Python

Argomenti trattati

1

Introduzione a Python

Storia, caratteristiche e diffusione

2

Ambiente di sviluppo

Installazione e configurazione

3

Elementi fondamentali

Variabili, tipi di dati e operazioni

4

Strutture di controllo

Selezione e cicli

5

Funzioni e classi

Modularità e programmazione orientata agli oggetti

6

Gestione dei file

Lettura e scrittura di dati

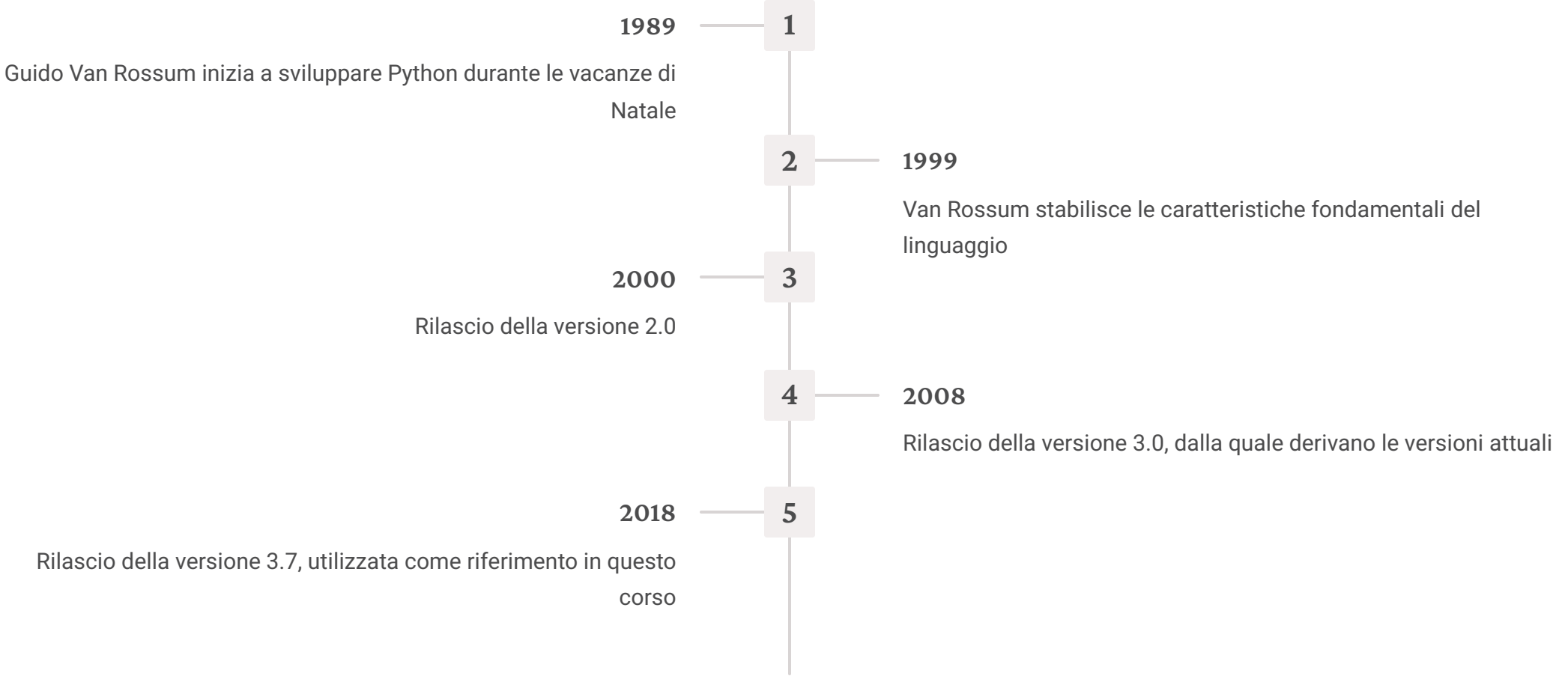
Breve storia di Python

Python fu ideato nel periodo di Natale del 1989 dall'informatico olandese **Guido Van Rossum**, come progetto di programmazione per hobby.

Il nome "**Python**" deriva dalla serie televisiva britannica "Monty Python's Flying Circus", di cui Van Rossum era un grande fan.

Van Rossum è definito dalla comunità di Python come «benevolo dittatore» a vita poiché continua ad avere l'ultima parola sulle decisioni che riguardano il processo di sviluppo del linguaggio.

Evoluzione di Python



Il sito web **https://www.python.org** è il riferimento per scaricare Python, consultare la documentazione e seguire gli eventi della comunità.

