Corso di **Python**

Lezione 7

Le classi

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Python è Object-Oriented

- Python è un linguaggio nativamente orientato agli oggetti
- È *multi-paradigma*, ossia permette l'uso di più paradigmi di programmazione:
  - Object-Oriented Programming
  - Programmazione procedurale (*imperativa*)
  - Programmazione funzionale (*dichiarativa*)
- I vari paradigmi possono essere utilizzati insieme senza particolari vincoli sintattici

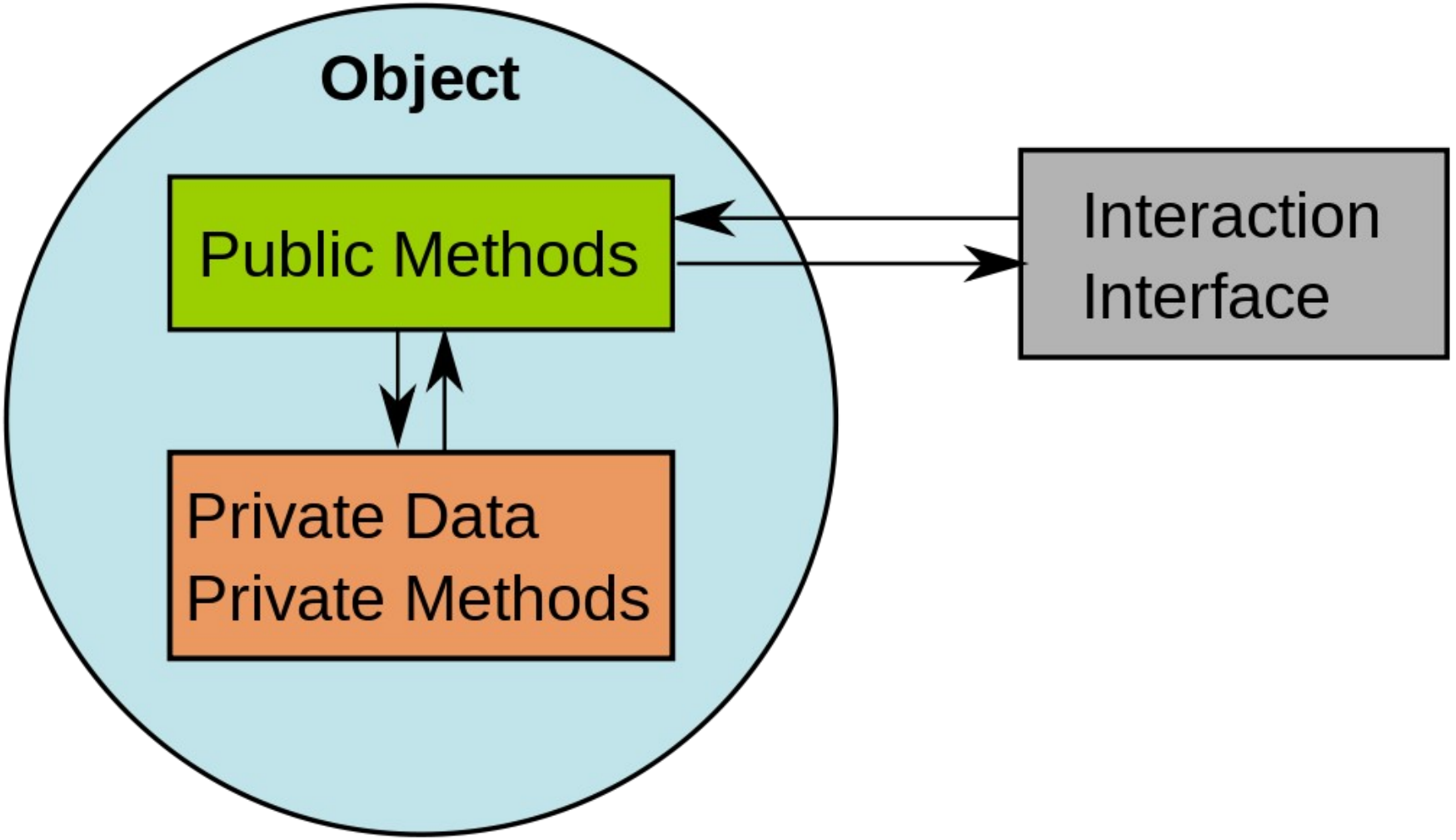
Cos'è la OOP

- La programmazione orientata agli oggetti (**OOP**) è un paradigma di programmazione focalizzato sul raggruppare insieme **stato** (dati) e **comportamento** (codice) in strutture chiamate **oggetti**
- Negli oggetti:
  - I dati sono rappresentati con **attributi** (spesso si usa definirli *proprietà*, in inglese si usa anche il termine *fields*)
  - Il codice è definito in funzioni chiamate **metodi** (*methods*)
- Per la dichiarazione degli oggetti si utilizza generalmente il concetto di **classe**
