

# Primi Passi in Python

---



# Tipi di Dato

## Abbiamo detto in passato che:

- Un elaboratore definisce quali tipi di dato possiamo manipolare
- Un linguaggio di alto livello definisce un elaboratore astratto

## Vediamo quindi quali tipi di dato Python ci permette di manipolare

Questi possono essere distinti in:

- Tipi di dato **semplici**:
  - numeri interi, numeri "reali" (floating point), valori logici, stringhe
- ...Tipi di dato **composti**
  - tuple, liste, dizionari, insiemi, classi
- ...Più qualche caso particolare (e.g. funzioni)



# Tipi di Dato Numerici

Per specificare un **numero intero** potete scriverlo come fate di solito:

```
In [1]: 10
```

```
Out[1]: 10
```

Per un **numero "reale"** potete usare la notazione decimale:

```
In [2]: 0.3
```

```
Out[2]: 0.3
```

...Oppure quella scientifica:

```
In [3]: 1.2e3
```

```
Out[3]: 1200.0
```



# Numeri Interi? Numeri Reali?

**Tenete in mente che un calcolatore ha una **quantità finita** di memoria**

Questo causa difficoltà in un paio di casi:

- Primo: tipicamente si usa un numero di bit fissati per rappresentare un numero
  - Quindi non possiamo rappresentare **numeri troppo grandi/piccoli!**
- Secondo: è complesso gestire numeri con un **rappresentazione infinita**
  - E.g. numeri periodici, o irrazionali
  - Per la precisione: non possono essere manipolati in modo esatto
  - ...Anche se ci sono tecniche per mitigare questi problemi



# Numeri Interi? Numeri Reali?

## Il caso dei numeri reali è particolarmente complesso

Internamente, sono rappresentati come:

$$\text{mantissa} \times 2^{\text{esponente}}$$

- I.e. viene letteralmente usata la notazione scientifica
- ...Semplicemente usando **2** come base invece di **10**

Per questa ragione, si parla di numeri **in virgola mobile (floating point)**

## Questa scelta ha delle conseguenze importanti

Non faremo una analisi approfondita, ma ricordate che:

- Alcuni numeri reali semplicemente **non sono rappresentabili**  $\Rightarrow$
- ...Ogni calcolo che usa numeri floating point è soggetto ad approssimazioni



# Espressioni

In un linguaggio di programmazione:

Si chiama **espressione** una notazione che denota un valore

...I.e. un testo che può essere **valutato** producendo un valore

**Quando abbiamo scritto ed "eseguito" dei numeri prima...**

...Quello che abbiamo fatto è stato valutare espressioni

- Per essere precisi, abbiamo **scritto del testo**
- ...Che è stato **interpretato**, denotando un **valore** (il risultato)

**Se l'ultima istruzione è una espressione, Jupyter ne stampa il valore**

```
In [4]: 3
```

```
Out[4]: 3
```



## ...Ed Istruzioni

Inoltre, in un linguaggio di programmazione:

Si chiama **istruzione** una notazione che può essere eseguita

Intuitivamente, è un ordine che diamo all'elaboratore

- Una espressione, da sola su una riga, è un esempio di istruzione, e.g.:

```
2
```

- ...Ma ci sono molti altri tipi di istruzione!

**Le istruzioni sono il fondamento dei linguaggi imperativi**

- Tipicamente, l'elaboratore le esegue **nell'ordine in cui sono scritte**
- ...Ed infatti anche in Python è così



# Variabili

È possibile memorizzare valori in **variabili**

Si usa una istruzione di **assegnamento** con la sintassi:

```
<identificatore> = <espressione>
```

Dove:

- `<identificatore>` è il nome della variabile da utilizzare e segue la sintassi:
- `<espressione>` è una qualsiasi espressione

**Quando l'istruzione viene eseguita:**

- L'espressione viene valutata
- Una variabile di nome `<identificatore>` viene **definita** (i.e. "creata")
- ...Ed il valore denotato viene memorizzato nella variabile