Caratteristiche fondamentali di Python

Linguaggio interpretato

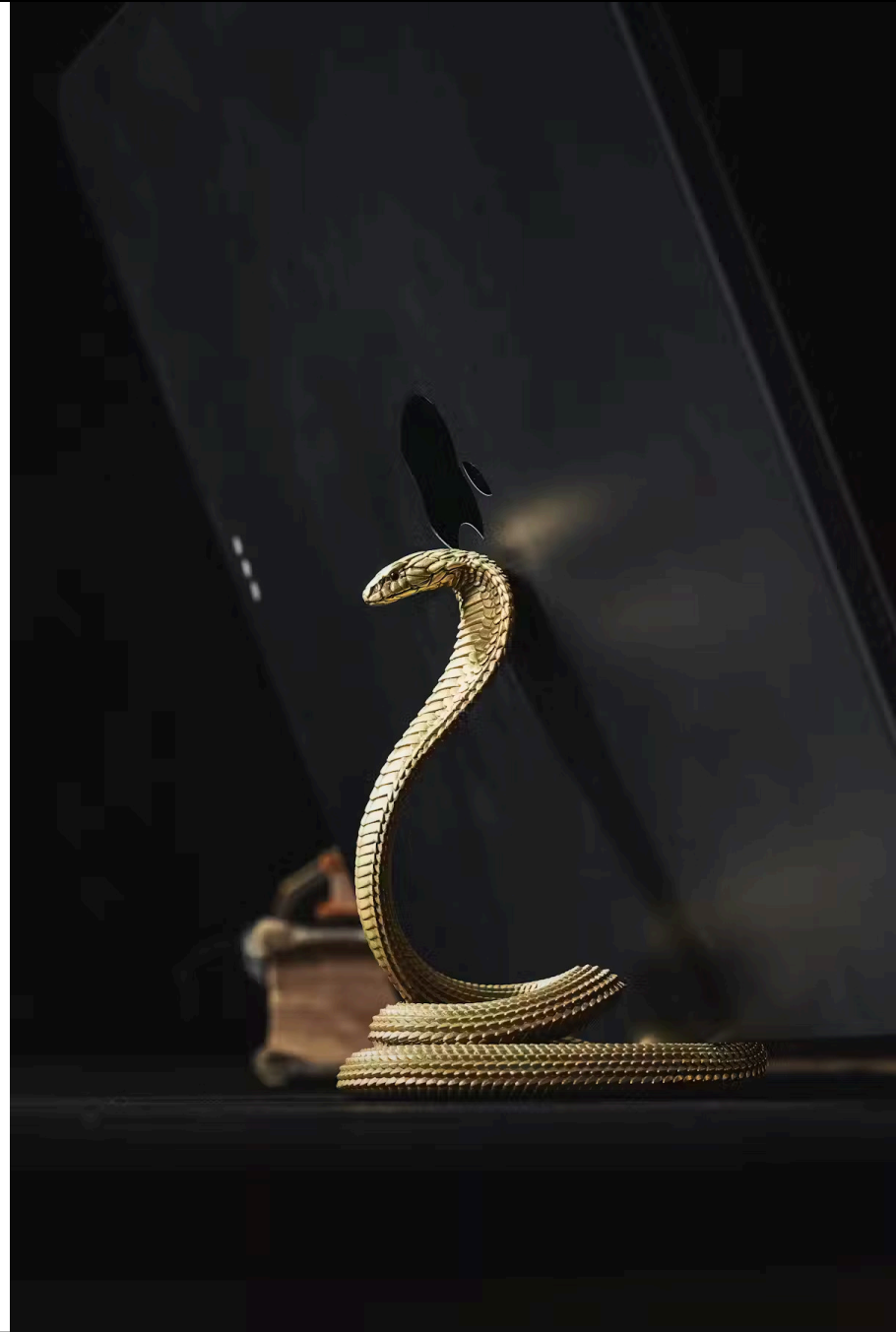
Richiede l'interprete per essere eseguito. Il codice sorgente viene tradotto in bytecode, eseguibile dalla **Python Virtual Machine**.

Multipiattaforma

Esistono interpreti per tutti i sistemi operativi più diffusi, rendendo Python un linguaggio utilizzabile ovunque.

Multiparadigma

Supporta diversi stili di programmazione: imperativo, orientato agli oggetti e funzionale.



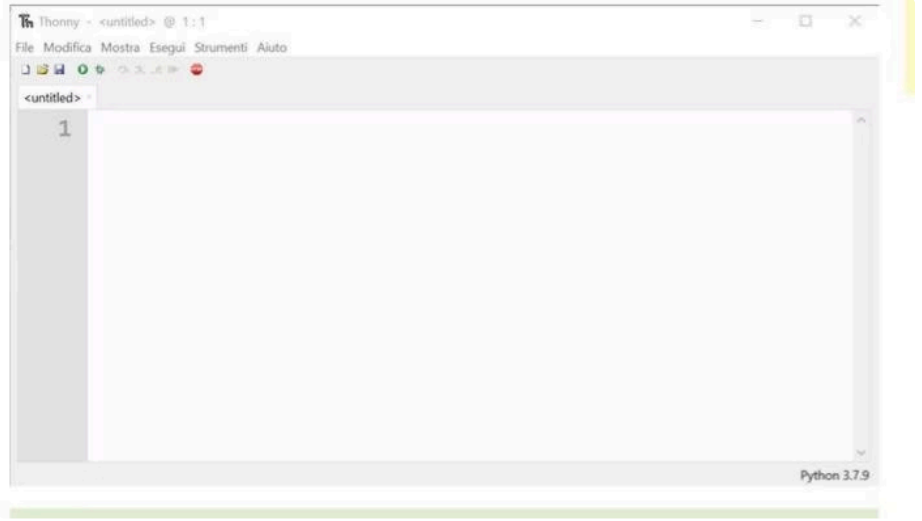
L'ambiente di sviluppo

Per iniziare a programmare in Python è necessario installare:

- L'**interprete Python** (versione $\geqslant$ 3.7)
- Un **editor di testo** o un IDE

In questo corso utilizzeremo Thonny IDE, un ambiente di sviluppo integrato ideato per l'apprendimento di Python.

Thonny include sia l'interprete Python sia un editor evoluto per la scrittura del codice sorgente.



Interfaccia di Thonny IDE

IDE - Integrated Development Environment

Un IDE (ambiente di sviluppo integrato) è un software fondamentale per i programmatori perché supporta lo sviluppo, il debugging e l'esecuzione del codice sorgente.

