Python 3 magic methods

I metodi magici sono l'anima della programmazione ad oggetti in Python, iniziano e terminano con un doppio underscore, si dividono in diversi gruppi in merito ai loro scopi.

- Costruttori, Inizializzatori e Distruttori
- Operatori di confronto
- Operatori aritmetici normali
- Operatori aritmetici riflessi
- Operatori di assegnamento
- Operatori unari
- Conversione di tipi
- Rappresentazione della classe
- Controllo accesso agli attributi
- Contenitori
- Riflessioni
- Callable Obiects
- Context Managers (Gestori di contesto)
- Copia
- Pickling (Decapaggio, come mettere sotto aceto un oggetto)

Costruttori, Inizializzatori e Distruttori :

Esistono due costruttori in Python, __new__ ed __init__ :
__new__(cls,[....]) è il primo metodo ad essere chiamato se non è presente Python ne crea uno di default, per questo si usa raramente di solito se si implementa una sottoclasse di un oggetto non mutevole stringhe, tuple etc.
__init__(self,[....]) inizializza la classe, la maggior parte delle classi scritte hanno bisogno di una inizializzazione.

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Esempio con entrambi i metodi:

```
class test:

    def __new__(cls): # primo costruttore  __new__
        print('sono dentro __new__')
        self = object.__new__(cls) # definisce self
        return self # uscita corretta per l' inizializzare la classe
    @staticmethod
    def reverse(cls,testo):
        return testo[::-1]
    def __init__(self):   # secondo costruttore, inizializzazione  con __init__
        print ('sono dentro __init__')
        self.msg="Ciao Mondo"
    def write(self):
        print (self.msg)
```

```
a=test()
```

vedrete due messaggi, sono dentro __new__ , sono dentro __init__ (__new__ viene eseguito per primo e istanzia il metodo reverse, __init__ per secondo e istanzia il metodo write.

__del__(self) Se __new__ e __init__ sono i costruttori di un oggetto, __del__ è il distruttore l' implementazione di questo metodo definisce come deve avvenire la pulizia della memoria (garbage collected).
Il commando del (x) viene traslato come x.__del__().

Esempio di __init__ e __del__ in azione

```
from os.path import join

class File:
  '''Wrapper per un oggetto file, apertura, chiusura e cancellamento sicuro.'''

      def __init__(self, filepath='~', filename='esempio.txt'):
        # apre un file chiamato filename con percorso filepath in
        # lettura e scrittura
       self.file = open(join(filepath, filename), 'r+')

    def __del__(self):
      self.file.close()
      del self.file
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Operatori di confronto :

Uno dei migliori vantaggi nell'uso degli operatori magici è che essi provvedono a una semplice via per creare degli oggetti con caratteristiche identiche a quelli incorporati in Python.

__eq__(self, other)
Definisce l'implementazione dell'operatore, ==

__ne__(self, other)

Definisce l'implementazione dell'operatore, !=

__lt__(self, other)

Definisce l'implementazione dell'operatore, <
__gt__(self, other)

Definisce l'implementazione dell'operatore, >
__le__(self, other)

Definisce l'implementazione dell'operatore, <=
__ge__(self, other)

Definisce l'implementazione dell'operatore, >=

In questo esempio implementiamo la classe numerica byte, aggiungeremo le varie funzionalità a mano a mano.

```
class byte(int):
   def __init__(self,value):
      if isinstance(value,int): # controllo la validità del tipo
         self.value=abs(value) & 255 # solo interi range tra 0 e 255
      else:
         raise TypeError("Tipo non corretto") # Errore di tipo
  def __eq__(self,other):
      return self.value == other.value
   def __ne__(self,other):
       return self.value != other.value
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
    def __lt__(self,other):
        return self.value < other.value
    def __gt__(self,other):
        return self.value > other.value
    def __le__(self,other):
        return self.value <= other.value
    def __ge__(self,other):
        return self.value >= other.value
```

Testiamo la classe dall' interprete

```
>>> from byte import byte
>>> a=byte(5)
>>> b=byte(23)
>>> a == b
False
>>> a > b
False
>>> a >= b
False
>>> a == a
True
>>> a < b
True
>>> a <= b
True
>>> a != b
True
```

Operatori aritmetici normali :

Ora vediamo come implementare gli operatori matematici + , -, * e simili:

__add__(self, other)
Inmplementa l'addizione self + other

__sub__(self, other)
Implementa la sottrazione self - other

__mul__(self, other)
Implementa la moltiplicazione self * other

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

__floordiv__(self, other)
Implementa divisione intera self // other
__truediv__(self, other)
Implementa la divisione reale self / other ritorna un valore float

__mod__(self, other)
Implementa l'operatore modulo self % other

__divmod__(self, other)
Implementa divisione intera con modulo ritorna una tupla **(self // other, self % other)**

__pow__(self, other)
Implementa l'elevamento a potenza self **other

__lshift__(self, other)
Implementa l'operatore bitwise scorrimento a sinistra self << other

__rshift__(self, other)
Implementa l'operatore bitwise scorrimento a destra self >> other

__and__(self, other)
Implementa l'operatore bitwise and self & other

__or__(self, other)
Implementa l'operatore bitwise or self | other
__xor__(self, other)
Implementa l'operatore bitwise xor self ^ other

Implementazione dei metodi matematici :

Aggiungiamo alla classe byte I seguenti metodi:

