Programmazione ad oggetti in Python

Object Oriented Programming (OOP)

- Unità fondamentale: classe come contenitore di variabili (attributi) e funzioni (metodi)
- OOP: strumenti a disposizione
  - Astrazione: capacità di ridurre il numero di dettagli implementativi
  - Incapsulamento: capacità di proteggere lo stato interno dai moduli esterni
  - Ereditarietà: capacità di ereditare il comportamento di una classe (riuso codice)
  - Polimorfismo: capacità di adattare il comportamento in funzione dei parametri

Definizione di classe in Python

- **Una classe tipicamente viene definita all'interno di un modulo tramite lo statement class**

class ClassName:
<statement 1>
…
<statement n>

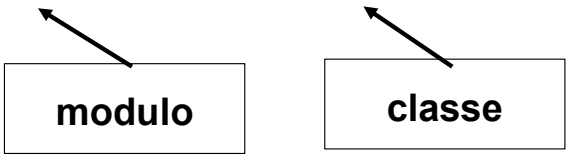
```
class MyClass:
    "Descrizione classe" #__doc__
    i = 12345
    def f(self):
        return 'hello world'
```

- **La definizione dei metodi nelle classi include un parametro speciale self**
- **La definizione di una classe crea un nuovo namespace (scope locale della classe)**
  - **Classi diverse possono avere variabili e funzioni con lo stesso nome**

Importazione di una classe

- ***Clausola import per accedere a classi definite in moduli Python***
  - **Es. classe MyClass definita nel modulo MyClass.py**

1) from MyClass import MyClass



uso: **inst = MyClass()**
var_doc = MyClass.__doc__

Oppure:

2) import MyClass
uso: **inst = MyClass.MyClass()**

→ESEMPI
MyClass.py

Oggetti classe

- Processata la definizione di una classe, l'interprete crea un oggetto (di tipo) classe
  - Involucro per contenuti del namespace
- Tale oggetto abilita due operazioni:
  - Instantiation: permette la creazione di istanze di classe tramite la *notazione funzionale* che invoca il metodo costruttore
    variabile = NomeClasse()
  - Attribute reference: permette l'accesso agli elementi di una classe (attributi di classe e metodi), tramite la sintassi:
    NomeClasse.nome_elemento

Uso dell'oggetto classe

```
>>> import MyClass
```

- Al momento dell'import, l'interprete processa la definizione della classe e genera l'oggetto classe

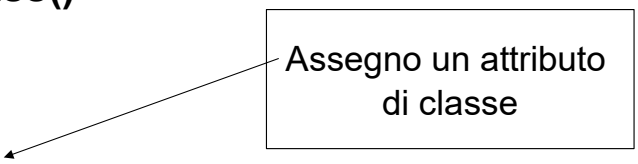
```
>>>  type(MyClass.MyClass)  → <type 'classobj'>
>>>  type(MyClass.MyClass.i)  → <type 'int'>
>>> MyClass.MyClass.i → 12345    #attributo di classe
>>> type(MyClass.MyClass.f)  → <type 'instancemethod'>
>>> MyClass.MyClass.f() → <unbound method MyClass.f>
```

Per essere invocato, un metodo deve essere riferito a una istanza di classe

```
>>> istance = MyClass.MyClass()  #creazione istanza
>>> istance.f()
Hello world
```

Oggetto classe: accesso agli attributi di classe

```
>>> from MyClass import MyClass
>>> istance = MyClass()
>>> print(instance.i)
12345
>>> MyClass.i = 100
>>> istance2 = MyClass()
>>> print(istance.i, istance2.i)
100 100
```



I cambiamenti agli attributi di classe si ripercuotono su tutte le istanze (equivalenti a var static in Java)

Il metodo costruttore

→ESEMPI
Complex.py
Bag.py

- **Il metodo costruttore di una classe ha nome __init__, e viene invocato automaticamente in seguito all'istanziazione di una classe**
- **Il metodo __init__, come una normale funzione, può avere argomenti in ingresso**
  - ***Argomenti obbligatori e di default***
- **C'è sempre un argomento implicito (self) che rappresenta un riferimento all'istanza stessa**
- **Lo stato interno di una classe è inizializzato attraverso gli statement inclusi in __init__**
  - **Attributi e metodi sono pubblici (default)**
  - **Attributi self.X rappresentano attributi di istanza**

Modifica a run time

- Come altre strutture dati, anche le classi Python possono essere 'modificate' a run time
- Possibile creare dinamicamente attributi all'interno di una istanza tramite semplici operazioni di assegnazione
- Es., creo un attributo counter sull'istanza x:

  x.counter = 1

- counter viene trattato come una variabile temporanea, valida solo per la specifica istanza x
  - Quando x viene distrutto, counter sparisce
  - Counter può anche essere distrutto con del

Riassumendo: attributi di classe e di istanza

```
class MyClass2:
    c = 100
    i = 12345
    def __init__(self,a,b):
        self.i = a
        self.x = b
    def print_i(self):
        print(self.i)
        print(MyClass2.i)
    def print_c(self):
        print(self.c)
        print(MyClass2.c)
```

Attributi di classe - pubblici

Attributi di istanza - pubblici

Attributo di istanza

Non obbligatorio ma buona norma: inizializzare gli attributi di istanza nel metodo __init__() !!

