

Classi - Accesso agli attributi

Abbiamo visto che un oggetto può avere più attributi:

- self
- di classe
- ereditati

Alcuni possono sovrascrivere od oscurare altri.

Come capire a quale attributo si sta accedendo?

Classi - Accesso agli attributi

Abbiamo visto che un oggetto può avere più attributi:

- self
- di classe
- ereditati
- metodi che consentono accesso “fake” agli attributi (`__getattr__`)

Alcuni possono sovrascrivere od oscurare altri.

Come capire a quale attributo si sta accedendo?

Prima i metodi o prima gli attributi?

Classi - Accesso agli attributi

A livello di eredità fa fede l'attributo del livello più basso (come in altri linguaggi)

Quindi quando si accede un attributo prima si guarda se vi è nella classe figlia e solo se non vi è si passa alla classe madre. (Attributi di classe)

Per gli attributi di istanza dipende da cosa fanno i vari `__init__` e da come vengono chiamati.

Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        self.a = 2
```

```
        super().__init__()
```

Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        self.a = 2
```

```
        super().__init__()
```

```
x -> Cane
```

```
y -> Animale
```

```
a -> Animale
```

```
b -> Animale
```

Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.a = 2
```

Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        super().__init__()
```

```
        self.a = 2
```

```
x -> Cane
```

```
y -> Animale
```

```
a -> Cane
```

```
b -> Animale
```

Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        self.a = 2
```


Classi - Accesso agli attributi

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.a = 1
```

```
        self.b = 1
```

```
class Cane(Animale):
```

```
    x = 10
```

```
    def __init__(self):
```

```
        self.a = 2
```

x -> Cane

y -> Animale

a -> Cane

b -> Error

Classi - Accesso agli attributi

Tra gli attributi di istanza e quelli di classe viene data priorità a quelli di istanza.

```
class Animale:
```

```
    x = 0
```

```
    y = 0
```

```
    def __init__(self):
```

```
        self.x = 1
```

```
        self.b = 1
```

```
    x -> self
```

```
    y -> class
```

```
    b -> self
```

Classi - Accesso agli attributi

Tra gli attributi di istanza e quelli di classe viene data priorità a quelli di istanza.

In realtà la documentazione dice “A class instance has a namespace implemented as a dictionary which is the first place in which attribute references are searched. When an attribute is not found there, and the instance’s class has an attribute by that name, the search continues with the class attributes”

Questo namespace è l’attributo speciale **`__dict__`**

Classi - Accesso agli attributi

Questo namespace è l'attributo speciale **__dict__**

In **__dict__** vi sono tutti gli attributi di istanza.

Ogni volta che faccio `self.x = 1` esso viene inserito nel **__dict__**

(anche nell'**__init__**)

```
a = Animale()
```

```
a.a = 1
```

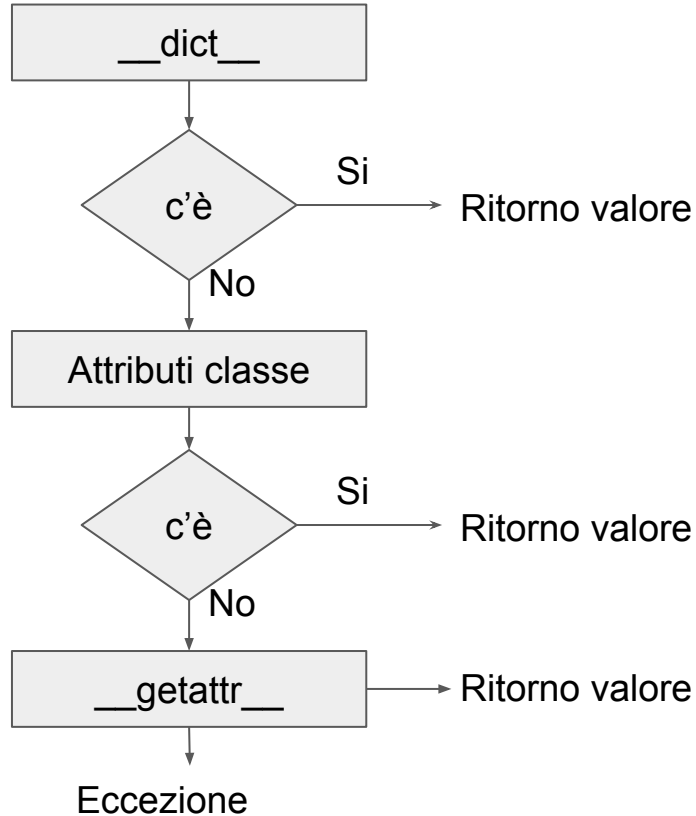
```
a.__dict__
```

Classi - Accesso agli attributi

Se un attributo non viene trovato né in `__dict__` né tra gli attributi di classe si passa al metodo speciale `__getattr__`.