```
def __add__(self, other):
    return byte(self.value + other.value)

def __sub__(self, other):
    return byte(self.value - other.value)

def __mul__(self, other):
    return byte(self.value + other.value)

def __floordiv__(self, other):
    return byte(self.value // other.value)
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
def __truediv__(self, other):
    return self.value / other.value) # ritorna un valore float

def __divmod__(self, other):
    return divmod(self.value , other.value) # ritorna (self // other, self % other)

def __pow__(self, other):
    return self.value ** other.value) # ritorna un valore int

def __lshift__(self, other):
    return byte(self.value << other.value)

def __rshift__(self, other):
    return byte(self.value << other.value)

def __and__(self, other):
    return byte(self.value & other.value)

def __or__(self, other):
    return byte(self.value | other.value)

def __xor__(self, other):
    return byte(self.value ^ other.value)
```

Testiamo la classe dall' interprete

```
>>> a=byte(5)
>>> b=byte(2)
>>> c = a + b
>>> c.value          per il momento accediamo al valore interno tramite value
7
>>> c          senza definire Il metodo __repr__ apparira qualcosa del genere
<byte.byte object at 0x0218F2B0>
>>> print( c )    in mancanza del metodo __str__
<byte.byte object at 0x0218F2B0>  rimandiamo per ora la loro implementazione

>>> c = a – b
>>> c.value
3
>>> a**b
25                il valore restituito è un intero
>>> divmod(5,2)
(2,1)             il valore restituito è una tupla
>>> a / b
2.5               il valore restituito è un float
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Operatori aritmetici riflessi :

Ogni operatore aritmetico a la sua forma speculare degli operatori:
forma normale: oggetto + altro oggetto
forma speculare: altro oggetto + oggetto

Di norma le due forme sono equivalenti se si tratta di due oggetti della stessa classe, ma se si tratta due oggetti di classe diversa. Es. un intero e un float o il tipo byte che abbiamo creato, questo cambia e Python deve sapere come gestirli.
i tipi riflessi hanno lo stesso nome dei tipi normali preceduto dal prefisso r.

__radd__(self, other)
Implementa l'addizione other + self

__rsub__(self, other)
Implementa la sottrazione other - self

__rmul__(self, other)
Implementa la moltiplicazione other * self

__rfloordiv__(self, other)
Implementa divisione intera other // self
__rtruediv__(self, other)
Implementa la divisione reale other / self ritorna un valore float

__rmod__(self, other)
Implementa l'operatore modulo other % self

__rpow__(self, other)
Implementa l'elevamento a potenza other ** self
__rlshift__(self, other)
Implementa l'operatore bitwise scorrimento a sinistra other << self
__rshift__(self, other)
Implementa l'operatore bitwise scorrimento a destra other >> self

__rand__(self, other)
Implementa l'operatore bitwise and a & b

__ror__(self, other)
Implementa l'operatore bitwise or a | b

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

__rxor__(self, other)
Implementa l'operatore bitwise xor a ^ b

Se come nella classe byte non sono previste operazioni con altri tipi es. interi, float non c'è bisogno di codificarli, se il tipo normale già prevede l'interazione con un altro tipo, non c'è bisogno di ripetere la codifica, basta utilizzare la seguente forma. es. __radd__=__add__ in questo caso quando Python chiamerà __radd__ eseguirà __add__.

Operatori di assegnamento :

Python possiede un numeroso set di operatori di assegnamento uno per ogni operatore matematico si riconoscono dal prefisso i davanti al nome,

__iadd__(self, other)
Implementa l'addizione self = self + other

__isub__(self, other)
Implementa la sottrazione self = self - other

__imul__(self, other)
Implementa la moltiplicazione self = self * other

__ifloordiv__(self, other)
Implementa divisione intera self = self // other
__itruediv__(self, other)
Implementa la divisione reale self = self / other ritorna un valore float