Thonny IDE

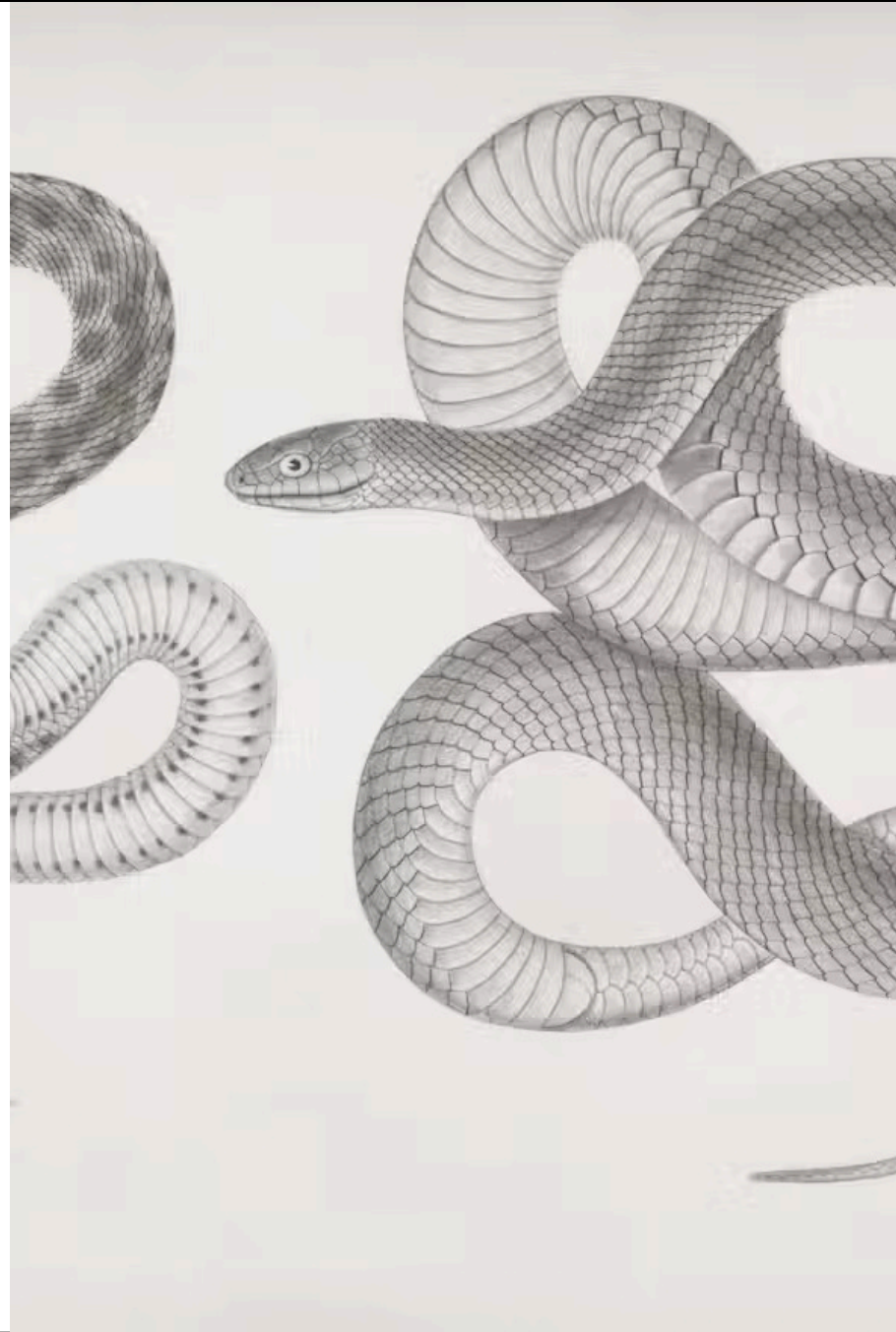
Ideale per principianti, semplice e intuitivo

Download: **https://thonny.org/**

Visual Studio Code

Più adatto per lo sviluppo di applicazioni complesse

Download: https://code.visualstudio.com



Il primo programma in Python

Iniziamo con il classico "Hello World" utilizzando l'interprete interattivo:

```
>>> print("Ciao mondo")
Ciao mondo
```

Possiamo anche usare l'interprete come calcolatrice:

```
>>> 2 + 3
5
>>> 10 / 2
5.0
```

Per scrivere programmi più complessi, utilizziamo l'editor di Thonny:

```
# Salviamo questo codice in ciaomondo.py
print("Ciao mondo")
```

Eseguiamo il programma cliccando sull'icona verde di esecuzione.

L'output sarà visibile nella finestra Shell.

Easter Egg di Python

Digitando il comando `import this` nell'interprete, otteniamo un easter egg molto istruttivo: le regole di buona programmazione in Python.

Gli **easter egg** sono messaggi, immagini o funzionalità nascoste in un software, attivabili con una sequenza di tasti o comandi.

Le parole chiave di Python

Il linguaggio Python ha solo 33 parole chiave native. Queste parole sono sufficienti per scrivere qualunque programma, anche quelli che risolvono problemi complessi.

Parole di controllo

if, else, elif, for, while, break, continue, return, try, except, finally, pass, yield

Operatori logici

and, or, not, is, in

Definizioni

def, class, lambda, import, from, as, global

Valori speciali

True, False, None

Definizioni in Python

Quando si parla di definizioni in Python si fa riferimento a delle parole chiave che consentono di definire funzioni, classi, variabili constanti etc

def saluta(): print("Ciao!")

class Studente:

def **init**(self, nome): self.nome = nome

I moduli in Python

I moduli sono **librerie di comandi, funzioni** e classi impacchettate insieme per fornire funzionalità aggiuntive.

Moduli nativi (built-in)

Inclusi nell'installazione di Python (es. `random`, `math`)

Moduli esterni

Creati dal programmatore o da terze parti (es. `numpy`, `pandas`, `matplotlib`)

La documentazione ufficiale, aggiornata e ricca di esempi, è consultabile sul sito web https://docs.python.org/3/

Caratteristiche di Python

Python è un linguaggio **tipizzato**, ma con alcune particolarità:

È **dinamicamente tipizzato** → non devi dichiarare il tipo delle variabili, lo interpreta Python al momento dell'esecuzione.

x = 10 # qui x è un int

x = "ciao" # ora x diventa una stringa

È **fortemente tipizzato** → non permette conversioni implicite "azzardate".