`__getattr__` di default solleva un'eccezione ma può essere ridefinito dall'utente.

Classi - Accesso agli attributi



Classi - Accesso agli attributi

Ricordatevi l'esempio del saldo. Perché funzionava?

```
class Prova:
```

```
    def __getattr__(self, campo):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non accessibile')
```

```
        else:
```

```
            return super().__getattr__(campo)
```

Classi - Accesso agli attributi

```
class Prova:

    def __getattr__(self, campo):

        if campo == 'saldo':

            raise AttributeError('campo non accessibile')

        else:

            return super().__getattr__(campo)
```

Saldo non è né un attributo di classe né inizializzato in `__init__`

Classi - Accesso agli attributi

se volessi avere veramente il campo, non si arriverebbe mai all'esecuzione di `__getattr__`

```
class Prova:
```

```
    def __init__(self):
```

```
        self.saldo = 0
```

```
    def __getattr__(self, campo):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non accessibile')
```

```
        else:
```

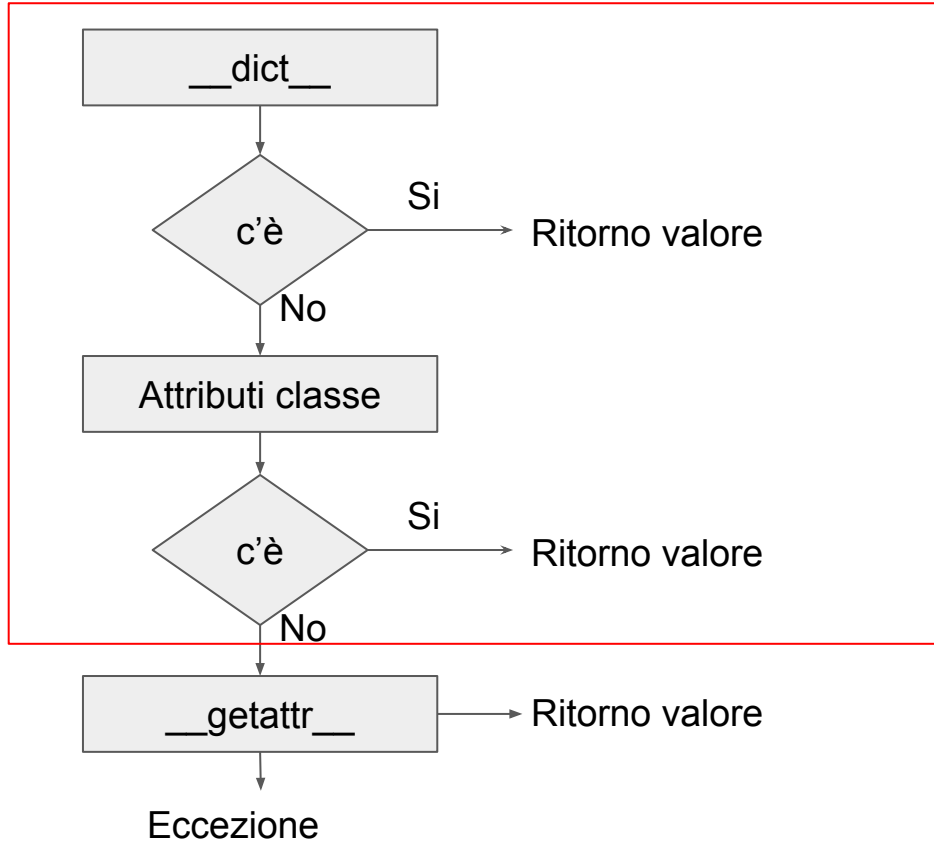
```
            return super().__getattr__(campo)
```

Classi - Accesso agli attributi

In realtà si passa per il metodo **__getattrute__** che di fatto controlla nel **__dict__** e negli attributi di classe.