__imod__(self, other)
Implementa l'operatore modulo self = self % other

__ipow__(self, other)
Implementa l'elevamento a potenza self = self **other

__ilshift__(self, other)
Implementa l'operatore bitwise scorrimento a sinistra self = self << other

__irshift__(self, other)
Implementa l'operatore bitwise scorrimento a destra self = self >> other

__iand__(self, other)
Implementa l'operatore bitwise and self = self & other

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

__ior__(self, other)

Implementa l'operatore bitwise or self = self | other

__ixor__(self, other)

Implementa l'operatore bitwise xor self = self ^ other

__Implementiamoli:

```
def __iadd__(self, other):  #  equivale ad +=
    return  byte(self.value + other.value)

def  __isub__(self, other):  # equivale ad  -=
    return  byte(self.value - other.value)

def  __imul__(self, other):  # equivale ad  *=
    return  byte(self.value + other.value)
def __ifloordiv__(self, other):  # equivale ad  //=
    return  byte(self.value // other.value)

def __itruediv__(self, other):  # equivale ad  /=
    return  self.value  /  other.value) # ritorna un valore float

def __ipow__(self, other): # equivale ad  **=
    return  self.value  **  other.value)
def __ilshift__(self, other): # equivale ad  <<=
    return  byte(self.value << other.value)

def __irshift__(self, other): # equivale ad  >>=
    return  byte(self.value << other.value)

def __iand__(self, other): # equivale ad  &=
    return  byte(self.value & other.value)

def __ior__(self, other): # equivale ad  |=
    return  byte(self.value | other.value)

def __ixor__(self, other): # equivale ad  ^=
    return  byte(self.value ^ other.value)
```

Ognuno di questi metodi effettua l'operazione indicata dal nome e quindi il nuovo assegnamento.

```
>>> a=5
>>> a+=1  incrementa di uno la variabile a quindi a=6
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Operatori unari :

Gli operatori e le funzioni unari hanno un solo operando.

__pos__(self) Operatore unario positivo (+ qualche_oggetto)
__neg__(self) Operatore unario negativo (- qualche_oggetto)

__abs__(self) Implementa il comportamento della funzione di built in abs()
__invert__(self) Implementa il comportamento del operatore (bitwise) ~ inversione di bit
__round__(self, n) Implementa il comportamento della funzione di built- in round(), n numero di decimali

__floor__(self) implementa il comportamento della funzione di built in math.floor() arrotondamento sotto l'intero più vicino

__ceil__(self) implementa il comportamento della funzione math.ceil() arrotondamento sopra l'intero più vicino

__trunc__(self) implementa il comportamento della funzione math.trunc() arrotondamento alla parte intera

Conversione di tipi :

__int__(self) implementa la conversione di tipo a intero
__float__(self) implementa la conversione di tipo a float
__complex__(self) implementa la conversione di tipo a numero complesso
__oct__(self) implementa la conversione numerica ottale
__hex__(self) implementa la conversione numerica esadecimale

__index__(self)

Implementa la conversione di tipo in un int quando l'oggetto viene utilizzato in un'espressione di sezione. Se si definisce un tipo numerico personalizzato che potrebbe essere utilizzato in slicing, è necessario definire __index__.

implementiamo la conversione a __int__ e __float__:

```
def  __int__(self):
   return self.value
def  __float__ (self):
   return float(self.value)
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Rappresentazione della classe :

Spesso è utile avere la rappresentazione di una classe in formato stringa

__str__(self)) definisce il comportamento per quando str() è usata su un istanza della vostra classe. Rappresenta il valore in modo informale

__repr__(self) definisce il comportamento per quando repr() è usata su un istanza della vostra classe. Rappresenta ufficialmente la classe e quindi può essere utilizzata da una espressione Python valida es. b=eval('byte.byte(5)')
b viene istanziata come byte con valore 5

__bytes__(self) definisce il comportamento quando un bytes()è usata su un istanza della vostra classe. E' un nuovo tipo di built-in per la costruzione di array di byte.

__format__(self, formatstr) definisce il comportamento quando un istanza della vostra classe usa il nuovo tipo di formattazione di stringa

__hash__(self) definisce il comportamento per quando hash() è usata su un istanza della vostra classe. Ritorna un intero, il risultato è utilizzato per una chiave rapida di confronto in dizionari, da notare che abitualmente richiede l'implementazione di __eq__ con la seguente regola a==b implica hash(a)==hash(b)

__bool__(self) definisce il comportamento per quando bool() è chiamata su un istanza della vostra classe

__dir__(self) definisce il comportamento per quando dir() è chiamata su un istanza della vostra classe, questo metodo ritorna una lista di attributi per l'utente. Di solito l'implementazione di questo metodo non è necessaria, ma può diventare di vitale importanza per un uso interattivo se ridefinite __getattr__ o __getattribute__

Curiosità dir(byte) equivale ad list(byte.__dict__.keys())

__sizeof__(self) definisce il comportamento per quando **sys.getsizeof**() chiama un istanza della vostra classe, ritorna la grandezza di un oggetto in byte. Questo metodo è generalmente più utile per le classi Python implementate come estensione C.

