```

# Uso del modulo `bisect`

- ❑ Come visto, la funzione **bisect** del modulo **bisect** consente di fare ricerche per bisezione in modo "indiretto"
  - L'obiettivo della funzione è, in realtà, quello di trovare la posizione di inserimento di un nuovo elemento in una lista ordinata in modo che questa continui a essere ordinata
- ❑ È più comodo definire una funzione di ricerca per bisezione che invochi **bisect**

```
from bisect import bisect
## Ricerca per bisezione in lista ordinata.
# @param a lista ordinata in cui cercare
# @param trgt elemento da cercare
# @return (trgt in a)
def binarySearch(a, trgt) :
    i = bisect(a, trgt)
    return i != 0 and a[i-1] == trgt

target = ... # elemento cercato
lst = [...] # lista di elementi ordinabili
lst.sort() # lista ordinata in senso crescente
if binarySearch(lst, target) :
    ... # target trovato
else :
    ... # target non trovato
```

**ATTENZIONE:** se la lista non è ordinata, la funzione fornisce risposte, in generale, errate

# Ordinamento di dati

# Ordinamento: Motivazioni

- ❑ Un problema davvero molto frequente nella elaborazione dei dati presenti in una sequenza consiste nell'*ordinamento* dei dati stessi, secondo un criterio prestabilito
  - numeri in ordine crescente (o decrescente),  
stringhe in ordine lessicografico (crescente o decrescente),  
studenti in ordine di matricola (crescente o decrescente),  
oggetti qualsiasi in ordine qualsiasi...
- ❑ Come abbiamo visto, su dati ordinati è possibile *fare ricerche in modo molto più efficiente* rispetto a quanto si può fare su dati non ordinati
  - Usando l'algoritmo di **ricerca per bisezione** anziché l'algoritmo di **ricerca sequenziale**
- ❑ **Analizzeremo** diversi algoritmi per l'ordinamento di sequenze, introducendo anche *un metodo analitico per la valutazione (approssimata) delle loro prestazioni temporali*

# Importanza delle prestazioni: ordinamento

| Dimensione della lista |  MergeSort<br>(log-lineare) |  SelectionSort<br>(quadratico) |
|------------------------|--|---|
| $10^5$                 | 7 ms   | 5 s   |
| $10^6$                 | 84 ms  | 9 m   |
| $3 \cdot 10^6$         | 270 ms   | 1 h   |
| $5 \cdot 10^6$         | 470 ms   | 3 h   |
| $10 \cdot 10^6$        | 1 s  | 15 h  |
| $20 \cdot 10^6$        | 2 s  | 2 d   |
| $50 \cdot 10^6$        | 5 s  | 15 d  |
| $100 \cdot 10^6$       | 11 s   | 62 d  |
| $200 \cdot 10^6$       | 23 s   | 250 d   |
| $500 \cdot 10^6$       | 1 m  | 4 y   |
| $10^9$                 | 2 m  | 17 y  |



# Ordinamento sul posto oppure no

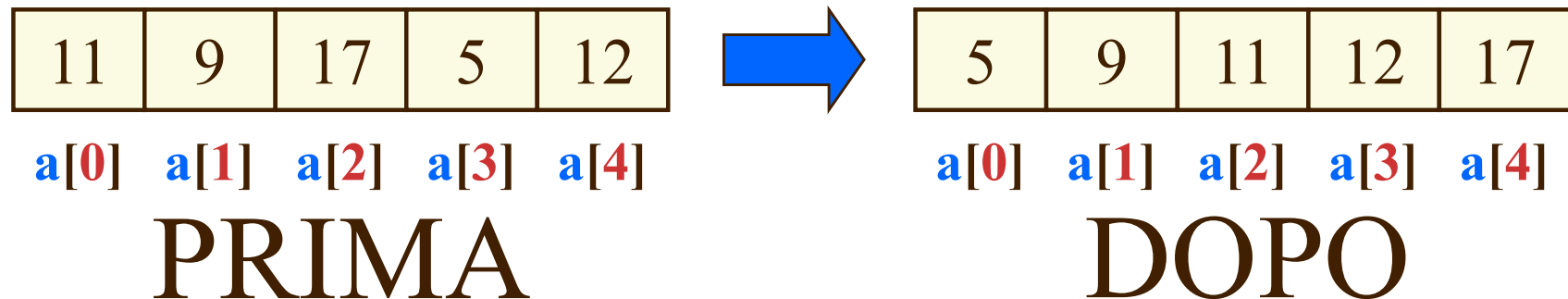
- ❑ Nei problemi di ordinamento di sequenze si parla di
  - Ordinamento “**sul posto**” quando la funzione di ordinamento riceve una sequenza e ne ordina il contenuto
    - Quindi, solitamente, non restituisce nulla:  
al termine dell'esecuzione della funzione di ordinamento, il codice invocante si ritrova il contenitore ordinato
      - Come il metodo **sort** applicabile a liste
  - Ordinamento “**non sul posto**” quando la funzione di ordinamento riceve una sequenza e ne restituisce un'altra contenente gli stessi elementi, ma in ordine; il contenitore ricevuto non viene modificato
    - Come la funzione **sorted**
- ❑ Gli algoritmi sono simili, vedremo solo quelli “sul posto”

# Ordinamento per selezione (SelectionSort)

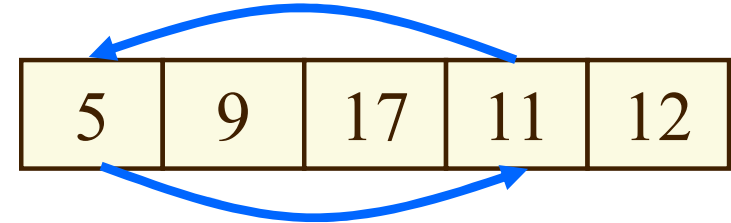
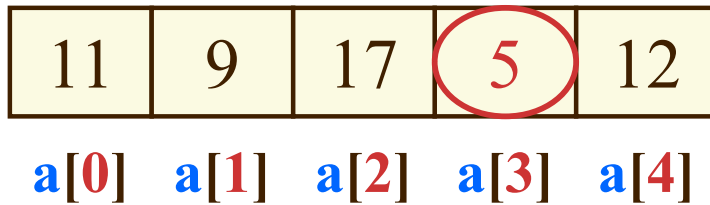
