Polimorfismo in Python

Cos'è il polimorfismo?

Il significato letterale di polimorfismo è l'assumere forme, aspetti, modi di essere diversi secondo le varie circostanze.

Il polimorfismo è un concetto molto importante nella programmazione OO. Si riferisce all'uso di una singola entità di tipo (metodo, operatore o oggetto) per rappresentare diversi tipi, in diversi scenari.

Facciamo un esempio:

Es 1: polimorfismo in aggiunta all'operatore

Sappiamo che l'operatore **+** è ampiamente utilizzato nei programmi Python. Ma non ha un singolo utilizzo.

Per i tipi di dati interi, l'operatore **+** viene utilizzato per eseguire l'operazione di addizione aritmetica.

```
1 # Polimorfismo operatori - (sovraccarico degli operatori)
2 print("Polimorfismo operatori - (sovraccarico degli operatori)")
3 num1 = 1
4 num2 = 2
5 print(num1+num2)
```

Quindi, il programma di cui sopra produce 3.

Allo stesso modo, per i tipi di dati stringa, l'operatore **+** viene utilizzato per eseguire la concatenazione.

```
1 str1 = "Python"
2 str2 = "Programming"
3 print(str1+" "+str2)
```

Di conseguenza, il programma precedente genera "*Python Programming*".

Qui possiamo vedere che un singolo operatore **+** è stato utilizzato per eseguire diverse operazioni per diversi tipi di dati. Questa è una delle occorrenze più semplici di polimorfismo in Python.

Polimorfismo delle funzioni in Python

Ci sono alcune funzioni in Python che sono compatibili per essere eseguite con più tipi di dati.

Una di queste funzioni è la funzione len(). Può funzionare con molti tipi di dati in Python. Diamo un'occhiata ad alcuni casi d'uso di esempio della funzione.

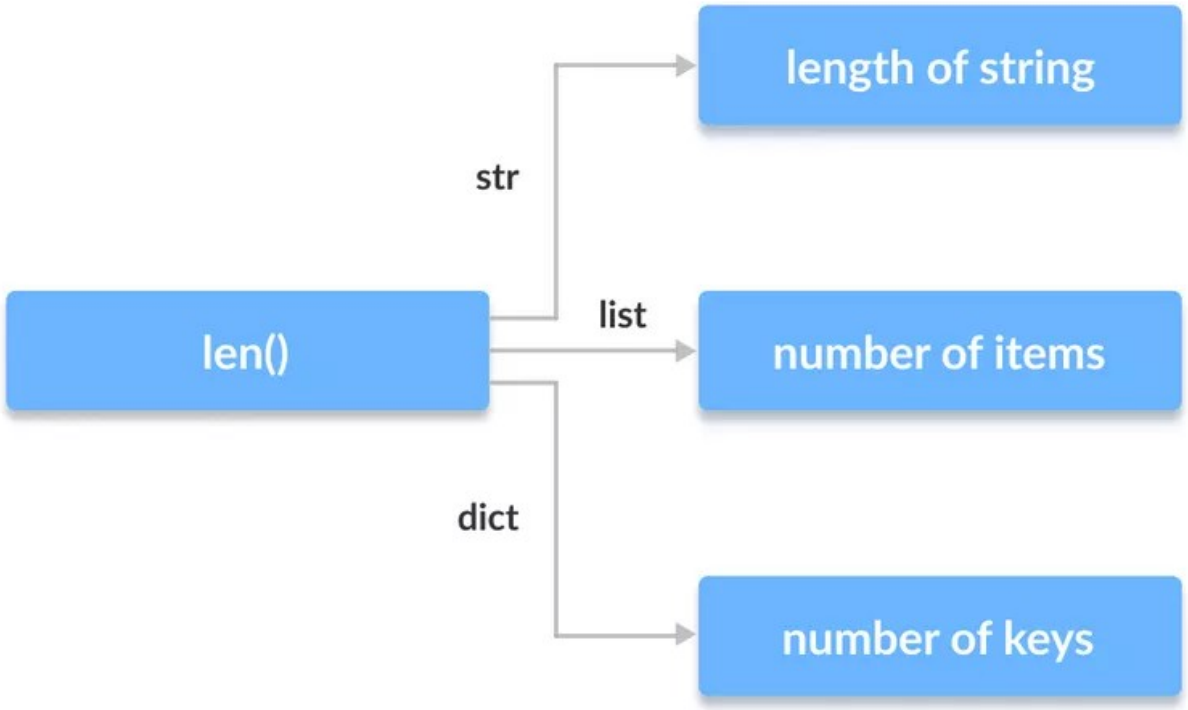
Es 2: funzione polimorfica len()

```
1 # polimorfismo delle funzioni
2 print("Polimorfismo delle funzioni")
3 print(len("Programiz"))
4 print(len(("Python", "Java", "C")))
5 print(len({"Name": "John", "Address": "Nepal"}))
```

Out

```
9
3
2
```

Qui, possiamo vedere che molti tipi di dati come string, list, tuple, set e dictionary possono funzionare con la funzione **len()**. Tuttavia, possiamo vedere che restituisce informazioni specifiche su tipi di dati specifici.



