Dispensa · Modulo 2

1. Che cosa cambia nel modulo 2

Il primo modulo introduceva il lessico minimo della programmazione: tipi fondamentali, operazioni, REPL, differenza tra script e modalità interattiva. Il secondo modulo inizia a trasformare quel lessico in programmi veri e propri. Le espressioni non bastano più: oltre a calcolare valori, bisogna saperli conservare, riusare, confrontare, leggere dall'esterno e seguire passo per passo l'effetto che hanno sullo stato del programma.

In questo modulo compaiono quindi quattro idee centrali. La prima è la distinzione tra **espressioni** e **istruzioni**. La seconda è il concetto di **variabile**, cioè un nome associato a un valore che può cambiare durante l'esecuzione. La terza è l'uso di **input**, **conversioni di tipo** e **valori booleani** per rendere il programma dipendente dai dati forniti dall'utente. La quarta è un metodo di lavoro: imparare a **tracciare l'esecuzione a mano**, cioè a simulare il comportamento dell'interprete senza usare il computer.

2. Due esercizi iniziali: numeri e stringhe

La parte iniziale del modulo 2 riprende i concetti del modulo 1 con due esercizi guidati. Il primo lavora sui numeri e serve a consolidare divisione intera e modulo. Il secondo lavora sulle stringhe e serve a consolidare indicizzazione e slicing. In entrambi i casi il punto metodologico è importante: gli studenti, a questo stadio del corso, non conoscono ancora il concetto di variabile, quindi il problema viene affrontato usando solo espressioni e operazioni di base.

Convertire secondi in ore, minuti e secondi

Supponiamo di avere un numero di secondi e di volerlo convertire in ore, minuti e secondi residui. Sappiamo, per esempio, che `75` secondi corrispondono a `1` minuto e `15` secondi, che `189` secondi corrispondono a `3` minuti e `9` secondi, e che `3982` secondi corrispondono a `1` ora, `6` minuti e `22` secondi.

La prima domanda è: quale operazione permette di passare da un numero di secondi al numero di minuti completi contenuti al suo interno? In un minuto ci sono `60` secondi, quindi la prima idea naturale è dividere per `60`. Ma la divisione ordinaria `/` in Python restituisce un numero decimale, mentre qui interessa un numero intero. Per questo si usa la **divisione intera**:

```
print("In 75 secondi ci sono", 75 // 60, "minuti")
print("In 189 secondi ci sono", 189 // 60, "minuti")
```

Una volta ottenuti i minuti completi, resta da capire quanti secondi avanzano. Quello che avanza dalla divisione intera è precisamente il **resto**, quindi si usa il modulo `%`:

```
print("In 75 secondi ci sono", 75 // 60, "minuti e", 75 % 60, "secondi")
print("In 189 secondi ci sono", 189 // 60, "minuti e", 189 % 60, "secondi")
```

Per `3982` secondi questo ragionamento non basta più, perché il risultato espresso soltanto in minuti e secondi sarebbe:

```
print("In 3982 secondi ci sono", 3982 // 60, "minuti e", 3982 % 60, "secondi")
```

che produce `66` minuti e `22` secondi. Il calcolo non è sbagliato, ma non è ancora nella forma richiesta, perché `66` minuti devono essere a loro volta convertiti in ore e minuti residui.

Il ragionamento resta però identico: prima si contano i minuti completi dividendo i secondi per `60`, poi si contano le ore complete dividendo i minuti per `60`. Senza introdurre variabili, tutto questo si può esprimere direttamente con tre espressioni:

```
3982 // 60 // 60
3982 // 60 % 60
3982 % 60
```

La prima calcola le ore, la seconda i minuti residui, la terza i secondi residui. In questo caso i risultati sono rispettivamente `1`, `6` e `22`.

Quindi:

- il numero di ore si ottiene con `secondi_totali // 60 // 60`, oppure in modo più compatto `secondi_totali // 3600`;
- il numero di minuti residui si ottiene con `secondi_totali // 60 % 60`;
- il numero di secondi residui si ottiene con `secondi_totali % 60`.

```
print("In 3982 secondi ci sono", 3982//3600, "ore e", 3982 // 60 % 60, "minuti e", 3982 % 60, '
```

Selezionare caratteri da una stringa

Il secondo esercizio chiede, data una stringa e una posizione `n`, di estrarre il carattere in posizione `n` insieme al precedente e al successivo.