**ATTENZIONE:** non viene qui presentato il modo migliore per risolvere il problema dell'ordinamento, semplicemente analizziamo uno dei tanti algoritmi che risolvono questo problema, uno di quelli più semplici dal punto di vista didattico

# Ordinamento per selezione

- ❑ Per semplicità, analizzeremo l'algoritmo nella versione che *ordina numeri* memorizzati in una lista
  - si estende banalmente al caso in cui si debbano elaborare altri tipi di dati "confrontabili", nella cui classe sia definito il metodo \_\_lt\_\_
- ❑ Prendiamo in esame una lista **a** da ordinare *in senso crescente* (meglio, in senso “non decrescente”, per la potenziale presenza di valori duplicati)
  - Cioè al crescere dell'indice della posizione, dovrà crescere (meglio, non decrescere) il valore in essa memorizzato



# Ordinamento per selezione



- ❑ Per prima cosa, bisogna trovare *la posizione dell'elemento minimo presente nell'intera lista*, come sappiamo già fare
  - in questo caso l'elemento minimo è **5** in posizione **3**
- ❑ Essendo l'elemento minimo, la sua *posizione finale corretta* nella lista ordinata è **0**
  - in  $a[0]$  è però memorizzato il numero **11**, da spostare
  - non sappiamo quale sarà la posizione finale di **11**
    - lo spostiamo temporaneamente in  $a[3]$
  - quindi, *scambiamo*  $a[3]$  con  $a[0]$

# Ordinamento per selezione

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 17 | 11 | 12 |
|---|---|----|----|----|

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$

la parte colorata della lista è  
già ordinata

- ❑ La parte sinistra della lista è già ordinata e non sarà più considerata, dobbiamo ordinare la parte destra
- ❑ Ordiniamo la parte destra *con lo stesso algoritmo*
  - cerchiamo l'elemento minimo nella porzione non colorata: è 9 in posizione 1
  - dato che è già nella prima posizione della parte da ordinare, *non c'è bisogno di fare scambi*
  - **semplicemente, la porzione ordinata cresce...**

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 17 | 11 | 12 |
|---|---|----|----|----|

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$

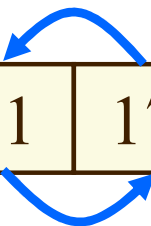
# Ordinamento per selezione

|        |        |        |        |        |
|--------|--------|--------|--------|--------|
| 5      | 9      | 17     | 11     | 12     |
| $a[0]$ | $a[1]$ | $a[2]$ | $a[3]$ | $a[4]$ |

□ Proseguiamo per ordinare la parte di lista che contiene gli elementi  $a[2]$ ,  $a[3]$  e  $a[4]$

- l'elemento minimo è il numero 11 in posizione 3
- scambiamo  $a[3]$  con  $a[2]$

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 11 | 17 | 12 |
|---|---|----|----|----|



|        |        |        |        |        |
|--------|--------|--------|--------|--------|
| 5      | 9      | 11     | 17     | 12     |
| $a[0]$ | $a[1]$ | $a[2]$ | $a[3]$ | $a[4]$ |

# Ordinamento per selezione

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 11 | 17 | 12 |
|---|---|----|----|----|

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$

□ Ora la lista da ordinare contiene  $a[3]$  e  $a[4]$

- l'elemento minimo è il numero 12 in posizione 4
- scambiamo  $a[4]$  con  $a[3]$

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 11 | 12 | 17 |
|---|---|----|----|----|

|   |   |    |    |    |
|---|---|----|----|----|
| 5 | 9 | 11 | 12 | 17 |
|---|---|----|----|----|

$a[0]$   $a[1]$   $a[2]$   $a[3]$   $a[4]$

□ A questo punto *la parte da ordinare contiene un solo elemento*, quindi *è ovviamente ordinata*

# Ordinamento per selezione

