Corso ITS:

PROGETTISTA E SVILUPPATORE SOFTWARE:

FULL STACK DEVELOPER E CLOUD SPECIALIST

Modulo: **Programmazione in Python**

Docente: *Andrea Ribuoli*

Martedì **1 Aprile** 2025

09:00 - 13:00

13:30 - 16:30

---

if

In [2]:
```
from random import randint
floor = randint(1, 20)
actualFloor = 0
if floor > 13 :
    actualFloor = floor - 1
else :
    actualFloor = floor
print(actualFloor)
```

```
13
```

- ricordarsi dei due punti `:`
- gli enunciati dopo `if` e dopo `else` possono essere multipli
- è l'indentazione a semplificare la clausola `else` multiriga
- qualora non esistano enunciati per la clausola `else`, questa va **omessa**

In [3]:
```
from random import randint
floor = randint(1, 20)
actualFloor = floor
```

```
if floor > 13 :
    actualFloor = floor - 1
print(actualFloor)
```

```
4
```

- l' `if` e l' `else` sono i primi casi di **enunciati composti**
- **intestazione** più **blocco di enunciati**
- in inglese: **header** più **statement block**
- l'intestazione termina con il carattere **due punti**
- un *blocco di enunciati*:
- - inizia nella riga che segue l'intestazione
- - gli enunciati che lo compongono sono **incolonnati a sinistra**
- - gli enunciati che lo compongono sono posizionati **più a destra** rispetto alla intestazione
- - più a destra di **un qualunque numero di spazi**
- Quanto detto non condiziona i **commenti** che si posizionano liberamente

- in **Python** la strutturazione a blocchi del codice è **parte della sintassi**
- in conflitto con le linee guida delle grammatiche **context-free**
- (si pensi alla *minificazione* in JavaScript)
- suggerimenti:
- - evitare duplicazioni nelle diramazioni
- - se opportuno ricorrere alle **espressioni condizionali** `v1 if cond else v2`

In [4]:
```
from random import randint
floor = randint(1, 20)
actualFloor = floor - 1 if floor > 13 else floor
print(actualFloor)
```

```
15
```

- `>` è un **operatore relazionale**
- ha lo stesso significato della notazione matematica
- questo non è vero per tutti gli operatori relazionali (solo per `<` e per `>`)

Python	Matematica	Descrizione
>	**>**	maggiore
>=	**≥**	maggiore o uguale
<	**<**	minore
<=	**≤**	minore o uguale
==	**=**	uguale
!=	**≠**	diverso

- l'**operatore di uguaglianza** (==) è classica fonte di confusione nei neofiti
- viene confuso con il **simbolo di assegnazione** (=)
- esattamente come nel *linguaggio C*
- gli stessi operatori si applicano anche alle **stringhe**

- gli operatori relazionali hanno **precedenza inferiore** a quelli aritmetici
- non ha senso testare la esatta uguaglianza di numeri in virgola mobile (**floating**)
- ordinamento **lessicografico** di stringhe
- ci sono particolarità:
- - le **maiuscole ** precedono le **minuscole**
- - lo spazio ( ) **precede tutti** i caratteri visualizzabili
- le **cifre** precedono le lettere
- l'ordine dei segni di punteggiatura è particolare
- se due stringhe sono uguali fino all'ultimo carattere della più corta, la più lunga è **maggiore**

- enunciati **annidati** (*nested*): un `if` all'interno di un altro `if`

---

imposta.py

dal sito della *Agenzia delle Entrate* leggiamo come l'imposta lorda si calcoli applicando al reddito complessivo, al netto degli oneri deducibili, le aliquote per scaglioni...

Realizzare un programma che chieda a video il reddito complessivo, al netto degli oneri deducibili, e restituisca l'imposta lorda.