# Variabili

## Non tutti i nomi di variabile sono **validi**

Gli identificatori devono rispettare la sintassi:

```
<identificatore> ::= <lettera> | _ {<lettera> | _ | <numero naturale>}
```

- Si inizia con una lettera o un "\_" (underscore)
- ...Cui seguono, lettere, underscore, o cifre

Qualche esempio di nome valido:

```
a = 2  
r2d2 = 1.5  
_nascosta = 1  
non_siate_troppo_prolissi = 0
```



# Variabili

**Il nome di una variabile, se non seguito dal simbolo "="...**

...È una **espressione**, che denota il valore delle variabile:

```
In [5]: a = 10  
a
```

```
Out[5]: 10
```

- Nella prima istruzione "a" compare a sx del simbolo "="
  - ...Quindi indica il nome della variabile in cui memorizzare il valore
- Nella seconda istruzione, "a" compare da solo:
  - ...Quindi conta come espressione



# Variabili come Contenitori

**Il concetto di variabile in Python è diverso da quello usato in matematica**

...Anche se un po' ci somigliano

- Una variabile in matematica è un nome/simbolo associato ad un valore
- Una variabile in Python è un **contenitore** per un dato
  - Inizia ad esistere solo **dopo** che gli viene assegnato un valore
  - Può essere riempita **più volte**
  - Può essere **"vuota"**

**Vediamo qualche conseguenza...**



# Variabili come Contenitori

In matematica potete scrivere così:

$$\begin{aligned}y &= x \\ x &= 2\end{aligned}$$

In Python no!

- Prima bisogna definire  $x$
- ...E solo successivamente può essere utilizzata

```
In [6]: y = z  
        z = 2
```

```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipykernel_11/2387470407.py in <module>  
----> 1 y = z  
      2 z = 2
```

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



# Variabili come Contenitori

In matematica potete scrivere così:

$$y = x$$
$$x = 2$$

In Python, questo è un modo corretto di ottenere lo stesso risultato:

```
In [7]: x = 2  
        y = x  
        y
```

```
Out[7]: 2
```



# Variabili con Contenitori

**Si può riempire una variabile più volte**

Qui un singolo riempimento:

```
In [8]: x = 2  
x
```

```
Out[8]: 2
```

...E qui due:

```
In [9]: x = 2  
x = 3  
x
```

```
Out[9]: 3
```

Il secondo assegnamento sovrascrive il contenuto



# Variabili come Contenitori

Una variabile può **esistere**, ma essere **vuota**

Si ottiene questo comportamento mediante il valore speciale **None**

```
In [10]: x = None
          x
```

- L'istruzione di assegnamento crea la variabile
- ...Ma essa rimane vuota
- O meglio ancora: **contiene niente** (cioè `None`)

Se l'ultima espressione denota `None`, Jupyter non la stampa



# Tempo di Vita delle Variabili

**Ogni variabile in Python ha un tempo di vita**

I.e. l'intervallo di tempo all'interno del quale essa esiste

- Le variabili che stiamo usando al momento si dicono **globali**
- ...Ed hanno come tempo di vita **l'intera esecuzione dell'interprete**

**In Jupyter, vuol dire che durano quanto il kernel stesso**

Una volta che una variabile globale è stata creata, è accessibile **da qualsiasi cella**

```
In [11]: b = 2
```

```
In [12]: b
```

```
Out[12]: 2
```





# Stato del Kernel

Il kernel Jupyter **non viene riavviato** ad ogni cella

...Ma solo se richiesto esplicitamente (o in caso di certi errori)

- Quindi ogni variabile globale "vive" quanto il kernel
- ...E lo "stato" delle variabili è indipendente dalle celle

**Questo può portare ad alcuni risultati bizzarri**

Provate ad eseguire la **seconda** cella, quindi la prima:

```
In [13]: c
```

```
-----  
NameError                                Traceback (most recent call last)  
/tmp/ipykernel_11/3235490055.py in <module>  
----> 1 c  
  
NameError: name 'c' is not defined
```

```
In [14]: c = 3
```



# Espressioni Semplici e Composte

**Le espressioni si possono dividere in semplici e composte**

Abbiamo già incontrato i due tipi più importanti di espressioni semplici:

- Costanti
- Nomi di variabile (se a dx del segno "=")

...Ma le espressioni più interessanti sono quelle composte

**Un'espressione composta è una espressione che consente di combinare altre espressioni**

**Ve ne sono di due tipi principali:**

- Chiamate a funzione
- Operatori



# Chiamate a Funzione

Il meccanismo base per la espressioni composte è la chiamata a funzione:

## Una chiamata a funzione

- È una notazione che esegue un sotto-programma
- ...Passandogli zero o più argomenti
- ...Per ottenere un risultati

## Qualche nota importante:

- Il sottoprogramma è individuato da un nome
- Gli argomenti sono espressioni
- ...Che vengono valutate prima dell'esecuzione del sottoprogramma
- Al sottoprogramma vengono passati i risultati della valutazione



# Chiamate a Funzione

La sintassi è la seguente:

```
<chiamata a funzione> ::= <nome funzione>(<argomenti>)  
<argomenti> ::= [<espressione> {, <espressione>}]
```

- Il nome della funzione è seguito da parentesi tonde
- Gli argomenti (se presenti) vanno tra parentesi
- Se ce ne è più di uno, vanno separati con virgole

**Vediamo un esempio**

```
In [15]: abs(-2)
```

```
Out[15]: 2
```

- `abs` è il nome della funzione (valore assoluto)



- Passando `-2` come argomenti, si ottiene `2`

# Chiamate a Funzione

**Gli argomenti possono essere espressioni di qualunque tipo**

...Incluse altre espressioni composte

```
In [16]: pow(3, abs(-2))
```

```
Out[16]: 9
```

- `abs(-2)` è passato come argomento a `pow`
- `pow(a, b)` restituisce il risultato di  $a^b$

**Gli argomenti sono valutati **prima** dell'esecuzione del sottoprogramma**

- Si comincia sempre dalle espressioni semplici
  - In questo caso 3 e -2
- Quindi si valutano una per una le chiamate a funzione:

  In questo caso prima `abs` e poi `pow`

# Operatori

Gli **operatori** sono (quasi tutti) funzioni con **sintassi semplificata**

Corrispondono ad operazioni di utilizzo comune

- E.g.  $+$ ,  $-$ , etc.

Nella maggior parte dei casi, la sintassi semplificata è quella "naturale"

- E.g. per indicare  $2 + 3$  scriviamo:

```
In [17]: 2 + 3
```

```
Out[17]: 5
```

**Dietro le quinte, sono equiparabili a chiamate a funzione**

- Il nome del sottoprogramma corrisponde al simbolo

- Prima si valutano gli argomenti, poi si esegue il sottoprogramma



# Operatori Aritmetici

Python offre i seguenti **operatori aritmetici** di uso comune

Il testo dopo il simbolo # è un **commento** e viene ignorato dall'interprete

```
In [18]: 2 + 3 # somma
```

```
Out[18]: 5
```

```
In [19]: 2 * 3 # moltiplicazione
```

```
Out[19]: 6
```

```
In [20]: 2 - 3 # differenza
```

```
Out[20]: -1
```

```
In [21]: 2 / 3 # divisione
```

```
Out[21]: 0.6666666666666666
```

```
In [22]: 2**3 # elevamento a potenza
```

```
Out[22]: 8
```



# Operatori Aritmetici

...Ma anche qualche operatore più bizzarro

```
In [23]: 5 // 2 # divisione intera
```

```
Out[23]: 2
```

La **divisione intera** è la divisione "delle elementari"

```
In [24]: 5 % 2 # modulo
```

```
Out[24]: 1
```

...Ed infatti ha un resto, recuperabile con l'operatore **modulo**





# Operatori di Confronto e Valori Logici

Possiamo confrontare numeri usando gli **operatori di confronto**

E.g.  $<$ ,  $\leq$ , etc.