```
def selectionSort(a) :  
    for i in range(len(a) - 1) : # tranne l'ultimo  
        # cerco l'elemento che deve andare in posizione i  
        minPos = findMinPosFrom(a, i)  
        # l'elemento cercato si trova in posizione minPos  
        if minPos != i : # se non metto if e lascio il corpo?  
            swap(a, minPos, i) # effettuo lo scambio necessario  
# funziona anche nei casi degeneri, len(a) uguale a 0 e 1,  
# nei quali non c'è niente da fare... e non fa niente!  
  
def swap(a, i, j) : # ok anche se i è uguale a j: non fa  
    temp = a[i]      # niente di utile, ma non fa errori  
    a[i] = a[j]  
    a[j] = temp  
  
def findMinPosFrom(a, frm) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if a[i] < a[pos] :  
            pos = i  
        i += 1  
    return pos
```

Per ordinare liste di **altri tipi di dati** non c'è bisogno di alcuna modifica!  
È sufficiente che con tali dati funzioni l'operatore **<**, cioè sia definito il metodo **\_\_lt\_\_**



# Funzioni ausiliarie

- ❑ Senza le funzioni ausiliarie il codice è meno comprensibile

```
def selectionSort(a) :  
    for i in range(len(a) - 1) :  
        minPos = i  
        j = i + 1  
        while j < len(a) :  
            if a[j] < a[minPos] :  
                minPos = j  
            j += 1  
        if minPos != i :  
            temp = a[i]  
            a[i] = a[minPos]  
            a[minPos] = temp
```

- ❑ Esegue la stessa elaborazione di prima

# Funzioni ausiliarie

- ❑ Senza le funzioni ausiliarie il codice è meno comprensibile