  - Una classe definisce una specifica tipologia di oggetti
  - Ogni oggetto è un'istanza di una specifica classe

Caratteristiche della OOP

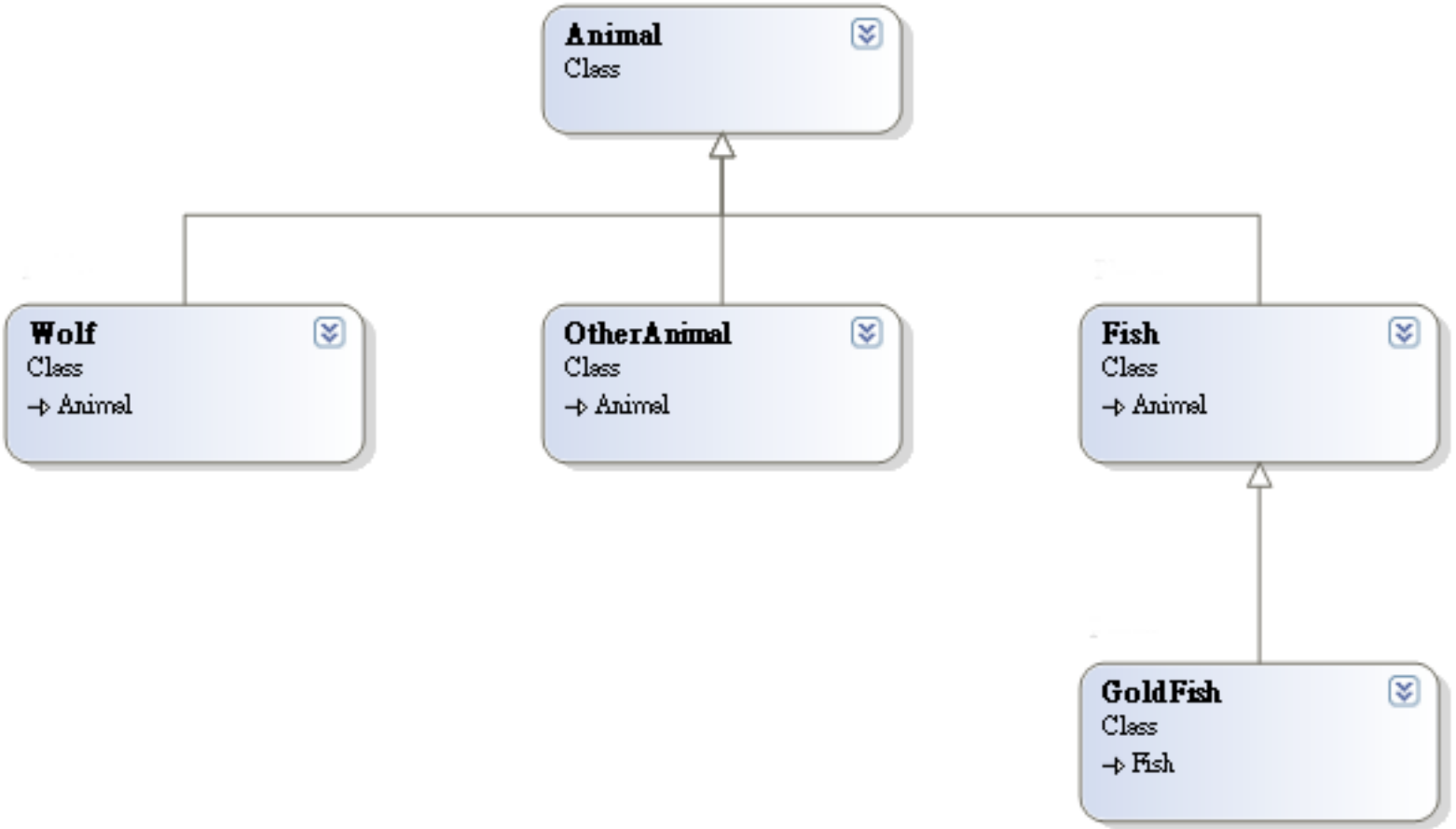
- ***Astrazione*** *(concetto non esclusivo della OOP ...)*
  - *Cogliere gli aspetti essenziali di un concetto, definendo una struttura per gestirlo*
- **Incapsulamento**
  - Ogni oggetto definisce uno stato interno mediante attributi e metodi di manipolazione ad uso prettamente interno, ossia limitato ad altri metodi dell'oggetto stesso
  - Separazione tra attributi/metodi interni ed esterni che facilita l'aggiornamento del codice nel suo complesso
- **Ereditarietà**
  - Si possono definire gerarchie di classi derivando classi da altre classi
  - Le classi derivate possono ereditare stato e comportamento
- **Polimorfismo**
  - La decisione di quale metodo applicare dipende dalla specifica istanza dell'oggetto (*binding dinamico* o *late binding*)
  - Possibilità di ridefinire o riutilizzare metodi anche in classi derivate