Ad esempio:

print("5" + 2) # Errore: non puoi sommare stringa e intero

print("5" + str(2)) # Funziona: "52"

Quindi: tipizzazione **dinamica e forte.**

Caratteristiche di Python

Python è un linguaggio **interpretato**, cioè il codice sorgente viene letto ed eseguito riga per riga da un interprete (es. CPython).

In realtà, prima dell'esecuzione, Python compila automaticamente il codice in un formato intermedio chiamato ***bytecode*** (.pyc), che poi viene eseguito dalla macchina virtuale di Python.

In sintesi:

Java

Il codice .java viene compilato in bytecode (.class) con javac.

Il bytecode viene eseguito dalla JVM (Java Virtual Machine). → Quindi Java è compilato in bytecode e poi interpretato/eseguito dalla JVM.

Python

Il codice **.py** viene letto dall'interprete.

L'interprete lo traduce automaticamente in **bytecode (.pyc).**

Il bytecode viene eseguito dalla Python Virtual Machine. → Quindi Python è principalmente interpretato, anche se **ha una fase di compilazione interna nascosta**.

Le variabili in Python

Una **variabile** è uno spazio di memoria RAM contenente dati a cui è stato attribuito un **identificatore** (nome).

```
# Inizializzazione di variabili
temperatura = 25
giorno = "Lunedì"

# Stampa dei valori
print(temperatura)  # Output: 25
print(giorno)       # Output: Lunedì
```

Python è **case sensitive**: le lettere maiuscole sono distinte da quelle minuscole. Ad esempio, `temperatura` è un nome diverso da `Temperatura`.

Regole per i nomi delle variabili

Caratteri consentiti

Lettere, numeri e underscore (_)

Primo carattere

Non può iniziare con un numero

Parole riservate

Non si possono usare le 33 parole chiave di Python

È consigliabile usare nomi esplicativi che descrivano il contenuto della variabile. Per nomi composti da più parole, è comune usare la convenzione snake_case, come `prezzo_giacca`.

Le buone pratiche di stile per i programmatori Python sono descritte nel documento PEP 8: https://www.python.org/dev/peps/pep-0008/

Tipi di dati in Python

Intero (int)

Numeri interi positivi o negativi

`età = 25`

Floating point (float)

Numeri reali con o senza virgola

`temperatura = 36.5`

Stringa (str)

Caratteri singoli e parole

`nome = "Mario"`

Booleano (bool)

Valori di verità

`attivo = True`

Oggetto (object)

Strutture dati complesse

`lista = [1, 2, 3]`

Oggetti

In Python **ogni cosa è un oggetto**: numeri, stringhe, funzioni, classi, persino i moduli.
Un oggetto è una **struttura che contiene**:

- uno **stato** (dati, attributi),
- un **comportamento** (metodi, cioè funzioni legate all'oggetto),
- un **tipo** (la classe da cui proviene).

x = 5 print(type(x)) # <class 'int'>

5 è un oggetto della classe int

Oggetti

E' possibile definire un tipo di oggetto personalizzato (classe) con la parola chiave **class**:

class Studente: def **init**(self, nome, corso):

self.nome = nome # attributo

self.corso = corso # attributo

```
def saluta(self):          # metodo
    return f "Ciao, sono {self.nome} e seguo {self.corso}."

s1 = Studente("Anna", "Informatica")
print(s1.nome)      # accedo all'attributo → Anna
print(s1.saluta())  # chiamo il metodo → "Ciao, sono Anna e seguo Informatica."
```

Controllo dinamico dei tipi

Python è un linguaggio con **controllo dinamico dei tipi**: non è necessario dichiarare in anticipo il tipo di una variabile.

```
a = 3.14       # float
b = 42         # int
c = "Python"    # str
d = True       # bool
e = "42"       # str (non int!)

print(type(a))  #
print(type(b))  #
print(type(c))  #
print(type(d))  #
print(type(e))  #
```

La funzione `type()` restituisce il tipo di un oggetto.

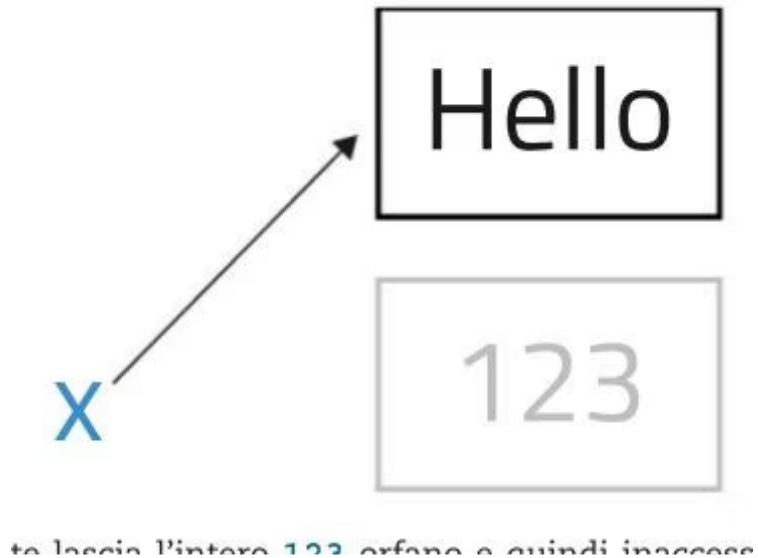
Garbage collection

In Python, le variabili sono come **etichette** apposte a oggetti contenenti valori. (Un pò come i riferimenti in Java)

```
x = 123
y = x    # y punta allo stesso oggetto di x
x = "Hello"  # x ora punta a un nuovo oggetto
```

L'oggetto `123` diventa "orfano" e sarà eliminato dal processo di **garbage collection**.

In **Python una variabile non è un "contenitore" di un valore (come succede ad esempio in C o in Java con i tipi primitivi)**, ma piuttosto un'**etichetta (o un riferimento)** che viene attaccata a un oggetto in memoria.