```
def selectionSort(a) :  
    for i in range(len(a) - 1) :  
        minPos = i  
        j = i + 1  
        while j < len(a) :  
            if a[j] < a[minPos] :  
                minPos = j  
            j += 1          # findMinPosFrom(...)  
        if minPos != i :  
            temp = a[i]  
            a[i] = a[minPos]  
            a[minPos] = temp      # swap(...)
```

- ❑ Con le funzioni ausiliarie, se si usano **nomi appropriati (come è obbligatorio fare!)**, il codice “si commenta da solo”

# Ordinamento decrescente

- ❑ Per ordinare, invece, in senso decrescente, è sufficiente
  - Definire e usare una funzione **findMaxPosFrom**, che identifichi di volta in volta la posizione dell'elemento **massimo**
  - In tale funzione, usare il confronto **>** anziché **<**

```
def findMaxPosFrom(a, frm) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if a[i] > a[pos] :  
            pos = i  
        i += 1  
    return pos
```

# Ordinamento **decrescente** per selezione

```
def reverseSelectionSort(a) :  
    for i in range(len(a) - 1) :  
        maxPos = findMaxPosFrom(a, i)  
        if maxPos != i :  
            swap(a, maxPos, i)  
  
def swap(a, i, j) :  
    temp = a[i]  
    a[i] = a[j]  
    a[j] = temp  
  
def findMaxPosFrom(a, frm) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if a[i] > a[pos] : # direzione cambiata!  
            pos = i  
        i += 1  
    return pos
```

# Usiamo il parametro `reverse`

IN AUTONOMIA

```
def selectionSort(a, reverse=False) :  
    for i in range(len(a) - 1) :  
        pos = findMinOrMaxPosFrom(a, i, reverse)  
        if pos != i :  
            swap(a, pos, i)
```