Implementiamo questi due metodi fondamentali nella classe byte

```
def __str__(self):
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
    return str(self.value)

def __repr__(self):
    return "%s.%s(%d)" % (self.__class__.__module__, self.__class__.__qualname__, self.value)
```

Testiamo la classe con le nuove modifiche

```
>>> a=byte(34)
>>> print(a)  oppure str(a) da come risultato
'34'    valore informale

>>> a          premendo invio si ha come risultato
byte.byte(34)     rappresentazione di classe
```

Implementare __repr__ semplicemente come repr(self.valore) avrebbe restituito il valore 34 di classe int non di classe byte.
La classe byte si conclude qui quello che rimane da fare è rinominare le istanze self.value in self.__value .
Il doppio underscore nasconderà la variabile all'esterno, è utile in fase di test di una classe la visibilità di alcune variabili per testarne il comportamento, ma poi bisogna ricordarsi di blindarle tutte. Curiosità in una classe Python nasconde le variabili con un doppio underscore solo all'inizio, facendole precedere da un underscore più nome della classe es x=byte(5) x._byte__value, renderà nuovamente visibile la variabile che abbiamo nascosto. Se lo desiderate lascio a voi il compito di migliorarla la classe.

Controllo accesso agli attributi :

__getattr__(self, name) potete definire un comportamento per quando un utente tenta un accesso a un attributo che non esiste. Questo può essere particolarmente utile per catturare e rindirizzare errori di ortografia comuni, fornendo avvisi intorno all'uso di attributi deprecati (AttributeError).

__setattr__(self, name, value) diversamente da __getattr__ é una soluzione di incapsulamento. Permette di definire il comportamento di assegnamento di un attributo indipendentemente se quest' attributo esista o meno, vi permette di definire una regola per qualsiasi cambiamento nei valori di un attributo.

__delattr__(self) Questo metodo è esattamente uguale a __setattr__, ma per la cancellazione di attributi creati proprio da __setattr__ stesso, dovete avere le stesse

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

precauzioni prese con __setattr__ è bene accertarsi di prevenire infinite ricorsioni, (chiamando del(self.name) nell'implentazione di __delattr__ potrebbe causare infinite ricorsioni).

__getattribute__(self, name) Può essere usato solo con il nuovo stile di classi (Python 3.x) esso vi permette di definire delle regole per ogni volta si accede ai valori degli attributi, può causare infinite ricorsioni, per ovviare a questo avete bisogno a priori di __getattr__ , usate __getattribute__ solo per ricevere chiamate, se chiamato direttamente lancia un AttributeError.
Questo metodo può essere usato, ma non lo raccomando.
Esempi:

Ora vi presento due esempi, notate la differenza sostanziale tra __getattr__ e __getattribute__

```
class Gelato:
   def __getattr__(self, key):
      if key == 'gusto':
         return 'Pistacchio'
      else:
            raise AttributeError
```

```
>>> cono=Gelato()
```

```
>>> dir(cono)
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattr__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
```

```
>>> cono.gusto
```

```
'Pistacchio'
```

```
>>> gelato.gusto = ”Cioccolato”
```

```
>>> gelato.gusto
```

```
‘Cioccolato’
```

```
>>> dir(cono)
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattr__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'gusto']

Si è creato un nuovo attributo, e __getattr__ ritorna il suo valore.

Esempio con __getattribute__

```
class GelatoMaxi:
   def __getattribute__(self, key):
       if key == 'gusto':
          return 'Pistacchio'
       else:
        raise AttributeError
```

```
>>> maxiCono=GelatoMaxi()
>>> dir(maxiCono)
[]
>>> maxiCono.gusto
'Pistacchio'
>>> maxiCono.gusto='Cioccolato'
>>> maxiCono.gusto
'Pistacchio'
>>> dir(maxiCono)
[]
```

Non è cambiato nulla __getattribute__ vede gli attributi in maniera assoluta.

Vediamo un modo come implementare correttamente __getattr__, __setattr__, e __delattr__, nelle vostre classi.

```
class attr_ext(object):
   def __getattr__(self, name):
      if name in self.__dict__:
         return self.__dict__[name]
      else:
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
            raise AttributeError("Attributo non presente”)
    def __setattr__(self, name ,value):
        self.__dict__[name]=value
    def __delattr__(self,name):
        if name in self.__dict__:
            del(self.__dict__[name])
        else:
            raise AttributeError("Attributo non presente")
>>> auto=attr_ext()
>>> auto.colore='Rosso'
>>> auto.carburante='Benzina'
>>> auto.marca='Toyota'
>>> def write(testo): print(testo)
>>> auto.write=write
>>> dir(auto)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattr__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__',
'__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'carburante', 'colore',
'marca', 'write']
>>> auto.write('Caratteristiche')
Caratteristiche
>>> auto.colore
'Rosso'
>>> auto.carburante
'Benzina'
>>> auto.marca
'Toyota'
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Python ci mette a disposizione 3 metodi pronti a questo scopo e sono:

getattr(object, name [,default]) Ritorna il valore dell' attributo name nell'oggetto desiderato se non presente ritorna default (senza creare l'attributo) quindi potete utilizzarlo anche per vedere se l'oggetto supporta un certo attributo. getattr(auto,'colore','????') se l'attributo 'colore' non è presente ritorna '????', un valore molto improbabile da trovare. Se il valore di default non è stato definito e l'attributo non esiste si avrà una eccezione AttributeError.

setattr(obiect, name, value) Setta l'attributo name dell'oggetto desiderato con il valore di value (se non presente lo crea e gli assegna il valore value).

delattr(obiect, name) Cancella l'attributo name dall'oggetto se l'attributo non esiste si avrà una eccezione AttributeError .