- Questi accettano come argomenti dei **valori numerici**
- ...Ma restituiscono un **valore logico**
- ...Ossia un valore vero (costante `True`) o falso (costante `False`)

**Vediamo un paio di esempi**

```
In [25]: 2 <= 3 # minore o uguale
```

```
Out[25]: True
```

```
In [26]: 3 <= 2
```

```
Out[26]: False
```

I valori logici sono un **tipo di dato primitivo** in Python



# Operatori di Confronto e Valori Logici

Vediamo tutti gli operatori di confronto disponibili in Python

```
In [27]: 2 < 3 # minore
```

```
Out[27]: True
```

```
In [28]: 3 <= 2 # minore o uguale
```

```
Out[28]: False
```

```
In [29]: 2 > 3 # maggiore
```

```
Out[29]: False
```

```
In [30]: 3 >= 2 # maggiore o uguale
```

```
Out[30]: True
```

```
In [31]: 2 != 3 # diverso
```

```
Out[31]: True
```

```
In [32]: 2 == 2 # uguale
```

```
Out[32]: True
```



# Uguaglianza ed Assegnamento

L'operatore di uguaglianza in Python è `==` invece che `=`

...E lo stesso succede in molti linguaggi di programmazione

- La ragione è il simbolo `=` è già riservato per un altro operatore
- Lo abbiamo già incontrato: si tratta dell'**operatore di assegnamento**

## Vediamo un esempio

Per assegnare il valore 3 ad `a` usiamo:

```
In [33]: a = 3
```

Per controllare se la variable `a` ha il valore 3 usiamo:

```
In [34]: a == 3
```

```
Out[34]: True
```



# Assegnamento con Accumulo

**Esiste una sintassi compatta per modificare una variabile con una operazione**

La sintassi è:

```
<variabile> <operatore>= <espressione>
```

Per esempio:

```
In [35]: a = 0  
a += 2  
a
```

```
Out[35]: 2
```

- Il valore di `<espressione>` viene (in questo caso) sommato ad `a`
- ...Ed il risultato viene scritto di nuovo in `a`

**Ne faremo uso di tanto in tanto**



# Operatori Logici

Gli **operatori logici** permettono di combinare valori logici

Disponibili i seguenti operatori:

- Operatore **and**, con sintassi: `<espr1> and <espr2>`
  - Restituisce `True` se sia `<espr1>` che `<espr2>` denotano `True`
- Operatore **or** con sintassi: `<espr1> or <espr2>`
  - Restituisce `True` se almeno uno tra `<espr1>` e `<espr2>` denota `True`
- Operatore **not** con sintassi: `not <espr>`
  - Restituisce `True` se `<espr>` denota `False`, e viceversa

**Sono di solito impiegati per formulare condizioni complesse**



# Operatori Logici

Vediamo qualche esempio:

```
In [36]: a = 3  
(2 <= a) and (a <= 4)
```

```
Out[36]: True
```

```
In [37]: (2 > a) or (a == a)
```

```
Out[37]: True
```

```
In [38]: not (a <= 4)
```

```
Out[38]: False
```

Per la condizione  $(1 \leq a) \text{ and } (a \leq u)$  esiste anche una sintassi compatta:

```
In [39]: 2 <= a <= 4
```

```
Out[39]: True
```



# Operatori sulla Rappresentazione Binaria

**Alcuni operatori agiscono sulla rappresentazione binaria dei numeri**


Non li useremo direttamente, ma è importante sapere che esistono

- Tutti gli operatori di questo gruppo si applicano a numeri interi
- ...Ed agiscono bit per bit sulla loro rappresentazione

**Vediamoli brevemente:**

```
In [40]: print(2 & 3)
         print(2 | 3)
         print(2 ^ 3)
```

```
2
3
1
```

- L'operatore & effettua un "and" logico, bit per bit
- L'operatore | effettua un "or" logico, bit per bit
-  L'operatore ^ effettua uno "xor" logico (or esclusivo)