La garbage collection è una forma di gestione automatica della memoria che ripulisce quella ancora allocata ma non più referenziata.

Garbage Collection

x = 10 In questo caso l'**oggetto** 10 (un intero) viene creato in memoria.

La variabile x non contiene il valore 10 direttamente: **è semplicemente un nome che punta a quell'oggetto**.

Se poi scriviamo:

y = x Ora sia x che y sono due etichette che indicano lo stesso oggetto 10. Non ci sono due copie distinte del numero 10, ma un solo oggetto condiviso.

Infine, se scriviamo: **x = 20** La variabile x non "cambia dentro" il valore 10.

Piuttosto, l'etichetta x viene staccata dall'oggetto 10 e appiccicata a un nuovo oggetto 20.

y, invece, resta ancora collegato al 10.

Oggetti mutabili e immutabili

In Python un oggetto è **immutabile** (es. numeri, stringhe, tuple), se **non può essere cambiato**: si creano nuovi oggetti e si ricollegano le variabili.

Se un oggetto è **mutabile** (es. liste, dizionari), le etichette possono riferirsi allo stesso oggetto, e le modifiche fatte tramite una variabile si riflettono anche se accedi con un'altra.

Esempio

x = 10

y = x

X e Y puntano allo stesso oggetto.

Oggetti mutabili

a = [1, 2, 3]

b = a

a.append(4)

Qui sia **a** che **b** puntano alla stessa lista, quindi modificando la lista con a, anche b “vede” il cambiamento.

Assegnazione multipla

Python permette di inizializzare più variabili con un solo operatore di assegnamento:

```
temperatura, giorno = 25, "Lunedì"
print(temperatura)  # Output: 25
print(giorno)       # Output: Lunedì
```

L'assegnazione multipla può essere usata anche per più di due variabili:

```
a, b, c = 4, 8, 5
```

Un'applicazione utile è lo scambio di valori tra due variabili in una sola linea:

```
a, b = b, a # Scambia i valori di a e b
```

La funzione print() e le stringhe

Le stringhe in Python possono essere delimitate sia da doppi apici (") che da apici singoli ('):

```
messaggio1 = "Ciao, come stai?"
messaggio2 = 'Sto bene, grazie!'
```

La funzione print() è utilizzata per stampare valori:

```
print(messaggio1) # Stampa una variabile
print("Ciao mondo!") # Stampa una stringa
print(2 + 3) # Stampa il risultato di un'espressione
print(print) # Stampa info sulla funzione stessa
```

Commenti in Python

I commenti sono parti di codice ignorate dall'interprete, utili per documentare il programma:

```
# Questo è un commento su una singola riga
x = 10  # Questo è un commento alla fine di una riga

"""
Questo è un commento
su più righe (docstring)
Utile per documentare funzioni e classi
"""
```

Le **docstring** sono utilizzate per documentare moduli, funzioni, classi e metodi. Vengono riconosciute dagli strumenti di generazione automatica della documentazione.

Stringhe formattate (f-string)

Le f-string sono un modo moderno e semplice per formattare le stringhe in Python:

```
a = 10
b = 5
print(f"La soluzione dell'equazione è {b/a}")
# Output: La soluzione dell'equazione è 0.5
```

In alternativa, si può usare la concatenazione con il carattere `+`:

```
print("La soluzione dell'equazione è " + str(b/a))
# Output: La soluzione dell'equazione è 0.5
```

Le f-string sono più leggibili e permettono di inserire espressioni Python direttamente nella stringa.

Sequenze di escape

Le sequenze di escape sono utilizzate per rappresentare caratteri speciali nelle stringhe:

\\ (backslash)

Scrive una barra rovesciata

\b (backspace)

Scrive uno spazio

\n (new line)

Inserisce un a capo

\t (tab)

Si sposta di una tabulazione

```
print(f"I computer sono stupidi.\nGli umani sono intelligenti.")
# Output:
# I computer sono stupidi.
# Gli umani sono intelligenti.
```

Operazioni con le stringhe

Concatenazione (somma)

```
a = "Ciao "
b = "mondo!"
c = a + b
print(c)  # Output: Ciao mondo!
```

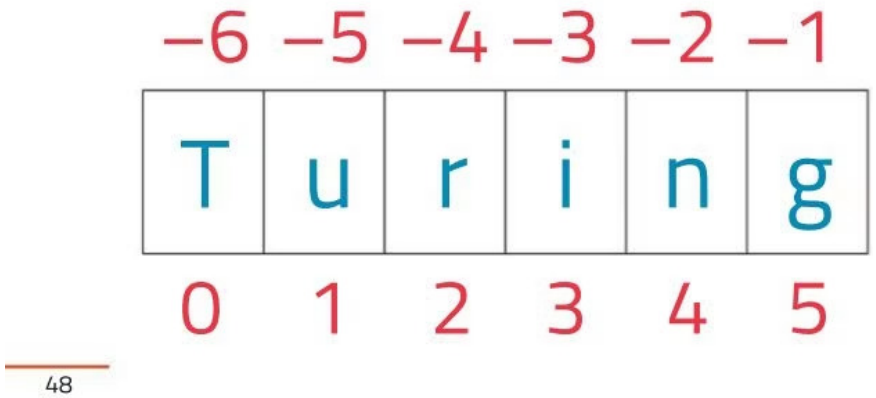
Ripetizione (moltiplicazione)

```
a = "Na"
b = a * 8 + " Batman!"
print(b)  # Output: NaNaNaNaNaNaNaNa
Batman!
```

Le stringhe in Python sono immutabili: una volta create, non possono essere modificate. Per "modificare" una stringa, è necessario crearne una nuova.