Classi - Accesso agli attributi

__getattr__



Classi - Accesso agli attributi

In realtà si passa per il metodo **__getattr__** che di fatto controlla nel **__dict__** e negli attributi di classe.

Anche esso può essere personalizzato.

Classi - Accesso agli attributi

dovrei usare `__getattrute__`

```
class Prova:
```

```
    def __init__(self):
```

```
        self.saldo = 0
```

```
    def __getattrute__(self, campo):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non accessibile')
```

```
        else:
```

```
            return super().__getattrute__(campo)
```

Classi - Accesso agli attributi

`__getattribute__` viene sempre invocato quando si accede ad un attributo.

Tranne in casi speciali in cui vengono chiamati dall'interprete gli attributi speciali.

Es. nei cicli `__iter__` e `__next__`; se usiamo `len()` che chiama `__len__`

Classi - Accesso agli attributi

Es. se usiamo `len()` che chiama `__len__`

`len()` restituisce la lunghezza di un elemento (es. elementi in una lista).

sotto l'interprete chiama la funzione speciale `__len__`

Classi - Accesso agli attributi

```
class MyList(list):
```

```
    def __getattr__(self, item):
```

```
        print(f'getattr {item}')
```

```
        return super().__getattr__(item)
```

```
    def foo(self):
```

```
        print('mi hai chiamato')
```

```
        pass
```

```
lista = MyList()
```

```
lista.foo -> getattr foo
```

```
lista.foo() -> getattr foo  
                mi hai chiamato
```

```
lista.__len__ -> getattr __len__
```

```
len(lista) -> //nessun output
```

- 1) l'interprete chiama direttamente
 __len__ senza passare da
 __getattr__
- 2) anche i metodi sono considerati come
 attributi -> coerente con il concetto
 che le funzioni sono anch'esse oggetti

Classi - Accesso agli attributi

Vi è l'analogo `__setattr__` per settare un attributo.

Di fatto modifica (o aggiunge) la corrispettiva entry in `__dict__`.

Non vi è il corrispettivo di `__getattr__` (non vi è un `__setattr__`).

`__setattr__` viene invocato ogni volta che si fa un assegnamento
(anche nell'`__init__`)

`self.x = 10 -> self.__setattr__('x', 10)`

Classi - Accesso agli attributi

Vi è l'analogo `__setattr__` per settare un attributo.

Di fatto modifica (o aggiunge) la corrispettiva entry in `__dict__`.

Ecco perchè se si fa

`prova.i = 10` non si modifica l'attributo di classe

```
class Prova:
```

```
    i = 0
```

```
prova = Prova()
```

Classi - Accesso agli attributi

```
class Prova:
```

```
    def __init__(self):
```

```
        self.saldo = 0
```

```
    def __setattr__(self, campo, valore):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non modificabile direttamente')
```

```
        else:
```

```
            return super().__bsetattr__(campo, valore)
```

Classi - Accesso agli attributi


```
class Prova:
```

```
    def __init__(self):
```

```
        self.saldo = 0
```

Problema!

Anche nell'__init__ verrà invocato il
__setattr__ impedendone
l'inizializzazione



```
    def __setattr__(self, campo, valore):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non modificabile direttamente')
```

```
        else:
```

```
            return super().__setattr__(campo, valore)
```


Classi - Accesso agli attributi

```
class Prova:
```

```
    def __init__(self):
```

```
        self.__dict__['saldo'] = 0
```

Accedendo tramite `__dict__` si bypassa la
`__setattr__`



```
    def __setattr__(self, campo, valore):
```

```
        if campo == 'saldo':
```

```
            raise AttributeError('campo non modificabile direttamente')
```

```
        else:
```