```
def swap(a, i, j) :  
    temp = a[i]  
    a[i] = a[j]  
    a[j] = temp
```

# con findMax=False trova il minimo

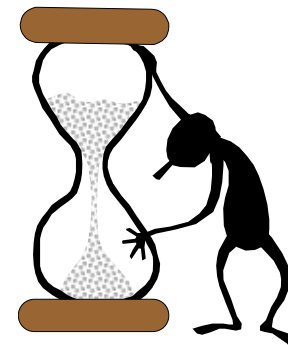
```
def findMinOrMaxPosFrom(a, frm, findMax) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if (findMax and a[i] > a[pos]) \  
            or (not findMax and a[i] < a[pos]) :  
            pos = i  
        i += 1  
    return pos
```

Tecnica: parametro con valore predefinito! Se nell'invocazione il parametro manca, non è un errore: viene usato il suo valore predefinito.  
Come **end** in **print**  
o **reverse** in **sorted**

**Lezione 37**  
**10/01/2025**  
**ore 16.30-18.30**  
**aula Ve**

# Ordinamento per selezione

- ❑ L'algoritmo di ordinamento per selezione è corretto ed è in grado di ordinare qualsiasi lista
  - allora perché si studiano altri algoritmi di ordinamento?
  - *a cosa serve avere diversi algoritmi per risolvere lo stesso problema?*
- ❑ *Esistono algoritmi di ordinamento che, a parità di dimensione della lista da ordinare, vengono eseguiti più velocemente*



# Altri fattori nelle prestazioni?

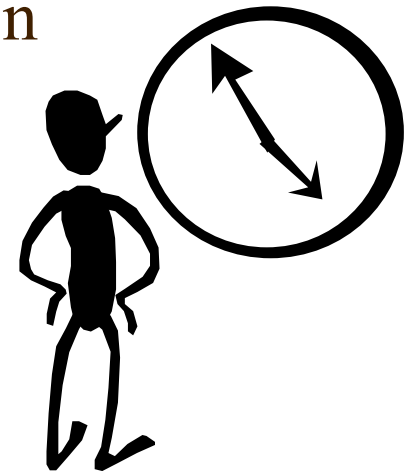
- ❑ Ci occupiamo del **tempo di esecuzione** degli algoritmi di ordinamento ma, in generale, tra le prestazioni degli algoritmi interessa anche l'**occupazione di memoria** durante l'esecuzione
  - Eventualmente anche il "consumo di energia"... soprattutto per i dispositivi a batteria
- ❑ Gli algoritmi di ordinamento **sul posto** spesso non usano memoria aggiuntiva rispetto alla lista da ordinare
  - Usano soltanto alcune variabili locali, ma in numero che, all'aumentare della dimensione della lista da ordinare, diviene rapidamente trascurabile rispetto allo spazio occupato dalla lista stessa
    - Un numero **fisso** di variabili, che non dipende dalla dimensione della lista da ordinare
- ❑ Quindi, solitamente **negli algoritmi di ordinamento interessa principalmente il tempo di esecuzione**



# Rilevazione delle prestazioni

# Rilevazione delle prestazioni

- ❑ Per valutare l'efficienza temporale di un algoritmo si misura il tempo impiegato per la sua esecuzione su insiemi di dati di dimensioni diverse, per poi magari fare un grafico
- ❑ Il tempo *non va misurato con un cronometro*, perché alcune componenti del tempo reale di esecuzione non dipendono dall'algoritmo stesso
  - caricamento in memoria dell'interprete da parte del S.O.
  - caricamento dei moduli del programma
  - lettura dei dati dallo standard input
  - visualizzazione dei risultati
- ❑ Tali componenti, poi, sono anche variabili nel tempo, da un'esecuzione all'altra, per effetto di altri programmi in esecuzione sul computer (bisognerebbe almeno calcolare un tempo di esecuzione medio)



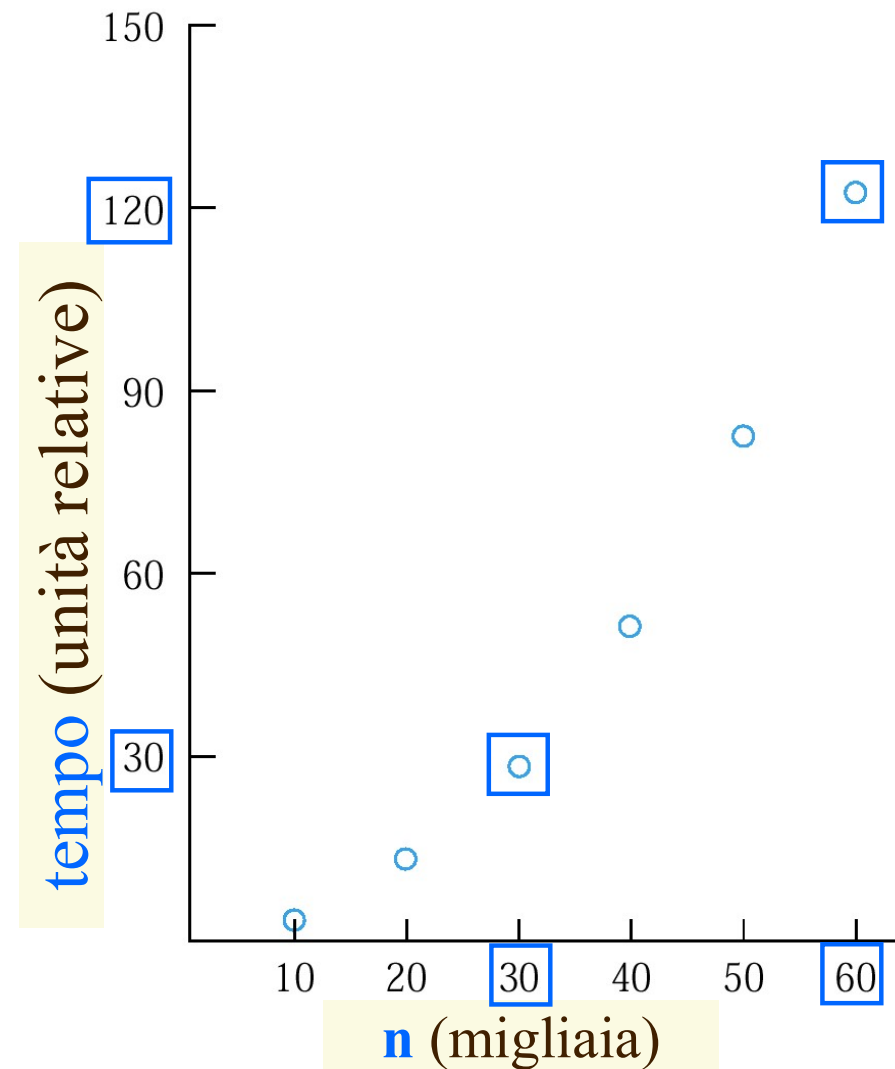
# Rilevazione delle prestazioni

- ❑ Il tempo di esecuzione di un algoritmo va misurato **all'interno del programma stesso**
- ❑ Si può usare la funzione `time` del modulo `time` che, a ogni invocazione, restituisce un valore di tipo `float` che rappresenta...
  - il numero di secondi trascorsi da un evento di riferimento (*la mezzanotte del 1 gennaio 1970*) !!
  - Ciò che interessa è la *differenza* tra due valori
    - si invoca la funzione *prima e dopo* l'esecuzione dell'algoritmo (escludendo anche le operazioni di input/output dei dati)