Contenitori :

I contenitori si distinguono in immutabili (stringhe e tuple) e mutabili (liste e dizionari), ma tutti possiedono i metodi seguenti.

__len__(self) Ritorna la lunghezza del contenitore,

__getitem__(self, key) Ritorna il valore associato alla chiave,

__setitem__(self, key, value) Setta la chiave key con il valore value **(tipi mutevoli)**

__delitem__(self, key) Elimina la chiave

__iter__(self) Ritorna un iteratore per il contenitore

__reversed__(self) Ritorna un iteratore contrario (da fine a inizio)

__contains__(self, key) Implementa il controllo sull'appartenenza al contenitore tramite in e not in ritorna True se appartiene, altrimenti False

__missing__(self, key) missing è usato in subclassi di dizionari, esso definisce l'implementazione di un metodo per ogni qualvolta si accede ad una chiave che non esiste.

Come contenitore di esempio creeremo una classe che simuli gli array del Basic con qualche estensione più Pythonica.

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

Python a già un ottima classe array ma gli indici degli array e di tutti gli altri tipi di contenitori partono sempre dallo 0, mentre nel Basic si può decidere da quale indice partire e possono essere anche negativi.

Caratteristiche della classe:

Nome dim la classe viene chiamata come il commando Basic che serve a dichiararli. Si possono creare array di tutti i tipi di built-in di Python.

Sintassi: array=dim(tipo, linf, lsup ,data=[])

tipo : stringa con il tipo di array : 'integer' , 'float' , 'complex' , 'bool' , 'any' (qualsiasi tipo)
linf : limite inferiore dell'array
lsup : limite superiore dell'array
data :lista di valori per l'inizializzazione dell'array, possono essere minori della lunghezza dell'array i rimanenti vengono settati con valori di default