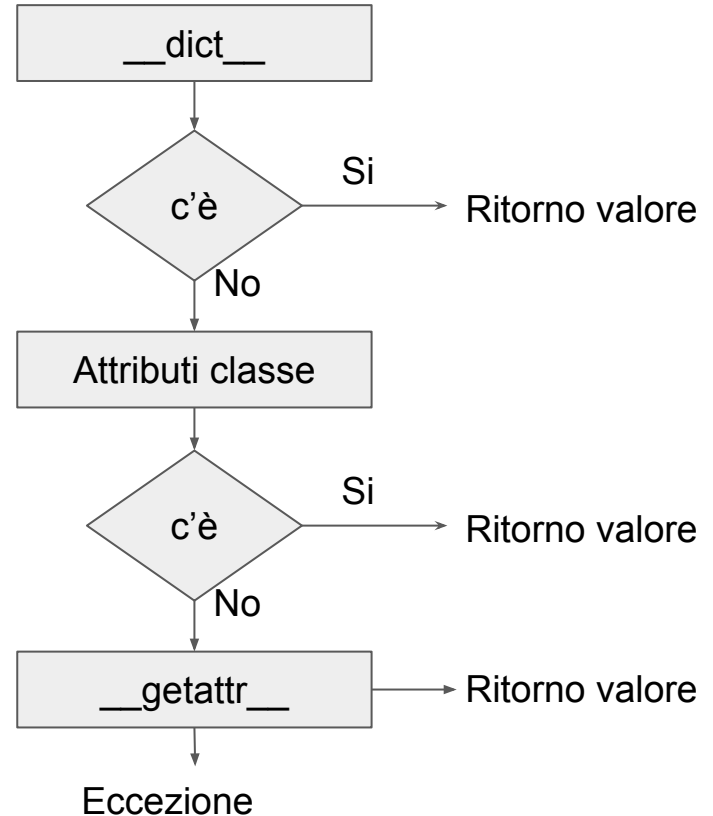
```
            return super().__basettribute__(campo, valore)
```

Classi - Accesso agli attributi

E @property?

E' un attributo di istanza o di classe o un metodo?

In che punto del flusso viene controllato?



Classi - Accesso agli attributi

Remind: @property è un decoratore e come tale si comporta

- aggiunge funzionalità ad una funzione
- più nel dettaglio il decoratore stesso è una funzione che prende in ingresso una funzione e ne ritorna un'altra
 - la nuova funzione viene associata al nome decorato

@dec

def fun():

pass

a **fun** sarà associata la funzione ritornata da
dec

Classi - Accesso agli attributi

Le funzioni sono entità come le altre (sono oggetti) in python.

Quindi il decoratore può prendere in ingresso qualsiasi entità e restituire qualsiasi entità.

Classi - Accesso agli attributi

```
class Prova:
```

```
    @property
```

```
    def foo(self):
```

```
        return 0
```

Sicuramente property prende in ingresso una funzione.

Cosa deve ritornare per avere l'effetto che conosciamo?

Classi - Accesso agli attributi

In python un attributo può essere anche una entità definita come **descrittore**.

In pratica un descrittore è un attributo il cui valore è un oggetto (e fin qui nulla di nuovo).

Tale oggetto implementa uno o più dei seguenti metodi:

`__get__`, `__set__`, `__delete__`

Classi - Accesso agli attributi

Tale oggetto implementa uno o più dei seguenti metodi:

`__get__`, `__set__`, `__delete__`

in questo caso l'interprete non setta/accede/cancella l'attributo direttamente ma chiama il corrispettivo metodo

Classi - Accesso agli attributi

```
class Desc:
```

```
    def __get__(self, istanza, owner):
```

```
        pass
```

```
class Prova:
```

```
    i = Desc()
```

Classi - Accesso agli attributi

```
class Desc:
```

```
    def __get__(self, istanza, owner):
```

```
        pass
```

```
class Prova:
```

```
    i = Desc()
```

Attributo di classe!

Classi - Accesso agli attributi

p = Prova()

p.i -> viene chiamato `__get__` di Desc

Classi - Accesso agli attributi

`p = Prova()`

`p.i` -> viene chiamato `__get__` di `Desc`

`class Desc:`

```
def __get__(self, istanza, owner):  
    pass
```

Istanza di `Desc`.

Nel nostro esempio `i` di `Prova`

Classi - Accesso agli attributi

```
p = Prova()
```

p.i -> viene chiamato `__get__` di Desc

```
class Desc:
```

```
    def __get__(self, istanza, owner):
```

```
        pass
```

Oggetto sul quale viene richiesto l'accesso all'attributo.