Polimorfismo nella funzione **len()** in Python.

Polimorfismo di classe in Python

Il polimorfismo è un concetto molto importante nella programmazione orientata agli oggetti.

Possiamo usare il concetto di polimorfismo durante la creazione di metodi di classe poiché Python consente a classi diverse di avere metodi con lo stesso nome.

Successivamente possiamo generalizzare la chiamata a questi metodi ignorando l'oggetto con cui stiamo lavorando.

Diamo un'occhiata a un esempio:

Es 3: polimorfismo nei metodi di classe

```
1  # Polimorfismo di classe in Python
2  print("Polimorfismo di classe in Python")
3
4  class Cat:
5      def __init__(self, name, age):
6              self.name = name
7              self.age = age
8
9      def info(self):
10             print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")
11
12     def make_sound(self):
13             print("Meow")
14
15 class Dog:
16     def __init__(self, name, age):
17             self.name = name
18             self.age = age
19
20     def info(self):
21             print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")
22
23     def make_sound(self):
24             print("Bau")
25
26 cat1 = Cat("Kitty", 2.5)
27 dog1 = Dog("Fluffy", 4)
28 tuplanimali=(cat1, dog1)
29 for animal in tuplanimali:
30     animal.make_sound()
31     animal.info()
32     animal.make_sound()
```

Out

```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bau
I am a dog. My name is Fluffy. I am 4 years old.
Bau
```

Qui abbiamo creato due classi **Cat** e **Dog**. Condividono una struttura simile e hanno gli stessi nomi di metodo **info()** e **make_sound()**.

Tuttavia, nota che non abbiamo creato una superclasse comune o collegato le classi in alcun modo. Anche allora, possiamo impacchettare questi due diversi oggetti in una tupla e iterare attraverso di essa usando una variabile animale comune. È possibile grazie al polimorfismo.

Polimorfismo ed ereditarietà

Come in altri linguaggi di programmazione, anche le classi figlie in Python ereditano metodi e attributi dalla classe genitore. Possiamo ridefinire alcuni metodi e attributi specificatamente per adattarli alla classe figlia, nota come **Method Overriding**.

Il polimorfismo ci consente di accedere a questi metodi e attributi sovrascritti che hanno lo stesso nome della classe genitore.

Diamo un'occhiata a un esempio:

Esempio 4: sostituzione del metodo

```
1
2  print("Polimorfismo - Method Overriding")
3
4  from math import pi
5
6  class Forma:
7
8      def __init__(self, nome):
9              self.nome = nome
10
11     def area(self):
12             pass
13
14     def definizione(self):
15             return "Sono una Forma bidimensionale."
16
17     def __str__(self):
18             return self.nome
19
20
21 class Quadrato(Forma):
22
```

```
23      def __init__(self, lato):
24              super().__init__("Quadrato")
25              self.lato = lato
26
27      def area(self):
28              return self.lato**2
29
30      def definizione(self):
31              return "I Quadrati hanno tutti angoli di 90° e lati uguali."
32
33
34  class Cerchio(Forma):
35
36      def __init__(self, raggio):
37              super().__init__("Cerchio")
38              self.raggio = raggio
39
40      def area(self):
41              return pi*self.raggio**2
42
43
44  a = Quadrato(4)
45  b = Cerchio(7)
46  print(b)
47  print(b.definizione())
48  print(a.definizione())
49  print(b.area())
50
```

Out

```
Polimorfismo - Method Overriding
Cerchio
Sono una Forma bidimensionale.
I Quadrati hanno tutti angoli di 90° e lati uguali.
153.93804002589985
```

Qui, possiamo vedere che i metodi come **__str__()**, che non sono stati sovrascritti nelle classi figlie, vengono usati dalla classe genitore.

A causa del polimorfismo, l'interprete Python riconosce automaticamente che il metodo **definizione()** per l'oggetto **a** (classe Quadrato) è sovrascritto. Quindi, usa quello definito nella classe figlia (shadowing).

D'altra parte, poiché il metodo **definizione()** per l'oggetto **b** non è sovrascritto, viene utilizzato dalla classe genitore Forma.