Se `s` è una stringa, `s[n]` seleziona il carattere in posizione `n`, mentre `s[n-1:n+2]` seleziona tre caratteri consecutivi: il precedente, quello in posizione `n` e il successivo. La regola decisiva è che nello slicing l'estremo iniziale è incluso e quello finale è escluso. Per questo, se si vogliono le posizioni `9`, `10` e `11`, bisogna scrivere `s[9:12]` e non `s[9:11]`.

```
print("informatica"[2:5])
```

3. Dalle operazioni alle espressioni

Finché si scrivono somme, divisioni, concatenazioni, accessi a caratteri o chiamate a funzioni come `len(...)`, si stanno costruendo **espressioni**. Una espressione è una porzione di codice che, quando viene valutata, **produce un valore**. Quel valore ha sempre un tipo: `3 + 4` produce un `int`, `len("ciao")` produce un `int`, `"ab" + "cd"` produce una `str`.

Il punto importante è che un'espressione non è definita dal tipo di simboli che contiene, ma dal fatto che il suo risultato finale è un valore. Anche espressioni più complesse continuano a comportarsi allo stesso modo:

```
3982 // 60 + 10
len("ciao") + 5
"info" + "rmatica"
```

In tutti questi casi Python legge l'espressione, controlla che la sintassi sia corretta, verifica che le operazioni abbiano senso sui tipi coinvolti e, se tutto va bene, restituisce un valore finale.

Le espressioni, da sole, non bastano a programmare. Possono produrre un valore, ma non permettono di conservarlo per usi futuri, di assegnargli un nome o di usarlo in più punti del programma senza ripetere il calcolo. Per passare dal puro calcolo alla costruzione di programmi bisogna introdurre le **istruzioni**.

Un'istruzione fa compiere un'azione al programma: per esempio salva un valore in memoria, stampa un risultato, controlla una condizione o decide quale blocco eseguire. In altre parole:

- una **espressione** produce un valore;
- una **istruzione** fa eseguire un'azione.

Per esempio:

```
3 + 4
```

è un'espressione, perché produce il valore `7`.

```
x = 3 + 4
```

è un'istruzione di assegnamento, perché salva da qualche parte il risultato dell'espressione `3 + 4`.

Questo permette di chiarire una distinzione già affacciata nel modulo 1. Un'espressione può essere:

- **sintatticamente scorretta**, come `3 +`, perché la struttura è incompleta;
- **sintatticamente corretta ma semanticamente scorretta**, come `3 + "ciao"`, perché l'operazione non ha senso per i tipi usati;
- **corretta**, come `3 + 4` o `len("ciao")`, e quindi valutabile con successo.

4. Struttura di uno script

Con l'introduzione delle variabili e dei costrutti condizionali, gli script diventano più articolati. È utile adottare fin da subito una convenzione per organizzarli in modo leggibile.

Un file `.py` tende a rispettare questa struttura in tre blocchi:

1. intestazione: commenti iniziali con autore, data e scopo del file;
2. eventuali `import` di librerie esterne;
3. definizione di eventuali funzioni, seguita dal corpo principale del programma.

Esempio schematico:

```
# autore, data, scopo dello script

import sys

def capitalizza(s):
    return s[0].upper() + s[1:].lower()

# corpo principale
nome = input("Nome: ")
print(capitalizza(nome))
```

Commenti

Le righe che iniziano con `#` sono **commenti**: vengono ignorate dall'interprete e servono a spiegare la struttura, lo scopo o le scelte del codice. Blocchi di testo descrittivo più lunghi possono comparire anche come stringhe multilinea usate come documentazione (*docstring*).

Usare i commenti è una buona pratica che rende il codice più leggibile, sia per chi lo scrive sia per chi lo rilegge in seguito.