Presenterò prima la classe completa e poi andrò ad analizzarla, la classe è ricca di doc string questo permetterà alla funzione help di mostrare una documentazione più accurata.

```
class dim:
    """
        Class     : dim
        Scopo     : creare degli array tipo Basic con indici personalizzabili
        Autore    : Marco Salvati
        Email     : salvatimarco61@gmail.com
        Data      : 2019-11-28
        Licenza   : GPL v.3
    """
    def __init__(self,atype,start,end,data=[]):
        """ atype tipo di array: integer, float, string, complex, bool, any
            any accetta tutti i tipi.
            Esempio d'uso: a=dim('integer',-7,7)
           Crea un array di interi con indici che vanno da -7 a 7
```

```
        """
        self.__atype=atype.upper()
        self.__start=start
        self.__end=end
        self.__range=range(self.__start,self.__end+1)
        self._posti=len(self.__range)
        self.__valori_iniziali={"INTEGER":0,"FLOAT":0.0,"COMPLEX":(0+0j),"STRING":"",
            "BOOL":False,"ANY":None}
        self.__array=[(self.__valori_iniziali[self.__atype])]* self._posti
        for i in range(len(data)): # Se sono presenti dei valori da settare
            self.__array[i] = data[i]
    def LBound(self):
        """ Ritorna il limite inferiore dell'array """
        return self.__start
    def UBound(self):
        """ Ritorna il limite superiore dell'array """
        return self.__end
    def __getitem__(self,item):
        if isinstance(item,slice):
            if item.start is None:
                start=0
            else:
                start=self.__range.index(item.start)
            if item.stop is None:
                stop=len(self.__array)
            else:
                stop=self.__range.index(item.stop)
            if item.step is None:
                step=1
```

```
            else:
                step=item.step
            return self.__array[start:stop:step]
        elif isinstance(item,int):
            if not item in self.__range :
                raise ValueError("Indice fuori scala")
            return self.__array[self.__range.index(item)]
        else:
            raise KeyError("L' item deve essere intero o slice")
    def __setitem__(self,item,value):
        if isinstance(item,slice):
            for v in value:
             if not self.__is_valid_value(v):
                 raise ValueError("il Valore richiesto deve essere {}".format(self.__atype))
            if item.start is None:
                start=0
            else:
                start=self.__range.index(item.start)
            if item.stop is None:
                stop=len(self.__array)
            else:
                stop=self.__range.index(item.stop)
            if item.step is None:
                step=1
            else:
                step=item.step
            self.__array[start:stop:step]=value
        elif isinstance(item,int):
            if not item in self.__range :
```

```
            raise ValueError("Indice fuori scala")
        if self.__is_valid_value(value):
            self.__array[self.__range.index(item)]=value
        else:
            raise ValueError("il Valore richiesto deve essere {}".format(self.__atype))
    def __is_valid_value(self,value):
        """ Ritorna True se il valore è del tipo corretto """
        if self.__atype=="ANY":
            return True
        if self.__atype=="INTEGER":
            if isinstance(value,int):
                return True
            else:
                return False
        elif self.__atype=="FLOAT":
            if isinstance(value,float):
                return True
            else:
                return False
        elif self.__atype=="COMPLEX":
            if isinstance(value,complex):
                return True
            else:
                return False
        elif self.__atype=="STRING":
            if isinstance(value,str):
                return True
            else:
                return False
```

```
        elif self.__atype=="BOOL":
            if isinstance(value,bool):
                return True
            else:
                return False
        else:
            return False
    def index(self,*args):
        """Ritorna l'indice relativo a un valore dell'array:
           Esempio su un array di interi:
           a.index(88) ritorna l'indice del valore 88
           a.index(88,3) ritorna l'indice del valore 88 a partire dall'indice 3
           a.index(88,3,5) ritorna l'indice del valore 88 nel range indici da 3 a 5 """
        if len(args)==1:
            ir=self.__array.index(args[0])
            return self.__range[ir]
        elif len(args)==2:
            s=self.__range.index(args[1])
            ir=self.__array.index(args[0],s)
            return self.__range[ir]
        elif len(args)==3:
            s=self.__range.index(args[1])
            e=self.__range.index(args[2])
            ir=self.__array.index(args[0],s,e)
            return self.__range[ir]
        else:
            raise ValueError('il valore non è presente')
    def clear(self):
        """ azzerra l'array """
```

```
        for  i in range(len(self.__array)):
                self.__array[i]=(self.__valori_iniziali[self.__atype])
    def __str__(self):
        """array to string"""
        return str(self.__array)
    def __repr__(self):
        """implementazione metodo repr"""
        return "%s.%s('%s',%d,%d,data=%s)" % (self.__class__.__module__,
                                              self.__class__.__qualname__,self.__atype,
                                              self.__start,self.__end,repr(self.__array))
    def __iter__(self):
        "Ritorna un iteratore su i valori"
        return iter(self.__array
    def __contains__(self, key):
        """operatore in"""
        return key in self.__array
    def sort(self, reverse=False):
        "Ordina l'array"
        self.__array.sort(reverse=reverse)
    def reverse(self):
        """ inverte l'array """
        self.__array.reverse()
    def __reversed__(self):
        """ Ritorna un iteratore con l'array invertito"""
        return reversed(self.__array)
    def count(self,value):
        """ Conta il numero di ripetizioni di un valore"""
        return self.__array.count(value)
    def keys(self):
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
        """ Ritorna un iteratore sugli indici """
        return self.__range
    def append(self,item=1):
        """ Aggiunge altri posti nell'array default 1 posto"""
        for i in range(item):
            self.__array.append(self.__valori_iniziali[self.__atype])
        self.__end+=item
        self.__range=range(self.__start,self.__end+1)
    def __len__(self):
        """ Ritorna la lunghezza dell'array"""
        return len(self.__array)
```

Ora vediamo i vari metodi implementati:

__init__ è abbastanza intuibile si fanno copia dei dati e si crea una lista self.__array, con i valori di default della lunghezza richiesta dagli indici e se presente data sovrascrive i dati,
l'unica cosa degna di nota la riga con self.__range=range(self.__start,self.__end+1) questa riga crea un range degli indici richiesti, questo trucco permette di accedere a self.array, in quanto ogni indice reale di range corrisponde a quello di self.__array:

LBound Nel Basic questa funzione ritorna il limite inferiore dell'array

UBound Nel Basic questa funzione ritorna il limite inferiore dell'array vediamo come self.__range[indice_reale] ritorna il falso_indice associato alla posizione interessata, mentre self.__range.index(falso_indice) ritorna l' indice_reale

__getitem__ Questo metodo è quello che permette di riportare un valore o parte dell'array, vediamo come:

La prima riga if isinstance(item,slice): inizia con un controllo sul tipo di oggetto e per sapere quale sarà l'azione da intraprendere. Ci sono 3 soluzioni possibili:
1- l'oggetto è uno slice (questi oggetti contengono l'informazione di quale parte del contenitore prendere, es a="In quel ramo del lago di Como" a[:7] ritorna "in quel", gli slice possiedono 3 proprietà a sola lettura start parte iniziale, stop parte finale, step passo

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

2- l'oggetto è un int ed è nel range, l' indice è valido, altrimenti errore "Indice fuori scala"
3- se l'oggetto non nè slice né int segnala un KeyError

__setitem__ Questo metodo è il complementare di __getitem__ setta uno più valori nell'array (l'assegnamento in caso di slice i valori a destra del segno uguale devono corrispondere alle posizioni indicate, a[2:6]=33,55,66 lo slice indica 3 posti da 2 a 6 escluso, 3 valori da settare

Tutte le altri metodi sono già documentati e il codice è abbastanza semplice.