Incapsulamento



Ref: https://commons.wikimedia.org/wiki/File:CPT-OOP-interfaces.svg

Ereditarietà e polimorfismo



Ref: https://commons.wikimedia.org/wiki/File:ClassDiagram_for_VirtualFunction.png

Classe in Python

- Dichiarazione di una classe in Python:

```
class classname[(base-classes)]:
    statement(s)
```

  - *classname* : è il nome della classe
  - *base-classes* : sono le classi base su cui la classe viene definita

- Esempio:

```
>>> class A(object):   # object è la classe base comune di Python
...     x = 0
...
>>> A.x
0
>>> A.x = 19
>>> A.x
19
```

Classi *old/new style*

- La classe predefinita `object` è stata introdotta in Python 2:
  - Classi *new-style* se derivate da `object`
  - Senza classi base si parla invece di classi *old-style*
- In Python 2 la classe è per default *old-style* se non si specificano classi base:

```
>>> class A:
...     x = 0
...
>>> dir(A)
['__doc__', '__module__', 'x']
```

- Mentre in Python 3 tutte le classi sono *new-style*:

```
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'x']
```

Attributi di una classe

- Esempio classe con attributo:

```
class A(object):
    x = 0   # x è un attributo della classe

>>> A.x
0
```

- Nel corpo della classe l'attributo è locale:

```
>>> class A(object):
...     x = 0
...     y = 3 + x
...
>>> A.y
3
```

Istanze di una classe

- Per definire un'istanza della classe basta chiamarla come fosse una funzione:

```
>>> a = A()          # In questo caso senza parametri ...
>>> a.x
0
>>> a.x = 20
>>> a.x              # Modifico l'attributo di un'instanza ...
20
>>> b = A()
>>> b.x              # ... l'attributo di nuove istanze non risulta modificato
0
>>> A.x = -1         # Se modifico l'attributo a livello di classe ...
>>> c = A()
>>> c.x
-1
>>> a.x, b.x, c.x   # ... la modifica si riflette sulle nuove istanze.
(20, 0, -1)
```

- Classe e singole istanze della classe definiscono namespace distinti

Sottoclassi ed ereditarietà

- Si possono derivare altre classi da una classe esistente:

```
class A(object):
    x = 0

class B(A):
     pass
```

- La classe A è detta *classe base* (o *superclasse*) di B
- La classe B è detta *classe derivata* (o *sottoclasse*) di A
- Le classi ereditano attributi e metodi:

```
>>> B.x
0
```

- Classe e sottoclasse rappresentano dinamicamente namespace distinti:

```
>>> B.x = 10
>>> A.x
0
```

- Per testare la relazione di sottoclasse si usa il metodo built-in **issubclass:**

```
>>> issubclass(B, A)
True
```

Classe di un'istanza

- Per testare se un'istanza appartiene ad una determinata classe si usa la funzione built-in *isinstance*:

  isinstance(*object***,** *classinfo***)**

- Esempio:

```
>>> class A(object):
...     x = 0
...
>>> class B(A):
...     y = 0
...
>>> a = A()
>>> b = B()
>>> isinstance(a, A)
True
>>> isinstance(a, B)       # a non è istanza di classe B
False
>>> isinstance(b, B)       # b è invece sia istanza di B che della sua classe base
A
True
>>> isinstance(b, A)
True
```