- **eseguire a mano** come test tenendo traccia su carta dei risultati intermedi e finali (*hand-tracing*)

- costrutto **elif**
- diagrammi di flusso (*flowchart*)
- regola: *non far mai entrare una freccia all'interno di una diversa diramazione
- passare da diagramma di flusso a pseudocodice al crescere delle dimensioni
- collaudo: **coverage**
- copertura di tutti i punti di decisione dell'algoritmo implementato
- si predispone un elenco dei casi di prova necessari e dei risultati previsti
- - ogni diramazione abbia un caso di prova
- - inserire un caso di prova per ogni valore limite

- - progettare i casi di prova **prima** di scrivere il codice
- piano di lavoro

- esigenza: valutare una condizione logica in un punto per poi utilizzarla altrove
- risposta: variabili **booleane**
- tipo dato: **bool**
- solamente **due** valori: **True** e **False**
- non sono stringhe !!
- inizializzione (`in_errore = False` o `in_errore = True`)
- successivamente: `if failed :`
- operatori **booleani**: `and` , `or` , `not`
- considerazioni:
- - confondere *and* e *or* non è infrequente
- - leggibilità: mai confronti diretti con *True* e *False*
- - concatenazione di operatori relazionali (possibile in Python ma ...)
- - valutazione **in cortocircuito** degli operatori *and* e *or*
- - le leggi di *De Morgan* (**not** applicato a espressioni **and** o **or**)

- operatore `in` (e la sua negazione `not in`)
- metodi `startswith e endswith`
- metodo `count`
- metodo `find`
- metodi `isalpha` , `isdigit and isalnum`
- metodi `islower and isupper`
- metodo `isspace`
- considerazioni:
- - sono operatore e metodi utili nella **input validation**
- - rischio di **eccezioni** (*run-time exception*)

- terminare un programma con la funzione `exit`

---

il ciclo `while`

- è il primo degli **enunciati di ciclo**
- esegue ripetutamente istruzioni fino al raggiungimento di un obiettivo

```
while condizione :
    enunciato_1
    enunciato_2
```

- fintanto che la condizione è vera gli enunciati presenti all'interno vengono eseguiti
- gli enunciati interni al `while` ne costituiscono il **corpo** (*body*)
- l'enunciato **while** è un esempio di **ciclo** (*loop*)
- cicli controllati da **contatore** (*definito*) o da **evento** (*indefinito*)
- considerazioni:
- - *Abbiamo finito?* .vs. *Fino a quando?*
- - cicli infiniti
- - errori per *scarto di uno*

- **named argument** (*argomento con nome*): `print con end=""`
- cicli per acquisire valori in sequenza
- valore **sentinella** per segnalare la fine di una sequenza
- esercizio: sviluppo del programma `media.py`
- utilizzo di `break`
- redirezione **I/O** (*input/output*)

Il ciclo **for**

```
nomeRegione = "Marche"
for letter in nomeRegione :
   print(letter)
nomeRegione = "Marche"
i = 0
while i < len(nomeRegione) :
   letter = nomeRegione[i]
   print(letter)
   i = i + 1
```

- il ciclo `for` può essere usato su **qualunque contenitore**
- funzione `range()` (crea una sequenza di numeri utilizzabile come contenitori)
- **cicli annidati** (*nested loop*)
- es: trovare la prima o l'ultima corrispondenza di una cifra in una stringa

---

funzione

- una **funzione** è una sequenza di istruzioni datata di un **nome**
- una funzione viene **invocata**(*called*)
- una funzione **restituisce**(*returns*)
- una funzione viene chiamata con *dati di ingresso* detti **argomenti**
- una funzione può **restituire** un valore (*returned value*)

- in fase di definizione di una **funzione utente** è necessario:
- - scegliere un **nome**
- - definire una variabile per ciascuno dei suoi *argomenti* (detti **variabili parametro**)

In [2]:
```
def volumeCubo(lunghezzaLato) :
    volume = lunghezzaLato ** 3
    return volume
```

- gli enunciati nella definizione di funzione non vengono eseguiti
- gli enunciati non interni a definizioni di funzione vengono eseguiti nell'ordine
- è importante che ciascuna funzione sia **definita prima di essere invocata**
- ambito di visibilità di una variabile (**variable scope**)
- **variabile locale** (*local variable*)
- **variabile globale** (*global variable*)
- suggerimenti:
- - evitare l'uso di variabili globali

In [4]:
```
DIFFERENZA = 127397
sigla = "IT"
flag  = ""
for lettera in sigla :
    flag += chr(ord(lettera) + DIFFERENZA)
print(flag)
```

In [5]:
```
DIFFERENZA = 127397
sigla = "US"
flag  = ""
for lettera in sigla :
    flag += chr(ord(lettera) + DIFFERENZA)
print(flag)
```