# Associatività e Priorità

**Gli operatori seguono le normali regole di priorità ed associatività**

Per esempio:

```
In [41]: 2 * 3 + 4
```

```
Out[41]: 10
```

- Prima viene eseguito  $2 * 3$ , quindi  $+ 4$

Ancora:

```
In [42]: 10 - 2 - 3
```

```
Out[42]: 5
```

- Prima viene eseguito  $10 - 2$ , poi  $- 3$





# Operatori per Stringhe

## Alcuni operatori sono applicabili anche alle stringhe

...Ed in questo caso assumono un significato particolare

- L'operatore "+", applicato a due stringhe, le **concatena**

```
In [44]: 'ciao ' + 'mondo'
```

```
Out[44]: 'ciao mondo'
```

- L'operatore "\*", applicato ad una stringa e ad un numero naturale  $n$
- ...**Ripete** la stringa ***n*** volte

```
In [45]: 'bla ' * 3
```

```
Out[45]: 'bla bla bla '
```



# Associatività e Priorità

## Visto che gli operatori sono tanti, però...

...È utile ricordarsi che le priorità seguono questo ordine:

- Chiamate a funzione
- Elevamento a potenza ( `**` )
- Operatori unari ( `+`, `-`, etc.)
- Operatori moltiplicativi ( `*`, `/`, `//`, `%` )
- Operatori additivi ( `+`, `-` )
- Operatori di confronto ( `<`, `<=`, `==`, etc.)
- Operatore logico `not`
- Operatore logico `and`
- Operatore logico `or`



Nel dubbio, potete **usare le parentesi** per forzare un ordine di valutazione

# Stampa e Stringhe

Per scrivere testo su terminale in Python si usa la funzione **print**

Per esempio:

```
In [44]: print('Hello, world!')
```

```
Hello, world!
```

**La funzione `print` accetta di solito come argomento una stringa**

...Ossia una **porzione di testo**

- Le stringhe in Python sono tipi primitivi
- Una costante stringa si costruisce scrivendo testo tra '...' (apici singoli)
- ...O tra doppi apici, i.e. "..."



# Stampa e Stringhe

## Vediamo qualche esempio e qualche eccezione

```
In [45]: print('questa è una stringa normale')
print('anche questa, del resto')
print("così posso scrivere l'apostrofo")
print('e così i "doppi apici"')
print("in alternativa posso usare una \"escape sequence\" ")
print("cioè una sequence di caratteri che inizia con \\ ed e seguita da altri caratteri")
print("...e corrisponde ad un determinato simbolo")
```

```
questa è una stringa normale
anche questa, del resto
così posso scrivere l'apostrofo
e così i "doppi apici"
in alternativa posso usare una "escape sequence"
cioè una sequence di caratteri che inizia con \ ed e seguita da altri caratteri
...e corrisponde ad un determinato simbolo
```

Trovate le escape sequence disponibili [su questa pagina](#)



# Stampa e Stringhe

**Possiamo passare più argomenti a `print`**

...Che li stamperà, separati da spazi

```
In [46]: print('Hello', 'world')
```

```
Hello world
```

`print` è in grado di stampare anche numeri

```
In [47]: print('La variabile "a" vale:', a)
```

```
La variabile "a" vale: 3
```



# Interpolazione di Stringhe

Possiamo far comparire **valori** all'interno di stringhe

Vediamo con un esempio:

```
In [48]: print(f'La variabile "a" vale {a}')
```

```
La variabile "a" vale 3
```

- Davanti alla stringa mettiamo una `f` (sta per "formatted")
- Nella stringa, possiamo **inserire espressioni**
- ...Mettendole tra parentesi graffe (qui spiega come scriverle)

**Così facendo:**

- L'espressione viene valutata
- ...Ed il risultato viene usato per costruire la costante stringa



# Interpolazione di Stringhe

Questo metodo si chiama **interpolazione di stringhe** (da Python 3.6 in poi)

- Permette di stampare espressioni qualsiasi:

```
In [49]: print(f'"a" + 2 vale: {a + 2}')
```

```
"a" + 2 vale: 5
```

- ...E di specificare **come** vogliamo stampare il numero:

```
In [111]: print(f'"a"/4 vale: {a/4:.3f}')
```

```
"a"/4 vale: 0.750
```

- : indica che vogliamo specificare un **formato** per la stampa
- .f che vogliamo stampare il numero in formato decimale
- .3f che vogliamo visualizzare tre cifre decimali