5. Variabili e assegnamento

La prima istruzione con cui ci confrontiamo è l'assegnamento di variabile. Una variabile è un nome che permette di salvare un valore in memoria e riutilizzarlo in seguito. Se si vuole, per esempio, convertire la temperatura di oggi (23 gradi Centigradi) a Kelvin e Fahrenheit, possiamo scrivere una serie di espressioni:

```
print(23 + 273.15)
print(23 * 9 / 5 + 32)
```

Scrivere ogni volta il numero `23` direttamente nelle espressioni è possibile, ma poco leggibile e poco flessibile.

Con una variabile il programma diventa più chiaro:

```
temperatura = 23
kelvin = temperatura + 273.15
fahrenheit = temperatura * 9 / 5 + 32
```

La forma generale dell'assegnamento è:

```
nome_variabile = espressione
```

Questo punto richiede attenzione, perché l'operatore `=` in Python **non** ha il significato dell'uguale matematico. In matematica `x = 3` e `3 = x` esprimono la stessa relazione; in Python no. A sinistra dell'assegnamento deve esserci un nome valido di variabile, a destra un valore o un'espressione da valutare. Per questo:

```
x = 3
```

è corretto, mentre:

```
3 = x
```

produce un errore di sintassi.

Semantica dell'assegnamento

Quando Python incontra un'istruzione come:

```
somma = 4 + 3
```

prima valuta l'espressione a destra, ottiene `7`, poi associa il nome `somma` a quel valore. Possiamo pensare alle variabili come a dei cassetti o a set barattoli: il nome è l'etichetta, il valore è il contenuto. L'assegnamento crea o aggiorna un barattolo con un'etichetta precisa.

Questo spiega anche perché una variabile può essere usata dentro altre espressioni:

```
y = 3 + 1
x = y + 2
```

Alla seconda riga Python non usa la lettera `y` come simbolo astratto: legge il valore che in quel momento è associato a `y`, il valore che è contenuto nel barattolo con etichetta `y`, cioè `4`, e calcola `4 + 2`.

Riassegnazione

La conseguenza più importante di questo modello è che una variabile può cambiare valore:

```
x = 15
x = x + 1
```

La seconda riga viene interpretata in una sequenza di tre passi: leggi il valore attuale di `x`, aggiungi `1`, salva di nuovo il risultato in `x`. Alla fine `x` vale `16`.

6. Nomi validi e stato del programma

Poiché Python è un linguaggio formale, anche i nomi delle variabili devono rispettare regole precise. Un nome valido deve iniziare con una lettera o con underscore; dal secondo carattere in poi può contenere lettere, cifre e underscore. Non può contenere spazi, non può iniziare con una cifra e non può coincidere con una parola chiave del linguaggio, come `if`.

Nome	Valido?	Motivo
`temperatura`	sì	inizia con lettera
`temperatura_2`	sì	contiene underscore e cifra
`_contatore`	sì	underscore iniziale ammesso
`2gradi`	no	inizia con cifra
`mio nome`	no	contiene spazio
`if`	no	è una parola chiave

Dal punto di vista strettamente tecnico, Python moderno accetta anche caratteri non ASCII in alcuni identificatori (possiamo chiamare una variabile `età`); tuttavia, come buona pratica, conviene evitarli per ragioni di leggibilità e compatibilità (meglio continuare a chiamarla `eta`).

Lo stato del programma: differenza tra codice sorgente ed esecuzione

L'introduzione delle variabili rende necessario distinguere con precisione due cose che, all'inizio, tendono a confondersi: il **codice sorgente** e lo **stato del programma**.

Il **codice sorgente** è il testo scritto nel file `.py`. Rimane uguale finché non lo modifichiamo: è la sequenza di istruzioni che abbiamo deciso di scrivere. Lo **stato del programma**, invece, non è testo ma memoria: è l'insieme dei nomi e dei valori che esistono in un certo istante durante l'esecuzione.

Questa distinzione è fondamentale perché il programma scritto nel file non coincide con ciò che il computer "sa" in ogni momento. Il file può essere sempre lo stesso, ma lo stato può cambiare continuamente mentre l'interprete esegue le istruzioni una dopo l'altra.