Nel nostro esempio p, istanza di Prova

Classi - Accesso agli attributi

`p = Prova()`

`p.i ->` viene chiamato `__get__` di `Desc`

`class Desc:`

```
def __get__(self, istanza, owner):
```

```
    pass
```

Classe dell'oggetto sul quale viene richiesto l'accesso all'attributo.

Nel nostro esempio `Prova`

Classi - Accesso agli attributi

```
p = Prova()
```

p.i -> viene chiamato `__get__` di Desc

```
class Desc:
```

```
    def __set__(self, istanza, value):
```

```
        pass
```

Oggetto Desc e oggetto Prova

Come in `__get__`

Classi - Accesso agli attributi

p = Prova()

p.i -> viene chiamato `__get__` di Desc

class Desc:

```
def __set__(self, istanza, value):
```

```
    pass
```

Valore da settare

Classi - Accesso agli attributi

p = Prova()

p.i -> viene chiamato `__get__` di Desc

class Desc:

```
def __delete__(self, istanza):  
    pass
```

Oggetto Desc e oggetto Prova

Come in `__get__`

Classi - Accesso agli attributi

Mettendo insieme i pezzi:

- Se `@property` ritorna un oggetto descrittore, tale oggetto potrà avere una funzione nel `__get__` (la funzione decorata)
- Tale descrittore verrà associato al nome della funzione originale
- Esso diventerà un attributo di classe in quando non è inizializzato usando `self`

Classi - Accesso agli attributi

- Esso diventerà un attributo di classe in quando non è inizializzato usando self

Occhio che a questo punto l'oggetto sarà il medesimo per tutte le istanze della classe Prova

(a @property non interessa in quanto usa una funzione dell'utente)

```
class Desc:
```

```
    def __get__(self, istanza, owner):
```

```
        pass
```

```
class Prova:
```

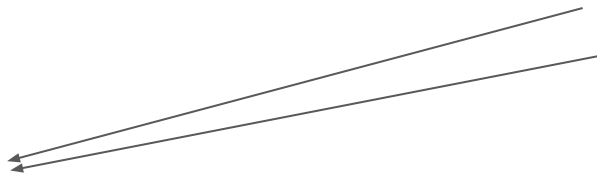
```
    i = Desc()
```

```
p1 = Prova()
```

```
p2 = Prova()
```

```
p1.i
```

```
p2.i
```



Classi - Accesso agli attributi

- Esso diventerà un attributo di classe in quando non è inizializzato usando self

Occhio che a questo punto l'oggetto sarà il medesimo per tutte le istanze della classe Prova

```
p1 = Prova()  
p2 = Prova()
```

```
p1.i = 10  
p2.i -> 10
```

```
class Desc:  
  
    def __init__(self):  
        self.x = 0  
  
    def __set__(self, instance, value):  
        self.x = value  
  
    def __get__(self, istanza, owner):  
        return self.x
```

```
class Prova:  
  
    i = Desc()
```

Classi - Accesso agli attributi

Se si vuole che ogni oggetto abbia la sua istanza dei valori di `i` bisogna usare **instance**.

```
p1 = Prova()  
p2 = Prova()
```

```
p1.i = 10  
p2.i -> Error p2 non ha 'i'
```

```
class Desc:  
    def __init__(self):  
        self.x = 0  
  
    def __set__(self, instance, value):  
        instance.__dict__['x'] = value  
  
    def __get__(self, instance, owner):  
        return instance.__dict__['x']
```

```
class Prova:  
    i = Desc()
```


Classi - Accesso agli attributi

In questo caso si utilizza `__dict__` per evitare ricorsioni ma questo implica....

```
class Desc:
```

```
    def __init__(self):
```