Testiamo la classe :

```
>>>  import dim
>>> Dim=dim.dim
>>>  a=Dim('integer',-4,11)
>>>  a
dim.dim('INTEGER',-4,11,data=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>>  a[-4]=66
>>> print(a)
'[66, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]'
>>> a.LBound
-4
>>> a.UBound()
11
>>> len(a)
16
>>> a[-3:2] = 4,67,55,7
>>> a
dim.dim('INTEGER',-4,11,data=[66, 4, 67, 55, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
>>>  a[11] = 99
>>>  a.append(3)
>>>  a
dim.dim('INTEGER',-4,11,data=[66, 4, 67, 55, 7, 0, 0, 0, 0, 0, 0, 0, 0, 0, 99,0,0,0])
```

Provate le altre funzionalità della classe.

Callable Objects :

Permette a una istanza di classe di essere eseguita come una funzione.

__call__ (self [,*args]) Può essere particolarmente utile in classi con istanze che hanno spesso bisogno di cambiare stato. Chiamare l'istanza può essere una via elegante e intuitiva.

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
>>> class ripeti
        def  __init__(self , msg, num_ripetizioni):
            self.msg=msg
             self.nrip=num_ripetizioni
        def __call__(self, num_ripetizioni):
           self.nrip=num_ripetizioni
        def dump(self):
             print(self.msg*self.nrip)


 >>> rip=ripeti('Ciao ',3)
>>> rip.dump()
Ciao Ciao Ciao
>>> rip(5)
>>> rip.dump()
Ciao Ciao Ciao Ciao Ciao
```

Context Managers (Gestori di Contesto):

I context managert permettono l'avvio e la chiusura pulita di un'azione, per gli, oggetti, quando questi utilizzano un'istruzione **with**. Il comportamento dei gestori di contesto è determinato da due metodi.

__enter__ (self) definisce l'inizio di un blocco creato con **with** statement.
se __enter__ ritorna la sua istanza o quella di un altro oggetto, possiamo assegnarla con la clausola **as** ad una nuova etichetta.

__exit__ (self, exception_type, exception_value, traceback)
definisce cosa il context manager farà al termine dell'esecuzione del blocco __enter__ ho in caso di errore.

Esempio :

```
class myFile:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):

      self.my_file = open(self.filename, self.mode)

      return self.my_file
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
    def __exit__(self, exception_type, exception_value, traceback):
        self.my_file.close()

with myFile('test.txt', 'w') as t:
    for  in range(10):
        t.write('riga  %d' % i)
```

Copia :

__copy__ (self) : Definisce il comportamento del metodo copy.copy(), con un istanza di una tua classe, ritorna una copia superficiale di una tua classe (shallow copy)
__deepcopy__ (self, memodict={}) : Definisce il comportamento del metodo copy.deepcopy(), con un istanza di una tua classe, ritorna una copia profonda di una tua classe (deep copy). Memodict è una cache di oggetti precedentemente copiati: questo ottimizza la copia e impedisce la ricorsione infinita durante la copia di strutture di dati ricorsive.

Picking (Decapaggio):

Non si applica solo ai tipi di built-in , ma per ogni classe che segue il protocollo pinckle. Questo protocollo a quattro metodi opzionali.

__getnewargs__ (self) : Ritorna una tupla di argomenti passati in __new__

__getstate__ (self) : Ritorna lo stato dell'oggetto per essere salvato

__setstate__ (self, state) : Ripristina lo stato di un oggetto

__reduce__ (self) : Quando definisci un tipo di estensione, (ovvero i tipi implementati utilizzando l'API C di Python), devi dire a Python come decaparlo, se si desidera farlo (ha la stessa funzione __getstate__).

Esempio:

Creiamo una struttura volutamente complessa e salviamo il suo stato per poi ripristinarlo in un nuovo oggetto.

class nucleo:

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
    "Nodo"
    def __init__(self):
        self._choises=dict() # informazione
        self.Back=None      # nodo precedente
        self.Next=None      # prossimo nodo


class neurone:
""" Classe       : neurone
    Scopo        :  Classe di memorizzazione dati associativa
    Autore       :  Marco Salvati
    Email        :  salvatimarco61@gmail.com
    Data         :  2019-03-05
    Licenza      :  GPL v.3 """
    def __init__(self):
        self.__root=nucleo()
    def new_choise(self,choise):
        """ Crea una nuova voce nel nodo"""
        self.__root._choises[choise]=None
    def choise_valueSet(self,choise,value):
        """ Setta o cambia il valore di una voce nel nodo"""
        self.__root._choises[choise]=value
    def choise_valueGet(self,choise):
        """  ritorna il valore di una nuova voce nel nodo"""
        return self.__root._choises[choise]
  def keys(self):
     """ Ritorna iteratore sulle chiavi presenti nel nodo"""
    return self.__root._choises.keys()
  def items(self):
       """ Ritorna iteratore sulle chiave e i valori  presenti nel nodo"""
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
        return self.__root._choises.items()

    def values(self):
        """ Ritorna iteratore sui valori presenti nel nodo"""

        return self.__root._choises.values()

    def sub_choise(self,choise,Lchoise):
        """ Setta una lista di sotto chiavi nel nodo"""

        d=dict.fromkeys(Lchoise)

        self.__root._choises[choise]=d

    def sub_choise_valueSet(self,choise,subchoise,value):
        """ Setta il valore di una sotto chiave nel nodo"""

        d=self.__root._choises[choise]

        d[subchoise]=value

    def sub_choise_valueGet(self,choise,subchoise):
        """ Ritorna il valore di una sotto chiave nel nodo"""

        d=self.__root._choises[choise]

        return d[subchoise]

    def del_choise(self,choise):
        """ Elimina una chiave nel nodo"""

    del self.__root._choises[choise]

    def del_sub_choise(self,choise,subchoise):
        """ Elimina una sotto chiave nel nodo"""

        d=self.__root._choises[choise]

        del d[subchoise]

    def sub_keys(self,choise):
        """ Ritorna un iteratore delle sotto chiavi presenti nel nodo"""

        return self.__root._choises[choise].keys()

    def sub_values(self,choise):

        """ Ritorna un iteratore sui valori delle sotto chiavi presenti nel nodo"""
        return self.__root._choises[choise].values()

    def sub_items(self,choise):
        """ Ritorna un iteratore coppia sotto chiavi - valori  presenti nel nodo"""

        return self.__root._choises[choise].items()
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
def new_node(self):
    """ Crea un nuovo nodo"""
      n=nucleo()
      while self.__root.Next!=None: self.next()
      self.__root.Next=n
      n.Back=self.__root
      self.__root=n
 def next(self):
       """ Vai al nodo successivo"""
      if self.__root.Next is not None: self.__root=self.__root.Next
  def back(self):
        """ Vai al nodo precedente"""
          if self.__root.Back is not None: self.__root=self.__root.Back
  def root(self):
       """ Vai al nodo radice"""
     while self.__root.Back!=None: self.back()
  def __getstate__ (self) :
      """ Ritorna l'oggetto per essere salvato"""
      posiz=self.__root
      self.root()
      buff=[]
      while True:
         buff.append(self.__root._choises)
         if self.__root.Next is not None:
            self.next()
         else:
            break
      self.__root=posiz
      return buff
  def __setstate__ (self,stato):
    """ Ripristina l'oggetto salvato"""
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
        self.__root=nucleo()
        conta=0
        for i in stato:
            self.__root._choises=stato[conta]
            conta+=1
            self.new_node()
        self.root()
```