Per esempio, nel codice:

```
x = 3
y = x + 2
x = x + 1
```

il codice sorgente resta fisso ed è sempre composto da queste tre righe. Lo stato, invece, evolve così:

1. dopo `x = 3`, lo stato contiene `x = 3`;
2. dopo `y = x + 2`, lo stato contiene `x = 3` e `y = 5`;
3. dopo `x = x + 1`, lo stato contiene `x = 4` e `y = 5`.

Questo mostra che lo stato è qualcosa che appartiene all'**esecuzione**, non al file in sé. È il motivo per cui due esecuzioni dello stesso programma possono produrre risultati diversi: possono cambiare i valori letti in input, le riassegnazioni effettuate o i rami condizionali eseguiti.

Capire questa differenza è essenziale per leggere correttamente i programmi. Quando si ragiona sul codice, non basta guardare "che cosa c'è scritto": bisogna chiedersi anche **che cosa esiste in memoria in quel momento** e **quale istruzione è già stata eseguita**.

Qui emerge un'altra regola importante: **l'ordine di esecuzione conta**. Se si prova a usare una variabile prima di averle assegnato un valore, Python si ferma con un errore:

```
print(eta)
eta = input("Quanti anni hai? ")
```

Al momento della prima riga, `eta` non esiste ancora nello stato del programma.

7. Input, ordine di esecuzione e conversioni di tipo

Finché i valori vengono scritti direttamente nel codice, ogni nuova esecuzione che richiede dati diversi costringe a modificare il file. Per evitare questo problema in questi primi esercizi usiamo la funzione `input()`, che permette di leggere un valore dal terminale durante l'esecuzione:

```
nome = input("Come ti chiami? ")
print("Il tuo nome è", nome)
```

Quando Python esegue la riga con `input()`, mostra il messaggio, attende che l'utente scriva qualcosa e salva il valore inserito nella variabile corrispondente.

`input()` restituisce sempre una stringa

Proviamo a eseguire questo codice:

```
eta = input("Eta: ")
print(eta + 1)
```

Questo codice genera un `TypeError`. Il motivo non è che il numero inserito "non sia valido", ma che `input()` restituisce sempre una **stringa**, anche quando l'utente digita qualcosa che sembra un numero. Se, per esempio, l'utente scrive `23`, Python salva nella variabile `eta` la stringa `"23"`, non l'intero `23`.

Quando l'interprete arriva alla riga:

```
print(eta + 1)
```

prova quindi a eseguire un'operazione tra oggetti di tipo diverso:

- `eta`, che è una stringa;
- `1`, che è un intero.

Questa somma non è definita, e per questo Python segnala un errore di tipo.

La soluzione è usare una **conversione di tipo** esplicita, o **cast**:

```
eta = input("Eta: ")
eta = int(eta)
print(eta + 1)
```

Le conversioni più usate in questa fase sono:

Cast	Significato	Esempio	Risultato
`int()`	converte in intero, se possibile	`int("12")`	`12`
`float()`	converte in numero decimale, se possibile	`float("3.5")`	`3.5`
`str()`	converte in stringa	`str(12)`	`"12"`

8. Tracciare l'esecuzione passo per passo

Una delle competenze metodologiche più importanti introdotte nel modulo 2 è la simulazione manuale del codice. L'obiettivo non è "indovinare" il comportamento del programma, ma ricostruirlo con precisione, come farebbe l'interprete. Questo tipo di esercizio è particolarmente rilevante perché la prova scritta si svolge su carta, senza computer.

Per tracciare un programma conviene costruire una tabella con almeno quattro elementi:

- il numero del passo;
- l'istruzione eseguita;
- lo stato della memoria dopo quel passo;
- l'output eventualmente prodotto.

Per esempio, dato il programma:

```
x = 4
print("ciao mondo")
print(x)
y = x + 3
```

la traccia minima è:

Passo	Istruzione	Stato della memoria	Output
1	`x = 4`	`x = 4`	
2	`print("ciao mondo")`	`x = 4`	`ciao mondo`
3	`print(x)`	`x = 4`	`4`
4	`y = x + 3`	`x = 4, y = 7`	

L'utilità di questo metodo è duplice. Da un lato obbliga a distinguere con precisione tra testo del programma, stato interno e output. Dall'altro mostra subito quali valori sono disponibili in un dato momento e quali no.