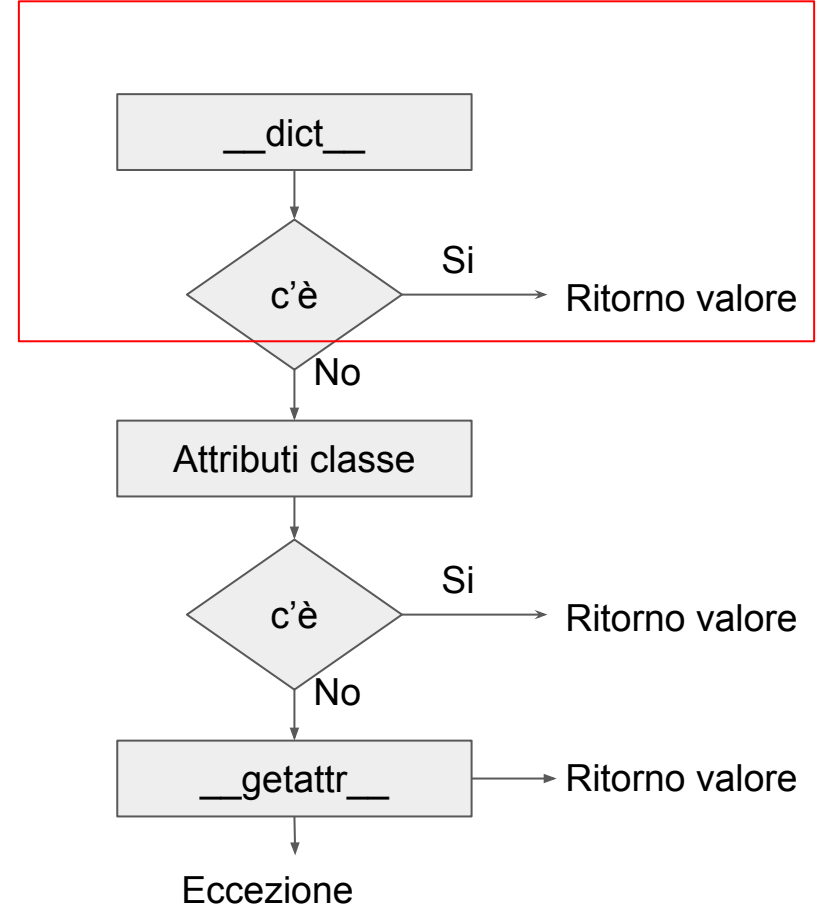
```
        self.x = 0
```

```
    def __set__(self, instance, value):
```

```
        instance.__dict__['x'] = value
```

```
    def __get__(self, instance, owner):
```

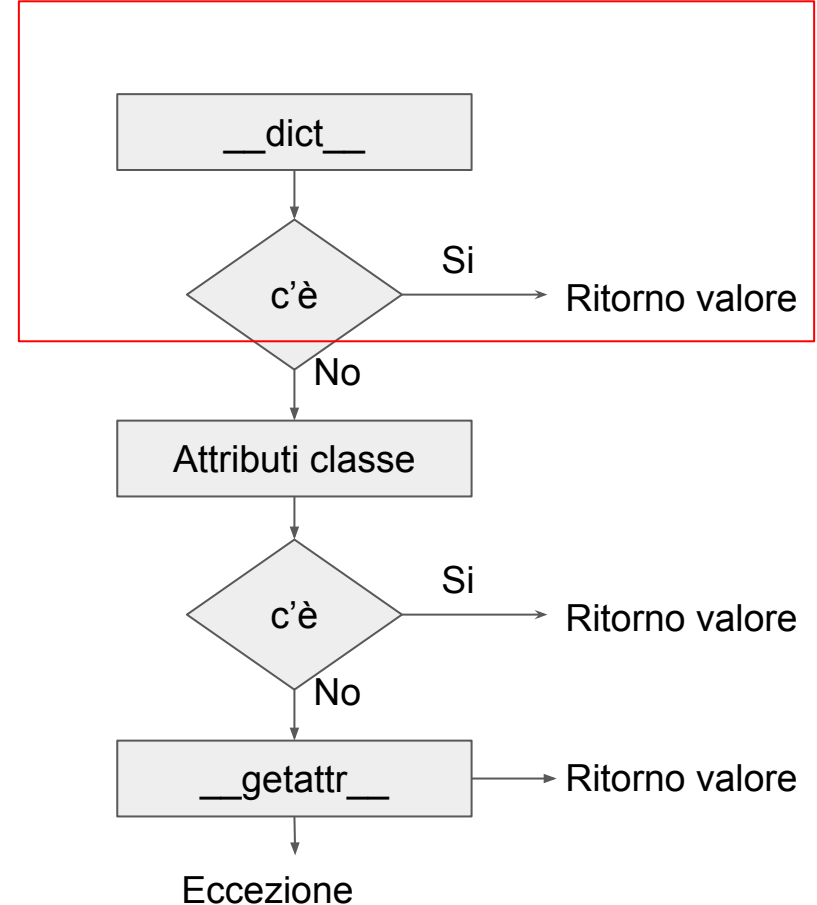
```
        return instance.__dict__['x']
```