test classe neurone

```
import pickle as pk
from neurone import neurone
n=neurone()
n.new_choise('Acqua')
n.new_choise('Stato')
n.new_choise('Composizione')
n.choise_valueSet('Acqua','elemento necessario alla vita')
n.choise_valueSet('Composizione',"La molecola dell'acqua é composta di due atomi di idrogeno e uno di ossigeno")
n.sub_choise('Stato',['Liquido','Gassoso','Solido','Cristallino'])
n.sub_choise_valueSet('Stato','Liquido','da 1 a 100 gradi centigradi')
n.sub_choise_valueSet('Stato','Gassoso','Sopra i 100  gradi centigradi ')
n.sub_choise_valueSet('Stato','Solido','Temperature sotto lo zero (ghiaccio)')
n.sub_choise_valueSet('Stato','Cristallino',' a zero gradi centigradi (neve)')
d={'Colori':['Giallo','Rosso','Blu'],'Temperature':['Kelvin','Centigradi','Farenaith']}
n.new_node()
n.new_choise('Varie')
n.choise_valueSet('Varie',d)
def nprint(n)
   n.root()
```

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA

```
    print("L'acqua : ",n.choise_valueGet('Acqua'))
    print (n.choise_valueGet('Composizione'))
    for stato in ['Liquido','Gassoso','Solido','Cristallino']:
        print ('Stato %s' % stato,n.sub_choise_valueGet('Stato',stato))
    print ("\n"*3+"Nuovo nodo")
    n.next()
    for stato in ['Colori','Temperature']:
        print ('Chiave =  %s' % stato,n.sub_choise_valueGet('Varie',stato))
    print ("\n"*3+'Nodo precedente')
    n.back()
    print ("L'acqua : ",n.choise_valueGet('Acqua'))
    print (n.choise_valueGet('Composizione'))
    for stato in ['Liquido','Gassoso','Solido','Cristallino']:
        print ('Stato %s' % stato,n.sub_choise_valueGet('Stato',stato))
nprint(n)
# Salvo l'oggetto in un buffer
buff=pk.dumps(n)
print ('\n@@@@@Oggetto Ricaricato@@@@@@@@\n')
x=pk.loads(buff)
nprint(x)
```

Termina qui questa guida sui Python magic method.

Autore : Marco Salvati

Email : salvatimarco61@gmail.com

Licenza : CC BY-SA