Metodi di una classe

- Per definire un metodo si usa l'istruzione *def* come per le normali funzioni:

```
class A(object):
    x = 0
    def get_value(self):
        print('My value is:', self.x)


>>> a = A()
>>> a.get_value()
('My value is:', 0)
```

- In generale ogni metodo di una classe ha almeno un parametro, riferito all'istanza:
  - Al primo parametro possono seguire altri parametri come nelle normali funzioni
  - Unica eccezione per i *metodi statici*
- Per convenzione il parametro che indica l'istanza è sempre ***self***

Docstring per le classi

- Come specificare le docstring in una classe di Python:

```
class A(object):
    "Docstring per la classe"

    x = 0
    "Docstring per l'attributo A.x"

    def f(self):
        "Docstring per il metodo A.f"
        ...
```

Override di metodi/attributi

- Fondamentale l'override di metodi e attributi, connesso al concetto di polimorfismo:

```
class A(object):
    x = 0
    def get_value(self):
        print('My value is:', self.x)


class B(A):
    y = 0
    def get_value(self):
        print('My value is:', self.x, self.y)
```

- Il metodo eseguito dipende dalla classe dell'istanza:

```
>>> a = A()
>>> a.get_value()
('My value is:', 0)
>>> b = B()
>>> b.get_value()
('My value is:', 0, 0)
```

La funzione built-in `super`

- Spesso per l'override di un metodo è necessario richiamare quello della classe base
- In tutte le tipologie di classi si può usare la forma esplicita:

```
class B(A):
    ...
    def get_baseclass_value(self):
        A.get_value(self)
```

  - Problema: è un metodo statico, che non può essere modificato dinamicamente (ereditarietà multipla ...)

- Nelle classi new-style meglio usare la funzione built-in *super*:

```
super([type[, object-or-type]])
```

Esempio funzione **super**

- Esempio (definizione modello per Django Web Framework):

```
from django.db import models

class Persona(models.Model):
    id = models.AutoField(primary_key=True)
    nome = models.CharField(max_length=100, blank=False)
    cognome = models.CharField(max_length=100, blank=False)
    displayFirstName = models.CharField(max_length=100, blank=True)
    displayLastName = models.CharField(max_length=100, blank=True)

    def save(self, *args, **kwargs):
        if self.displayFirstName.strip() == '':
            self.displayFirstName = self.nome
        if self.displayLastName.strip() == '':
            self.displayLastName = self.cognome
        super(Persona, self).save(*args, **kwargs)
```

Metodi statici e di classe

- In C++ e Java si possono definire metodi/attributi non legati alla singola istanza ma solo alla classe (*metodi statici*)
- In Python i metodi statici e di classe si definiscono mediante appositi *decoratori*:

```
class A(object):
    @staticmethod
    def f1(arg1, arg2, ...):
        ...

    @classmethod
    def f2(cls, arg1, arg2, ...):
        ...
```

- La differenza tra le due definizioni è che nella seconda viene inserito un parametro che rappresenta la classe

Classi annidate

- All'interno di una classe si possono definire altre classi come fossero attributi strutturati
- Esempio:

```
class A(object):
    def __init__(self):
        self.x = 0

    class B(object):
        def __init__(self):
            self.y = 0

        def f(self):
            pass

>>> b = A.B()
>>> type(b)
<class '__main__.B'>
>>> b.f()
```

Attributi e metodi speciali

- Attributi read-only e metodi per le operazioni caratteristiche degli oggetti di Python
  - Il nome di questi attributi/metodi inizia e finisce con il *doppio underscore*
  - Esempi: `__class__, __init__()`
- Si differenziano in attributi/metodi speciali di *istanza o di classe*
- Attributi e metodi speciali sono definiti anche negli oggetti di base:

```
>>> dir(1000)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
'__coerce__', '__delattr__', '__div__', '__divmod__', '__doc__',
'__float__', '__floordiv__', '__format__', '__getattribute__',
'__getnewargs__', '__hash__', '__hex__', '__index__', '__init__',
'__int__', '__invert__', '__long__', '__lshift__', '__mod__',
'__mul__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdiv__',
'__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__',
'__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__',
'__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__',
'__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

Attributi speciali principali

Nome	Descrizione	Di classe	Di istanza
`__dict__`	Mappa/dizionario per gli attributi scrivibili	Sì	Sì
`__class__`	Classe a cui l'istanza appartiene	No	Sì
`__bases__`	Lista delle classi base	Sì	No
`__name__`	Nome della classe	Sì	No
`__mro__`	Lista gerarchia classi per i lookup	Sì	No

- Esempi:

```
>>> i = 1000
>>> i.__class__
<type 'int'>
>>> int.__name__
'int'
>>> i.__class__.__name__
'int'
>>> i.__class__.__bases__
(<type 'object'>,)
```

Metodi speciali

- Metodi legati a caratteristiche dell'oggetto stesso
- Possono rappresentare funzioni built-in o operatori
- In genere sono richiamati implicitamente
  - chiamate esplicite solo nel corpo della classe/sottoclasse
- I metodi speciali sono elencati nel modello dei dati di Python 2 e 3:
  - https://docs.python.org/2/reference/datamodel.html
  - https://docs.python.org/3/reference/datamodel.html

Metodi speciali principali

Nome	Descrizione
`__new__`	Crea una nuova istanza della classe
`__init__`	Inizializza l'instanza dopo la creazione
`__del__`	Chiamato quando l'istanza viene distrutta
`__repr__`	Rappresentazione stringa formale dell'oggetto
`__str__`	Rappresentazione stringa informale dell'oggetto
`__unicode__`	Rappresentazione come stringa unicode
`__hash__`	Ritorna un intero univoco per l'oggetto
`__cmp__`	Funzione di comparazione dell'oggetto rispetto ad altro oggetto
`__eq__, __ne__, __le__, __lt__, __ge__, __gt__`	Funzioni che definiscono il risultato degli operatori ==, !=, <=, <, >=, >
`__byte__`	Ritorna rappresentazione come bytes dell'oggetto (Python 3)
`__format__`	Ritorna la rappresentazione come stringa formattata (Python 3)

Tutti i metodi speciali ad esclusione di `__new__` sono legati alla specifica istanza

Il metodo `__init__`

- Inizializza l'istanza creata da `__new__`:

```
object.__init__(self[, ...])
```

- È il metodo speciale più usato negli override
- In genere ha più di un parametro, per definire istanze diverse
- Non deve ritornare nulla altrimenti viene generato un errore (*None* è però ammesso):

```
>>> class A(object):
...     def __init__(self):
...         print("INIT CALLED")
...         return "OK"
...
>>> a1 = A()
INIT CALLED
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() should return None, not 'str'
```

Un override di `__init__`

```
class BaseModel(object):
    attr1 = None

    def __init__(self):
        self.attr1 = 'one'
        self.attr2 = 'two'

class MyModel(BaseModel):
    def __init__(self):
        self.attr3 = 'three'
        super(MyModel, self).__init__()
```

Il metodo __new__

- Crea un nuovo oggetto della classe
- Esempio:

```
>>> class A(object):
...     def __new__(cls):
...         print("NEW CALLED")
...         return super(A, cls).__new__(cls)
...     def __init__(self):
...         print("INIT CALLED")
...
>>> a1 = A()              # Crea l'oggetto e poi lo inizializza
NEW CALLED
INIT CALLED
>>> a2 = A.__new__(A)     # In questo caso si crea solo l'oggetto
NEW CALLED
```

- Utile un override di `__new__` per alterare il valore di un oggetto immutabile (int, float, str)
  - Dettagli in "Unifying types and classes in Python 2.2" di Guido van Rossum

Il metodo `__del__`

- Richiamato quando l'istanza viene eliminata
  - Via garbage collection
  - Esplicitamente con il comando **del**

- Esempio:

```
>>> class A(object):
...     def __del__(self):
...         print("DEL CALLED")
...
>>> a = A()
>>> del a
DEL CALLED
```

Il metodo `__call__`

- Viene utilizzato quando si richiama l'istanza come fosse una funzione:

```
A(arg1, arg2, ...)   # Equivale a A.__call__(arg1, arg2, ...)
```

- Esempio:

```
>>> class C(object):
...     def __call__(self, *args, **kwargs):
...         print('Called:', args, kwargs)
...
>>> c = C()
>>> c(1, 2, 3)
('Called:', (1, 2, 3), {})
>>> c(1, 2, 3, x=4, y=5)
('Called:', (1, 2, 3), {'y': 5, 'x': 4})
```

Overloading di operatori

- Posso ridefinire gli operatori aritmetici e logici (`==`, `!=`, `<`, `...`)
- Esempio:

```
>>> class ComparableFloat(float):
...     def __eq__(self, other):
...         return abs(self - other) <= 0.01
...
>>> ComparableFloat(10) == ComparableFloat(10.1)
False
>>> ComparableFloat(10) == ComparableFloat(10.01)
True
>>> 10 == 10.000000000000001
False
```

Attributi e metodi privati

- Per definire se un oggetto è privato non si usano direttive esplicite ma convenzioni sintattiche:
  - Pubblica: **`name`**
  - Non-pubblica e soggetta a modifiche: **`_name`**
  - Privata alla classe: **`__name`** (o al limite anche `__name_`)
- Queste convenzioni permettono facilmente di sapere la tipologia di oggetto a cui si sta accedendo
- Sugli attributi/metodi privati viene applicato il cosiddetto *name mangling* (storpiatura del nome), ridefinendo dinamicamente il nome come `_classname__name`
- Il name mangling non viene applicato su attributi e metodi speciali (doppio underscore iniziale e finale)

Name mangling ... perché?

- Si usa quando serve che all'interno di una classe base si continui ad usare il metodo originario e non un suo override:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Funzioni built-in specifiche per classi e oggetti

- object
- hash
- repr
- ascii
- property
- classmethod
- staticmethod
- isinstance
- issubclass
- super

Funzione built-in **property**

- Ritorna un attributo per la classe:

```
class property(fget=None, fset=None, fdel=None, doc=None)
```

  - *fget*: funzione per leggere l'attributo
  - *fset*: funzione per scrivere l'attributo
  - *fdel*: funzione per cancellare l'attributo
  - *doc*: stringa di documentazione

- Esempio:

```
class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")
```

N.B.: *property* è una classe che è usata come funzione o come decoratore ...

Argomenti OOP avanzati

- Metaclassi
- Metaprogrammazione
- Ereditarietà multipla
- OOP design patterns

Metaclassi

- In Python una **metaclasse** è il tipo definito da una classe:

```
>>> class OldStyleClass: pass
...
>>> class NewStyleClass(object): pass
...
>>> print(type(OldStyleClass), type(NewStyleClass))
(<type 'classobj'>, <type 'type'>)
```

- È possibile definire proprie metaclassi:

```
class MyMeta(type):
    def __str__(cls): return "Beautiful class '%s'" % cls.__name__
class MyClass(object):
    __metaclass__ = MyMeta

>>> x = MyClass()
>>> print(type())
Beautiful class 'MyClass'
```

Metaprogrammazione

- La metaprogrammazione riguarda programmi in grado di utilizzare un codice come dato per generare nuovo codice
- In Python si usa molto il concetto OOP di *factory*, implementato in genere mediante l'uso della funzione built-in *type*
- Esempio (tratto dall'implementazione di Django):

```
def formset_factory(form, formset=BaseFormSet, extra=1, can_order=False,
                    can_delete=False, max_num=None, validate_max=False,
                    min_num=None, validate_min=False):
    if min_num is None: min_num = DEFAULT_MIN_NUM
    if max_num is None: max_num = DEFAULT_MAX_NUM
    absolute_max = max_num + DEFAULT_MAX_NUM
    attrs = {'form': form, 'extra': extra,
             'can_order': can_order, 'can_delete': can_delete,
             'min_num': min_num, 'max_num': max_num,
             'absolute_max': absolute_max, 'validate_min': validate_min,
             'validate_max': validate_max}
    return type(form.__name__ + str('FormSet'), (formset,), attrs)
```

Ereditarietà multipla

- Quando una classe ha più classi base
- MRO – Method Resolution Order

```
class A:
  def save(self):
    print('A.save')

class B(A):
    pass

class C:
  def save(self): print('C.save')

class D(B, C): pass    # L'ordine di lookup è D, B, A, C
```

- **Mixins**: classi usate per inserire proprietà e metodi extra in una classe

```
class MixinClass(object):
    def new_method(self): pass

class MyClass(MixinClass, BaseClass):
    pass
```

Design Patterns

- Per sfruttare al meglio la potenza della OOP creando corrette gerarchie di classi
- Alcuni pattern molto usati:
  - Singleton
  - Decorator
  - Factory
  - Iterator
- Per una panoramica su tutti i pattern:
  - http://www.oodesign.com/
- Per capirli serve masticare un po' di UML ...