9. Valori booleani e operatori di confronto

Abbiamo visto fin qui vari tipi di operazioni che ci permettono di operare su `int`, `float` e `str`. Introduciamo ora una nuova classe di operazioni, le **operazioni di confronto**, che lavorano sempre su `int`, `float` e `str` ma calcolano un valore di un quarto tipo fondamentale: un valore **booleano** o `bool`.

I possibili valori booleani sono solo due, `True` e `False`, e rappresentano il risultato di una condizione vera o falsa.

```
3 > 1       # True
3 < 1       # False
"Anna" == "Anna"    # True
```

Gli operatori di confronto principali sono:

Operatore	Significato
`==`	uguale
`!=`	diverso
`<`	minore
`>`	maggiore
`<=`	minore o uguale
`>=`	maggiore o uguale

Attenzione a non confondere `=` con `==`: il primo assegna, il secondo confronta.

Queste operazioni diventano importanti perché preparano il terreno al controllo del flusso: una volta che il programma sa esprimere una condizione vera o falsa, può decidere che cosa fare in base al risultato.

Operazioni booleane su stringhe

Oltre ai confronti classici, Python fornisce strumenti utili per verificare proprietà delle stringhe:

Operazione	Significato	Esempio
`"cia" in "ciao"`	controlla se una sottostringa è contenuta	`True`
`"xyz" not in "ciao"`	controlla se una sottostringa non è contenuta	`True`
`"ciao".startswith("ci")`	controlla l'inizio della stringa	`True`
`"ciao".endswith("ao")`	controlla la fine della stringa	`True`
`"123".isdigit()`	verifica che tutti i caratteri siano cifre	`True`
`"abc".isalpha()`	verifica che tutti i caratteri siano alfabetici	`True`

10. Condizioni e scelta del ramo

Con i valori booleani il programma può **decidere il flusso di esecuzione**: eseguire certi blocchi di codice soltanto se una condizione è vera. Fino a questo momento tutte le istruzioni venivano eseguite in sequenza; con i costrutti condizionali il programma può saltare o scegliere tra blocchi alternativi.

Sintassi di `if`

```
if espressione_booleana:
    istruzione-1
    istruzione-2
    ...
```

- La parola chiave `if` introduce una condizione.
- Dopo `if` c'è un'espressione booleana.
- La prima riga termina con i due punti `:`, che aprono un blocco.
- Il blocco di istruzioni da eseguire deve essere **indentato** (rientrato).

Semantica di `if`

1. L'interprete valuta l'espressione booleana.
2. Se vale `True`, esegue il blocco indentato.
3. Se vale `False`, salta quel blocco e continua dopo.

Esempio e traccia:

```
x = input("Inserisci la tua età: ")
x = int(x)

if x > 18:
    print("Sei maggiorenne")

print("Hai " + str(x) + " anni")
```

Con input `33`:

Passo	Istruzione	Stato della memoria	Output
1	`x = input("Inserisci la tua età: ")`	`x` → `"33"`	`Inserisci la tua età:`
2	`x = int(x)`	`x` → `33`	
3	`if x > 18:` → True	`x` → `33`	
4	`print("Sei maggiorenne")`	`x` → `33`	`Sei maggiorenne`
5	`print("Hai " + str(x) + " anni")`	`x` → `33`	`Hai 33 anni`

Con input `15`, il passo 4 viene saltato: tra il passo 3 (condizione falsa) e il passo 4 l'interprete salta direttamente a `print("Hai " + str(x) + " anni")`. Il numero di passi eseguiti cambia anche se il codice sorgente è rimasto invariato.

Sintassi di `if-else`

```
if espressione_booleana:
    istruzione-1
    ...
else:
    istruzione-1
    ...
```

`else` non ha una condizione propria: raccoglie tutti i casi in cui la condizione dell'`if` vale `False`. Tra i due rami se ne esegue sempre uno solo.

Semantica di `if-else`

1. L'interprete valuta la condizione dell'`if`.
2. Se vale `True`, esegue il blocco dell'`if`.
3. Se vale `False`, esegue il blocco dell'`else`.