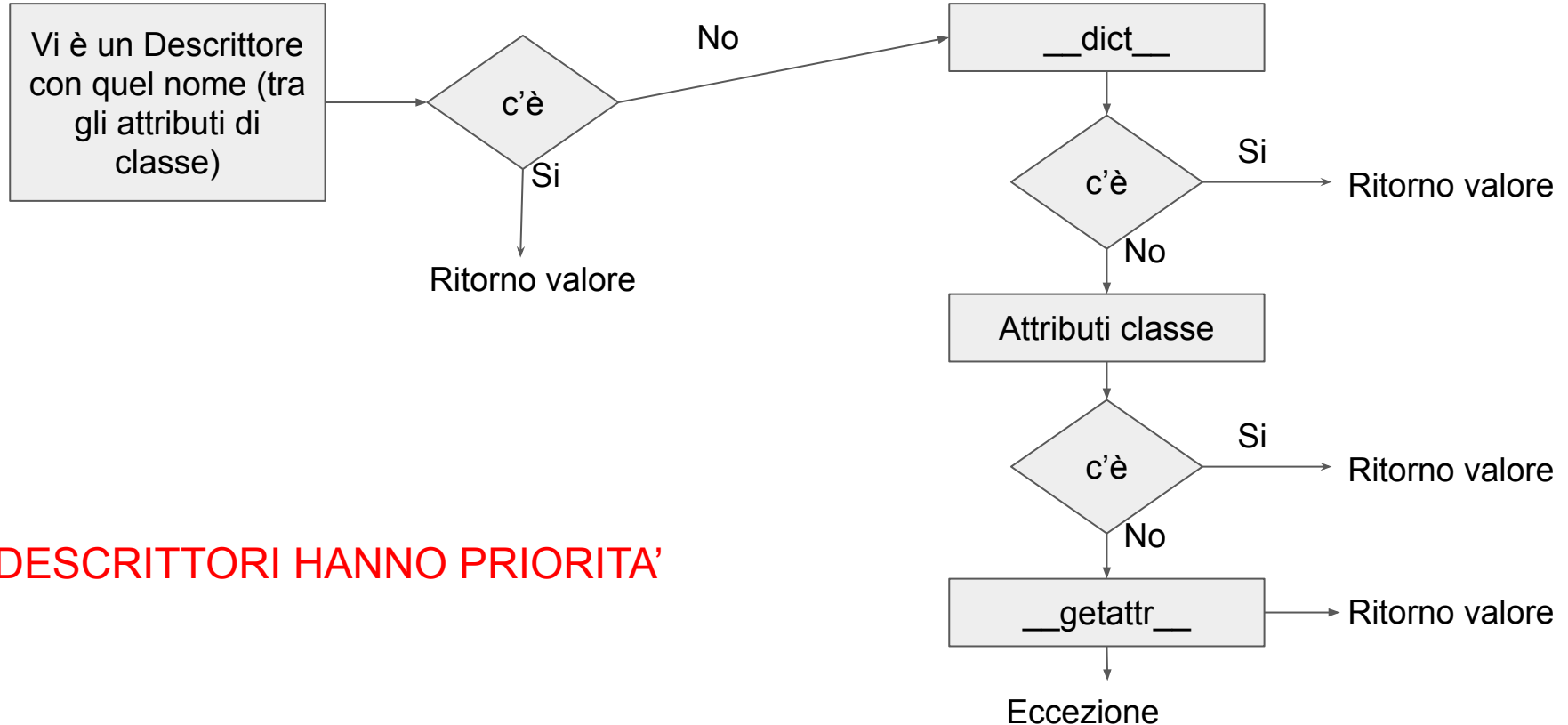
Classi - Accesso agli attributi

In questo caso si utilizza `__dict__` per evitare ricorsioni ma questo implica....

Che al secondo accesso non andrei a verificare gli attributi di classe e non vedrei il descrittore.