Indicizzazione e slicing delle stringhe

Una stringa è una sequenza di caratteri indicizzati. Python supporta sia indici positivi (da sinistra) che negativi (da destra):



48

```
frase = "I am Alan Turing"

# Accesso a singoli caratteri
print(frase[15]) # Output: g
print(frase[-1]) # Output: g

# Slicing
print(frase[5:9]) # Output: Alan
print(frase[:4]) # Output: I am
print(frase[10:]) # Output: Turing
print(frase[::2]) # Output: Ia lnTrn
```

I contenitori in Python

Python offre diversi tipi di contenitori per gestire collezioni di dati:

Tuple

Collezioni **immutabili** di oggetti

`(1, 2, 3)`

Liste

Collezioni **mutabili** di oggetti

`[1, 2, 3]`

Dizionari

Collezioni di coppie **chiave-valore**

`{"a": 1, "b": 2}`

Ciascun contenitore è costituito da elementi che possono essere interi, floating point, stringhe, booleani o altri oggetti.

Le tuple

Le tuple sono collezioni immutabili di oggetti: una volta create, non possono essere modificate.

```
punto = (3.5, 7.2)  # Tupla con due elementi floating point
print(punto[0])    # Output: 3.5 (coordinata x)
print(punto[1])    # Output: 7.2 (coordinata y)

giorni = ("Lunedì", "Martedì", "Mercoledì", "Giovedì",
          "Venerdì", "Sabato", "Domenica")
print(giorni[2])   # Output: Mercoledì
```

È possibile scomporre una tupla assegnando ciascun elemento a una variabile:

```
x, y = punto
print(x) # Output: 3.5
print(y) # Output: 7.2
```

Le liste

Le liste sono contenitori mutabili: sia la lista sia i suoi elementi possono essere modificati.

```
bookmarks = ["www.python.org", "www.wikipedia.org",
 "docs.python.org", "www.github.com"]

# Modifica di un elemento
bookmarks[1] = "it.wikipedia.org"

# Aggiunta di un elemento
bookmarks.append("scikit-learn.org")

# Accesso all'ultimo elemento
print(bookmarks[-1]) # Output: scikit-learn.org

# Slicing
print(bookmarks[1:4]) # Output: ['it.wikipedia.org', 'docs.python.org', 'www.github.com']

# Rimozione di un elemento
bookmarks.remove("www.wikipedia.org")
```

I dizionari

I dizionari contengono coppie chiave-valore e sono indicizzati tramite le loro chiavi.

```
inglese_italiano = {
    "hello": "ciao",
    "world": "mondo",
    "random": "casuale"
}

parola_inglese = "random"
print(f"La traduzione di '{parola_inglese}' è '{inglese_italiano[parola_inglese]}'")
# Output: La traduzione di 'random' è 'casuale'
```

I dizionari sono mutabili: è possibile aggiungere, modificare o rimuovere elementi.

Esempio avanzato di dizionario

```
modelloIA = {
    "nome": "Random Forest",
    "accuratezza": 78.5,
    "numero di feature": 2
}

print(f"Il nome del modello di machine learning è {modelloIA['nome']}")

# Aggiunta di un nuovo elemento (una lista)
modelloIA["features"] = ["lunghezza", "larghezza"]
print(modelloIA)

# Rimozione di un elemento
del modelloIA["accuratezza"]
print(modelloIA)
```

I valori in un dizionario possono essere di qualsiasi tipo, inclusi liste, tuple o altri dizionari.

La funzione input()

La funzione `input()` permette di ricevere input dall'utente tramite tastiera:

```
nome = input("Dimmi il tuo nome: ")
print(f"Ciao {nome}!")
```

La funzione `input()` restituisce sempre una stringa. Per ottenere valori numerici, è necessario convertire l'input:

```
anni = int(input("Quanti anni hai? "))
altezza = float(input("Quanto sei alto in metri? "))
print(f"Hai {anni} anni e sei alto {altezza} metri.")
```

Le funzioni `int()` e `float()` convertono rispettivamente in intero e in floating point.

Il controllo di flusso: selezione

Il **flusso** di un programma è l'ordine di esecuzione delle istruzioni. Il **costrutto di selezione** permette di modificare questo flusso in base a una condizione.

```
score = float(input("Inserisci lo score del tuo modello (0.0-1.0): "))

if score > 0.8:
    print("Complimenti, ottimo modello!")
elif (score > 0.6) and (score <= 0.8):
    print("Buon modello!")
else:
    print("Il modello necessita di miglioramenti.")
```

Il costrutto `if...elif...else` permette di selezionare il flusso del programma in base a condizioni booleane.

Operatori di confronto e logici

Operatori di confronto

- `==` uguale
- `!=` diverso
- `>` maggiore
- `<` minore
- `>=` maggiore o uguale
- `<=` minore o uguale

Operatori logici

- `and` (entrambe le condizioni devono essere vere)
- `or` (almeno una condizione deve essere vera)
- `not` (nega una condizione)

Attenzione: non confondere l'operatore di confronto `==` con l'operatore di assegnamento `=`.

L'indentazione in Python

In Python, l'indentazione è obbligatoria per identificare i blocchi di codice all'interno delle strutture di controllo.

```
if temperatura > 30:
    print("Fa caldo!")
    print("Bevi molta acqua.")
else:
    print("La temperatura è accettabile.")
    if temperatura < 10:
        print("Anzi, fa freddo!")
        print("Copriti bene.")
```

L'indentazione può essere fatta con spazi o tabulazioni. Convenzionalmente, si usano 4 spazi per ogni livello di indentazione.

Il carattere `:` indica sempre l'inizio di un blocco indentato.