Esempio:

```
x = int(input("Inserisci la tua età: "))

if x > 18:
    print("Sei maggiorenne")
else:
    print("Sei minorenne")

print("Hai " + str(x) + " anni")
```

Sintassi di `if-elif-else`

Quando i casi possibili sono più di due, si usa `elif` per aggiungere ulteriori condizioni:

```
if espressione_booleana:
    istruzione
elif espressione_booleana:
    istruzione
elif espressione_booleana:
    istruzione
else:
    istruzione
```

Semantica di `if-elif-else`

L'interprete controlla i rami nell'ordine in cui sono scritti:

1. prova la condizione dell'`if`;
2. se è falsa, prova il primo `elif`;
3. continua finché trova la prima condizione vera ed esegue quel blocco;
4. se nessuna condizione è vera, esegue il blocco `else`.

Due conseguenze importanti:

- **conta l'ordine dei rami**: viene eseguito solo il primo ramo che risulta vero;
- `else` copre il caso residuo e viene eseguito solo se tutte le condizioni precedenti sono false.

Esempio: la maggiore età varia da paese a paese.

```
x = int(input("Inserisci la tua età: "))

if x > 21:
    print("Sei maggiorenne in USA")
elif x > 18:
    print("Sei maggiorenne in Italia ma non in USA")
else:
    print("Sei minorenne")
```

Stato e output nei rami condizionali

Durante la simulazione di programmi con condizioni valgono le stesse regole già viste:

- le variabili già presenti nello stato continuano a esistere;
- una riassegnazione aggiorna il valore di una variabile;
- una nuova assegnazione aggiunge un nuovo nome allo stato;
- `print(...)` produce in output il valore corrente di un'espressione, non il suo nome testuale.

Se, per esempio, una variabile `etichetta` contiene `"medio"`, allora `print(etichetta)` produce in output `medio`, non la parola `etichetta`.

11. Condizioni composte

Fino a qui abbiamo usato condizioni semplici come `x > 0` o `eta >= 18`. In molti programmi però occorre combinare più condizioni insieme. Python fornisce tre operatori che lavorano direttamente su valori booleani: `and`, `or` e `not`.

`and`

```
A and B
```

vale `True` soltanto quando **entrambe** le condizioni sono vere. Se almeno una è falsa, il risultato è `False`.

Tavola di verità:

A	B	A and B
`False`	`False`	`False`
`False`	`True`	`False`
`True`	`False`	`False`
`True`	`True`	`True`

Esempio: `x > 0 and x < 10` vale `True` solo quando `x` è contemporaneamente positivo e minore di 10.

`or`

```
A or B
```

vale `True` quando **almeno una** delle due condizioni è vera. Vale `False` soltanto se sono false entrambe.

Tavola di verità:

A	B	A or B
`False`	`False`	`False`
`False`	`True`	`True`
`True`	`False`	`True`
`True`	`True`	`True`

Esempio: `voto < 18 or voto > 30` vale `True` se il voto è fuori dall'intervallo valido.

`not`

```
not A
```

inverte il valore booleano: se `A` è `True`, `not A` è `False`; se `A` è `False`, `not A` è `True`.

Tavola di verità:

A	not A
`False`	`True`
`True`	`False`

Esempio: `not x > 0` equivale a `x <= 0`.

12. Ordine di precedenza degli operatori

Quando una condizione contiene più operatori, Python usa un **ordine di precedenza**: alcune operazioni vengono valutate prima di altre. Nelle espressioni più comuni vale questa gerarchia:

1. parentesi `()`;
2. confronti: `<`, `<=`, `==`, `!=`, `>`, `>=`;
3. `not`;
4. `and`;
5. `or`.

Quindi `not` ha precedenza su `and`, e `and` ha precedenza su `or`. Per esempio:

```
True or False and False
```

non si legge come `(True or False) and False`, ma come `True or (False and False)`, che vale `True`.

Allo stesso modo:

```
not x > 0 and y > 0
```

si legge come `(not (x > 0)) and (y > 0)`.

Come in matematica, le parentesi permettono di cambiare esplicitamente l'ordine delle operazioni. Quando una condizione è complessa, è buona pratica usarle per rendere il codice più chiaro, anche quando non strettamente necessarie.