Classi - Accesso agli attributi



I DESCRITTORI HANNO PRIORITA'

Classi - Metaclassi

In programmazione a oggetti, una metaclasse è una classe le cui istanze sono a loro volta classi.

In python tutto è un oggetto. Quindi anche le classi sono oggetti. Quindi sono istanze di un'altra classe.

In python la metaclasse madre è **type**

Tutti le classi sono istanze di **type**

Classi - Metaclassi

Internamente Python crea le classi (le definizioni delle classi) istanziando la classe: "type", che, e' una "metaclassa", cioe' una classe le cui istanze sono delle classi.

È possibile estendere la classe "type" e creare una propria metaclassa, ove si ridefiniscono le funzioni `__init__` e la `__new__` in modo da modificare il comportamento di base delle classi.

```
class MiaClasse(metaclass=MiaMetaclassa):  
  
    pass
```

Classi - Metaclassi

type() ha due comportamenti

- con un argomento ritorna il tipo dell'oggetto
- con tre argomenti ritorna una nuova classe. E' l'equivalente di scrivere una nuova classe

```
class X(Y):
```

```
    a = 1
```

```
type('X', (Y), dict(a=1))
```

Classi - Metaclassi

Esempio Metaclassa - eredita da type

```
class MiaMetaclassa(type):  
    def __new__(cls, classname, super, classdict):  
        return super().__new__(cls, classname, super, classdict)  
    def __init__(self):  
        super().__init__()
```

Classi - Metaclassi

Le metaclassi possono essere utili per avere meccanismi simili all'ereditarietà ma anche per creare classi in modo diverso a runtime.

Es. si potrebbero creare classi con attributi di classe che hanno nomi definiti dall'utente.

Classi - metodo `__call__`

`__call__` è un metodo speciale che viene chiamato quando un oggetto viene invocato come una funzione

```
class Prova:
```

```
    def __call__(self):  
        print('ecco mi')
```

```
a = Prova()
```

```
a()
```

Classi - metodo `__call__`

Questo implica che anche una classe può essere un decoratore

class dec:

```
def __init__(self, f):
```

```
    self.f = f
```

```
def __call__(self):
```

```
    pass
```

```
@dec
```

```
def foo():
```

```
    print('ciao')
```

Classi - metodo `__call__`

Questo implica che anche una classe può essere un decoratore

class dec:

```
def __init__(self, f):
```

```
    self.f = f
```

```
def __call__(self):
```

```
    pass
```

```
@dec
```

```
def foo():
```

```
    print('ciao')
```

foo viene passato ad `__init__`

Classi - metodo `__call__`

Questo implica che anche una classe può essere un decoratore

class dec:

```
def __init__(self, f):
```

```
    self.f = f
```

```
def __call__(self):
```

```
    pass
```

```
@dec
```

```
def foo():
```

```
    print('ciao')
```

foo viene passato ad `__init__`

il nuovo oggetto ritornato verrà associato a foo

Classi - metodo `__call__`

Questo implica che anche una classe può essere un decoratore

class dec:

```
def __init__(self, f):
```

```
    self.f = f
```

```
def __call__(self):
```

```
    pass
```

```
@dec
```

```
def foo():
```

```
    print('ciao')
```

```
foo()
```

foo viene passato ad `__init__`

il nuovo oggetto ritornato verrà associato a foo

quando verrà invocato foo verrà chiamata la `__call__`

Classi - classi astratte

Una classe astratta è una classe che definisce almeno un metodo astratto.

Un metodo astratto è un metodo che non ha implementazione.

Esso va definito nelle sottoclassi.

Classi - classi astratte

Python non ha built-in la possibilità di definire metodi astratti.

Vi è però un modulo che permette ciò (Abstract Base (ABC))

Al suo interno vi è il decoratore `@abstractmethod` che permette di decorare i metodi che si vogliono come astratti.

```
from abc import ABC, abstractmethod
```

Classi - classi astratte

```
from abc import ABC, abstractmethod
```

```
class Astratta(ABC):
```

```
    @abstractmethod
```

```
    def metodo_astratto(self):
```

```
        pass
```


Classi - classi astratte

```
from abc import ABC, abstractmethod
```

```
class Astratta(ABC):
```

```
    @abstractmethod
```

```
    def metodo_astratto(self):
```

```
        pass
```

Deve ereditare da ABC

E solo da altre classi astratte!