```
from time import time
a = list()
# fase di input dei dati
...
beginTime = time()
selectionSort(a)
elapsedTime = time() - beginTime
print("%f secondi" % elapsedTime)
# fase di output dei dati
...
```

# Rilevazione delle prestazioni

- Proviamo a eseguire l'ordinamento di liste di diverse dimensioni (**n**), contenenti numeri casuali
- Usando l'algoritmo di **ordinamento per selezione**, si nota che l'andamento del tempo di esecuzione **non è lineare**
  - *se **n** diventa il doppio, il tempo diventa circa il quadruplo!*




# Rilevazione delle prestazioni

- ❑ Le prestazioni dell'algoritmo di ordinamento per selezione hanno quindi un *andamento quadratico* (o *parabolico*) in funzione delle dimensioni della lista
- ❑ Le prossime domande che ci poniamo sono
  - *è possibile valutare le prestazioni di un algoritmo dal punto di vista teorico?*
    - cioè **senza** eseguire molte volte un programma al solo fine di rilevarne i tempi di esecuzione e osservarne l'andamento?
  - *esiste un algoritmo più efficiente per l'ordinamento di una lista?*


# Analisi teorica delle prestazioni

# Analisi teorica delle prestazioni

 □ Il tempo d'esecuzione di un algoritmo dipende, ovviamente, dal numero e dal tipo di istruzioni in *linguaggio macchina* eseguite dal processore

□ Per fare un'analisi teorica  
senza fare esperimenti di esecuzione

dobbiamo fare delle *semplificazioni (apparentemente) drastiche*

 ■ *nell'esecuzione del codice dell'algoritmo, contiamo soltanto gli accessi in lettura o scrittura a singoli elementi della lista*, ipotizzando che tali accessi siano le operazioni elementari più lente durante l'esecuzione del programma e che tutte le altre attività richiedano un tempo trascurabile

■ Questo, ovviamente, è **un modello** (computazionale), la cui validità deve essere sottoposta a verifica!

- *Se con questo modello riusciremo a fare previsioni realistiche della realtà, sarà un modello valido, altrimenti no!*

- **Si osserva sperimentalmente che è un modello valido per algoritmi che non modificano la lunghezza della lista**

# Analisi teorica delle prestazioni

- ❑ Rivediamo il codice di **selectionSort**: gli accessi alle celle della lista avvengono nelle funzioni **swap** e **findMinPosFrom**

```
def swap(a, i, j) :  
    temp = a[i]  
    a[i] = a[j]  
    a[j] = temp
```

- ❑ È possibile migliorare il codice della funzione **swap** in modo che effettui un minor numero di accessi?
- ❑ Una singola invocazione della funzione **swap**, che scambia il contenuto di due celle, effettua **4 accessi** (due in lettura e due in scrittura) e **non è possibile fare uno scambio con un minor numero di accessi: certamente bisogna leggere entrambe le celle e bisogna anche scriverle entrambe**
- ❑ E cosa possiamo dire per la funzione **findMinPosFrom**?



# Analisi teorica delle prestazioni

- ❑ Quanti accessi effettua ciascuna invocazione della funzione **findMinPosFrom**?