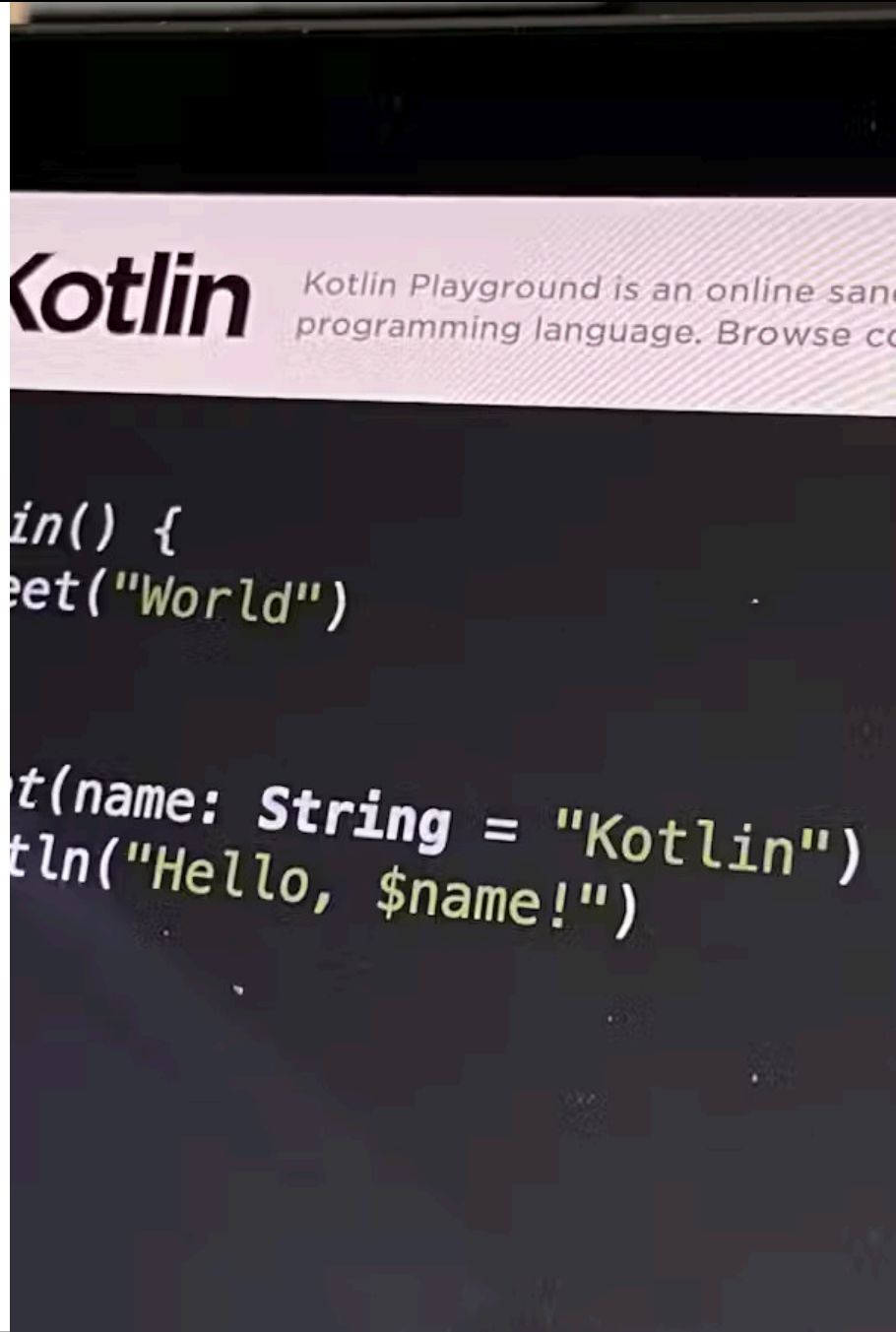
Il controllo di flusso: ciclo while

Il ciclo `while` esegue un blocco di codice fintanto che una condizione è vera:

```
score = 0.0
while score < 0.6:
    score = float(input("Inserisci un nuovo score (0.0-1.0): "))
    print(f"Score attuale: {score}")
print("Score accettabile raggiunto!")
```

Se la condizione è falsa all'inizio, il blocco di codice non viene mai eseguito.

Il corpo del ciclo è l'insieme delle istruzioni nel blocco indentato.



Il controllo di flusso: ciclo for

Il ciclo `for` permette di scorrere gli elementi di un contenitore ed effettuare azioni su di essi:

```
bookmarks = ["www.kaggle.com", "www.wikipedia.org",
             "docs.python.org", "www.github.com"]

# Ciclo for su una lista
for sito in bookmarks:
    print(f"Sito: {sito}, dominio: {sito[-4:]}")

# Ciclo for con enumerate()
for contatore, sito in enumerate(bookmarks):
    print(f"{contatore+1}. {sito}")
```

La funzione `enumerate()` restituisce simultaneamente indice ed elemento per tutti gli elementi della lista.

Ciclo for con dizionari

Per iterare su un dizionario, possiamo usare il metodo `items()`:

```
modello = {
 "nome": "Random Forest",
 "accuratezza": 78.5,
 "features": ["lunghezza", "larghezza"]
}

# Ciclo su chiavi e valori
for chiave, valore in modello.items():
 print(f"Chiave: {chiave}, Valore: {valore}")

# Ciclo solo sulle chiavi
for chiave in modello:
 print(f"Chiave: {chiave}, Valore: {modello[chiave]}")
```

La funzione range()

La funzione `range()` genera sequenze di numeri interi, utile nei cicli for:

```
# range con un argomento: da 0 a n-1
for i in range(5):
    print(i)  # Output: 0 1 2 3 4

# range con due argomenti: da start a stop-1
for i in range(5, 10):
    print(i)  # Output: 5 6 7 8 9

# range con tre argomenti: da start a stop-1 con passo step
for i in range(5, 10, 2):
    print(i)  # Output: 5 7 9
```

È possibile convertire un oggetto range in una lista:

```
lista = list(range(7))
print(lista) # Output: [0, 1, 2, 3, 4, 5, 6]
```

Controllo dei cicli: break, continue, pass

break

Interrompe il ciclo ed esce dallo stesso

```
for i in range(10):
    if i == 5:
        break
    print(i)  # Output: 0 1 2 3 4
```

continue

Salta alla prossima iterazione del ciclo

```
for i in range(5):
    if i == 2:
        continue
    print(i)  # Output: 0 1 3 4
```

pass

Non fa nulla, utile come segnaposto

```
for i in range(5):
 if i == 2:
 pass # Da implementare in futuro
 print(i) # Output: 0 1 2 3 4
```

Le funzioni in Python

Le funzioni sono sequenze di istruzioni identificate da un nome univoco, che possono essere richiamate nel programma.

```
def fibonacci(n):
    """
    Calcola e stampa i primi n numeri della sequenza di Fibonacci.
    La sequenza inizia con 0 e 1.
    """
    a, b = 0, 1
    for i in range(n):
        print(a, end=' ')
        a, b = b, a + b

# Chiamata della funzione
fibonacci(10)  # Output: 0 1 1 2 3 5 8 13 21 34
```

Le funzioni migliorano la modularità e la riusabilità del codice.

