# Python - ligação dinâmica

Na programação orientada a objetos, o conceito de ligação dinâmica está intimamente relacionado ao polimorfismo. Em Python, ligação dinâmica é o processo de resolução de um método ou atributo em tempo de execução, em vez de em tempo de compilação.

De acordo com o recurso de polimorfismo, objetos diferentes respondem de maneira diferente à mesma chamada de método com base em suas implementações individuais. Esse comportamento é obtido através da substituição de métodos, onde uma subclasse fornece sua própria implementação de um método definido em sua superclasse.

O interpretador Python determina qual é o método ou atributo apropriado a ser invocado com base no tipo do objeto ou na hierarquia de classes em tempo de execução. Isso significa que o método ou atributo específico a ser chamado é determinado dinamicamente, com base no tipo real do objeto.

## Exemplo

O exemplo a seguir ilustra a ligação dinâmica em Python -

```
class shape:
   def draw(self):
      print ("draw method")
      return

class circle(shape):
   def draw(self):
      print ("Draw a circle")
      return

class rectangle(shape):
   def draw(self):
      print ("Draw a rectangle")
      return

shapes = [circle(), rectangle()]
for shp in shapes:
   shp.draw()
```

Ele produzirá a seguinte **saída** -

```
Draw a circle
Draw a rectangle
```

Como você pode ver, o método draw() está vinculado dinamicamente à implementação correspondente com base no tipo do objeto. É assim que a ligação dinâmica é implementada em Python.

## Digitação de pato

Outro conceito intimamente relacionado à vinculação dinâmica é **a digitação de pato** . Se um objeto é adequado para um uso específico é determinado pela presença de certos métodos ou atributos, e não pelo seu tipo. Isso permite maior flexibilidade e reutilização de código em Python.

A digitação Duck é um recurso importante de linguagens de digitação dinâmica como Python (Perl, Ruby, PHP, Javascript, etc.) que se concentra no comportamento de um objeto e não em seu tipo específico. De acordo com o conceito de "digitação de pato", "Se ele anda como um pato e grasna como um pato, então deve ser um pato".

A digitação Duck permite que objetos de diferentes tipos sejam usados de forma intercambiável, desde que tenham os métodos ou atributos necessários. O objetivo é promover flexibilidade e reutilização de código. É um conceito mais amplo que enfatiza o comportamento e a interface dos objetos, em vez dos tipos formais.

Aqui está um exemplo de digitação de pato -

```
class circle:
   def draw(self):
      print ("Draw a circle")
      return

class rectangle:
   def draw(self):
      print ("Draw a rectangle")
      return

class area:
   def area(self):
      print ("calculate area")
      return

def duck_function(obj):
   obj.draw()

objects = [circle(), rectangle(), area()]
```

```
for obj in objects:
   duck_function(obj)
```

Ele produzirá a seguinte **saída** -

```
Draw a circle
Draw a rectangle
Traceback (most recent call last):
 File "C:\Python311\hello.py", line 21, in <module>
  duck_function(obj)
 File "C:\Python311\hello.py", line 17, in duck_function
 obj.draw()
AttributeError: 'area' object has no attribute 'draw'
```

A ideia mais importante por trás da digitação duck é que duck_function() não se importa com os tipos específicos de objetos que recebe. Requer apenas que os objetos tenham um método draw(). Se um objeto "grasna como um pato" por ter o comportamento necessário, ele é tratado como um "pato" com o propósito de invocar o método draw().

Assim, na digitação duck, o foco está no comportamento do objeto e não em seu tipo explícito, permitindo que diferentes tipos de objetos sejam usados de forma intercambiável, desde que exibam o comportamento necessário.