```
def findMinPosFrom(a, frm) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if a[i] < a[pos] :  
            pos = i  
        i += 1  
    return pos
```

- ❑ Naturalmente, il numero di accessi dipende dalla **dimensione della porzione di lista da esaminare**, che è uguale a **(len(a) - frm)**
  - **Si può dimostrare** che, per trovare il valore minimo (o massimo) in una sequenza **non ordinata** avente  $k$  elementi, è necessario (e sufficiente) leggere ciascun elemento una e una sola volta, cioè fare  $k$  accessi
  - Senza fare questo (peraltro utile) esercizio, immaginiamo di aver riscritto la funzione **findMinPosFrom** in modo che sia ottima, cioè faccia il numero minimo di accessi [codice nelle slide successive]
    - $k$  accessi quando elabora una porzione di lista di lunghezza  $k$

- ❑ Quanti accessi effettua la funzione **findMinPosFrom**?

```
def findMinPosFrom(a, frm) :  
    pos = frm  
    i = pos + 1  
    while i < len(a) :  
        if a[i] < a[pos] :  
            pos = i  
        i += 1  
    return pos
```


- ❑ Nel conteggio occorre ovviamente contare anche eventuali accessi effettuati all'interno di metodi/funzioni **invocati** dalla funzione in esame

- ❑ In questo caso viene (ripetutamente) invocata la funzione **len**
- Non sappiamo come avviene l'elaborazione al suo interno, ma la documentazione della classe **list** ci dice che gli oggetti di tipo lista hanno una variabile di esemplare che ne memorizza la lunghezza, quindi l'esecuzione di **len** ispeziona semplicemente tale variabile: **non fa accessi agli elementi della lista per contarli!**
  - Analogamente per stringhe, tuple, dizionari, insiemi

❑ Progettiamo diversamente la funzione **findMinPosFrom**, in modo che effettui un minor numero di accessi

- Come vedremo, spesso ci si trova di fronte a un **compromesso tra tempo d'esecuzione e occupazione di memoria**: per ridurre il tempo, si può decidere di occupare più memoria (o viceversa)

```
def findMinPosFrom(a, frm) :
    pos = frm
    i = pos + 1
    while i < len(a) :
        if a[i] < a[pos] :
            pos = i
        i += 1
    return pos
```



```
def findMinPosFrom(a, frm) :
    pos = frm
    currentMin = a[pos] # uso una variabile in più
    i = pos + 1
    while i < len(a) :
        if a[i] < currentMin :
            pos = i
            currentMin = a[i]
        i += 1
    return pos
```

Osservo che **a[pos]** viene potenzialmente riletta molte volte, sempre uguale finché il valore di **pos** non cambia...

```
""" in questo modo quando faccio i confronti
    non ho bisogno di (ri)leggere a[pos]
    ma per aggiornare currentMin rileggo a[i]
    Si può eliminare? Aggiungo un'altra variabile
    """
```