Funzioni con valore di ritorno

Le funzioni possono restituire valori utilizzando la parola chiave `return`:

```
def fibonacci(n):
    """
    Restituisce una lista con i primi n numeri della sequenza di Fibonacci.
    """
    fibonacci_seq = []
    a, b = 0, 1
    for i in range(n):
        fibonacci_seq.append(a)
        a, b = b, a + b
    return fibonacci_seq

# Chiamata della funzione
sequenza = fibonacci(10)
print(sequenza)  # Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Le variabili definite all'interno di una funzione hanno scope locale: sono visibili solo all'interno della funzione.

Parametri con valori predefiniti

È possibile assegnare valori predefiniti ai parametri di una funzione:

```
def fibonacci(n, a=0, b=1):
 """
 Restituisce una lista con i primi n numeri della sequenza di Fibonacci,
 partendo dai valori a e b.
 """
 fibonacci_seq = []
 for i in range(n):
 fibonacci_seq.append(a)
 a, b = b, a + b
 return fibonacci_seq

# Chiamate della funzione
seq1 = fibonacci(10) # Usa i valori predefiniti a=0, b=1
seq2 = fibonacci(10, 8, 13) # Usa i valori specificati a=8, b=13
```

Funzioni con valori di ritorno multipli

Una funzione può restituire più valori separati da virgole:

```
def calcola_somma_prodotto(a, b):
    """
    Calcola e restituisce sia la somma che il prodotto di due numeri.
    """
    somma = a + b
    prodotto = a * b
    return somma, prodotto

# Chiamata della funzione
s, p = calcola_somma_prodotto(5, 3)
print(f"Somma: {s}, Prodotto: {p}")  # Output: Somma: 8, Prodotto: 15
```

I valori restituiti possono essere assegnati a variabili separate utilizzando l'assegnazione multipla.

Funzioni lambda

Le funzioni lambda sono funzioni anonime definite in una sola riga:

```
moltiplicazione = lambda a, b: a * b
print(moltiplicazione(5, 3))  # Output: 15

# Funzione lambda con valori di ritorno multipli
calcola = lambda a, b: (a + b, a * b)
s, p = calcola(5, 3)
print(f"Somma: {s}, Prodotto: {p}")  # Output: Somma: 8, Prodotto: 15
```

Le funzioni lambda sono utili per definire funzioni semplici in modo conciso, spesso come argomenti di altre funzioni.

Lettura e scrittura di file

Scrittura su file

```
modello = {
    "nome": "Random Forest",
    "punteggio": 78.5,
    "numero di feature": 2
}

f = open("modello.txt", "w")
for chiave, valore in modello.items():
    f.write(f"{chiave}:{valore}\n")
f.close()
```

Lettura da file

```
modello = {}
f = open("modello.txt", "r")
for riga in f.readlines():
    elementi = riga.split(":")
    chiave = elementi[0]
    valore = elementi[1][:-1]
    modello[chiave] = valore
f.close()
print(modello)
```

La funzione `open()` accetta il nome del file e la modalità di apertura: `"r"` (lettura), `"w"` (scrittura), `"a"` (accodamento).

Lettura di dati tabulari da file CSV

I file CSV (Comma-Separated Values) sono un formato comune per salvare dati tabulari:

```
dati_marziani = {"tipo":[], "colore":[], "n_arti":[],
                 "peso":[], "altezza":[], "larghezza":[]}

file = open("marziani.csv", "r")
for riga in file.readlines():
    dati_linea = riga[:-1].split(",")
    if not((" " in dati_linea) or ("" in dati_linea) or
           ("specie" in dati_linea)):
        dati_marziani['tipo'].append(dati_linea[0])
        dati_marziani['colore'].append(dati_linea[1])
        dati_marziani['n_arti'].append(float(dati_linea[2]))
        dati_marziani['peso'].append(float(dati_linea[3]))
        dati_marziani['altezza'].append(float(dati_linea[4]))
        dati_marziani['larghezza'].append(float(dati_linea[5]))
file.close()
```

Questo approccio carica i dati colonna per colonna in un dizionario.

Programmazione orientata agli oggetti: le classi

Una **classe** è un modello astratto che combina dati (attributi) e comportamenti (metodi):

```
PI = 3.14159

def distanza(x1, y1, x2, y2):
    return ((x2-x1)**2 + (y2-y1)**2)**0.5

class Circonferenza:
    def __init__(self, x, y, raggio):
        self.x = x
        self.y = y
        self.raggio = raggio

    def stampa(self):
        print(f"Centro: ({self.x}, {self.y}), Raggio: {self.raggio}")

    def area(self):
        return PI * self.raggio**2

    def contiene_punto(self, px, py):
        return distanza(self.x, self.y, px, py) <= self.raggio

    def tangente(self, altra_circonferenza):
        d = distanza(self.x, self.y,
                     altra_circonferenza.x, altra_circonferenza.y)
        return abs(d - (self.raggio + altra_circonferenza.raggio)) < 0.001
```

Un **oggetto** è un'istanza specifica di una classe.