```
def findMinPosFrom(a, frm) :  
    pos = frm  
    currentMin = a[pos]  
    i = pos + 1  
    while i < len(a) :  
        maybeNewMin = a[i] # uso un'altra variabile in più  
        if maybeNewMin < currentMin :  
            pos = i  
            currentMin = maybeNewMin  
        i += 1  
    return pos  
    """ ora ogni cella viene letta una e una sola  
        volta, certamente per trovare il minimo  
        non si può far meglio, devo leggere tutte le  
        celle della lista almeno una volta!  
    """
```

- ❑ Se la porzione di lista da esaminare ha  $k$  celle
  - Questa funzione effettua esattamente  $k$  accessi e non è possibile calcolare il valore minimo tra  $k$  valori senza leggerli tutti! È una funzione **ottima**
  - La versione originale effettuava  $2k - 2$  accessi, **quasi il doppio**

# Analisi teorica delle prestazioni

- ❑ Ordiniamo con la selezione una lista di  $n$  elementi
  - ❑ Vediamo la **prima iterazione del ciclo**
    - per trovare l'elemento minimo si fanno  $n$  accessi
    - per scambiare due elementi si fanno **quattro** accessi
      - trascuriamo il caso in cui non serva lo scambio, non ci interessano i casi “fortunati”... valutiamo le prestazioni “**nel caso pessimo**”
    - in totale,  $(n+4)$  accessi
  - ❑ A questo punto dobbiamo ordinare la parte rimanente, cioè una lista di  $(n-1)$  elementi
    - serviranno quindi  $(n-1) + 4$  accessi
- e via così fino al passo con  $n=2$ , incluso, che richiede  $(2+4)$  accessi

# Analisi teorica delle prestazioni

- Il conteggio totale  $T(n)$  degli accessi in lettura o scrittura è

$$T(n) = (n+4) + ((n-1)+4) + \dots + (3+4) + (2+4)$$

$$= n + (n-1) + \dots + 3 + 2 + (n-1) * 4$$

$$\Rightarrow n * (n+1) / 2 - 1 + (n-1) * 4$$

$$= (1/2) * n^2 + (9/2) * n - 5$$

- Si ottiene quindi *un'equazione di secondo grado* in  $n$ , che giustifica l'andamento *parabolico* dei tempi rilevati sperimentalmente (anche se in realtà **questa formula non calcola il tempo, ma il numero di accessi**)

- Il tempo di esecuzione è (approssimativamente) proporzionale al numero di accessi

Formula di Gauss

$$n + (n-1) + \dots + 3 + 2 + 1 = n * (n+1) / 2$$

# Con funzioni di libreria?

- ❑ Ovviamente nel codice di **selectionSort** avremmo potuto usare funzioni di libreria

```
def selectionSort(a) :  
    for i in range(len(a) - 1) :  
        minPos = a.index(min(a[i:]), i)  
        if minPos != i : # scambia  
            (a[i], a[minPos]) = (a[minPos], a[i])
```

- ❑ In questo modo, però, la valutazione delle prestazioni asintotiche sarebbe stata più complessa, perché **avremmo dovuto fare indagini sul codice di funzioni e/o metodi di libreria, oltre che di operatori** (come la creazione di sotto-liste)

# Algoritmi di ordinamento più efficienti

- Esistono algoritmi di ordinamento, come *MergeSort* (che non vedremo), che impiegano un **tempo log-lineare** in funzione della lunghezza  $n$  della lista da ordinare, cioè un tempo proporzionale a una funzione del tipo  $n \log_2 n$ 
  - Come visto nella tabella iniziale, queste sono prestazioni temporali MOLTO migliori rispetto a quelle dell'ordinamento per selezione, che richiede un tempo proporzionale a una funzione quadratica
  - Le funzioni e i metodi di ordinamento di sequenze presenti nella libreria standard usano algoritmi con tali prestazioni log-lineari



# Un caso particolare importante

□ Vediamo un ultimo problema relativo all'ordinamento

- **Inserire un elemento in una lista ordinata, in modo che rimanga ordinata**
  - Si parla di "ordinamento mediante inserimento" o "inserimento in ordine"
  - Utile nelle **situazioni in cui i dati vengono conservati in una lista ordinata** (per poter fare ricerche veloci al suo interno, usando la ricerca per bisezione) e **"ogni tanto" viene aggiunto un dato alla lista**
- Per prima cosa bisogna **trovare la posizione  $x$  di inserimento**, mediante una ricerca per bisezione (ricordando che la lista è ordinata)
  - Si può usare la funzione **bisect** del modulo **bisect** (che, in effetti, è stata progettata per risolvere proprio questo problema)
- Poi, tutti gli elementi della lista, **dalla posizione  $x$  (inclusa) in poi**, vanno spostati nella posizione immediatamente successiva a quella che occupano (aumentando, ovviamente, di un'unità la lunghezza della lista)
  - Si può usare il metodo **insert** che agisce su una lista, fornendo il parametro  $x$  ottenuto al passo precedente
- Alternativa (**meno efficiente**): inserire il nuovo elemento in fondo alla lista e ordinare di nuovo l'intera lista (ignorando il fatto che fosse ordinata)

Il materiale necessario per il  
Laboratorio 12  
termina qui