→ESEMPI
MyClass2.py

→ESEMPI
Private.py

Attributi e metodi privati

In Python viene considerato privato ogni identificatore di attributo o metodo preceduto da (almeno) due underscore e seguito da (al più) un underscore

```
def __init__(self, n):
    # inizializza la variabile privata __priv1_
    self.__priv1_ = n
```

Accessibile solo attraverso metodi

```
def get_priv1(self):
    return self.__priv1_
```

Se si tenta accesso diretto attraverso istanza.__priv1_

AttributeError: instance has no attribute '__priv1_'

Argomento self

- Nella definizione dei metodi di classe è sempre presente un primo argomento self, che rappresenta il riferimento all'istanza stessa
- L'invocazione di un metodo di classe richiede la presenza di un'istanza della classe
- L'argomento self, ossia il riferimento all'istanza, viene passato:
  - Implicitamente dall'interprete, se il metodo è stato invocato tramite un'istanza
  - Esplicitamente dal programmatore, se il metodo è stato invocato tramite l'oggetto classe

Invocazione Metodi

Definizione

```
class MyClass:
def f(self):
    return 'hello world'
```

```
>>> from MyClass import MyClass
>>> x = MyClass()
#La chiamata:
>>>  x.f()
#corrisponde a:
>>> MyClass.f(x)
```

MyClass: oggetto classe
x : istanza della classe

Stampa 'hello world'

Metodi static e class

- Metodi speciali della classe che possono essere invocati senza bisogno di istanziare un oggetto
- Definizione di metodi static:

```
class MyClass:
        @staticmethod
        def the_static_method(x):
                print x
>>> MyClass.the_static_method(2)    #output: 2
```

- I metodi static non ricevono in input il primo argomento implicito self

→ESEMPI
class_static.py

Metodi static e class

- **Definizione di metodi class:**

```
class MyClass:
        @classmethod
         def the_class_method(self, x):
                print x
>>> MyClass.the_class_method(2)    #output: 2
```

ATT: self è un riferimento all'oggetto classe, non all'istanza

- **Entrambi i metodi lavorano solo su attributi di classe e non di istanza**
- **Entrambi invocabili sia su classe che su istanze**
- **Le differenze sono legate al loro comportamento nel caso di presenza di sottoclassi *(vedremo)***

Ereditarietà

- **Sintassi per l'ereditarietà:**

```
class DerivedClassName(BaseClassName):
   <statement 1>
   …
   <statement n>
```

Se assente, sottinteso object
class MyClass(object):

- **Riferimento alla superclasse:**
  - **nome completo (notazione puntata) della superclasse**
  - **importazione diretta del nome della superclasse dal relativo modulo**

→ESEMPI
Ereditarieta.py

Polimorfismo

- **Method overriding: è sufficiente ridefinire un metodo in una sottoclasse**
- **All'interno di un metodo ridefinito nella sottoclasse, possibile invocare il metodo della superclasse con la sintassi:**

BaseClassName.methodname(self, arguments)

Differenze tra metodi class e static

- **Differenza: comportamento nel caso di sottoclassi**
- **Metodi Class**
  - **In caso di metodi con riferimenti a self (oggetto classe), l'invocazione si riferirà alla (sotto)classe attraverso cui viene invocato il metodo**
- **Metodi static**
  - **Ogni riferimento deve essere fatto ad attributi di classe in maniera esplicita (non accetta self)**
  - **Ogni chiamata si riferirà ad attributi della Parent Class**

ESEMPI
class_static_ered.py

Ereditarietà: funzioni di controllo

Funzioni:

- **isinstance(object, class): ritorna True (False) se obj è (non è) un'istanza della classe class**
- **issubclass(class1, class2): ritorna True (False) se class1 è (non è) una sottoclasse di class2**

Capacità di metaprogramming di Python (introspezione)

>>> isinstance("37",str) → True

>>> isinstance(3.7,float) → True

>>> issubclass(bool,int) → True

>>> issubclass(bool,float) → False

Ereditarietà multipla

→ESEMPI
ereditarieta_mult.py
ereditarieta_mult_Base.py

- **Una classe può ereditare da più classi con la seguente sintassi:**

```
class DerivedClassName(C1, C2, C3):
    <statement 1>
    …
    <statement n>
```

In Java realizzata attraverso le interfacce

- **La ricerca di attributi o metodi non definiti in DerivedClassName avviene con visita in profondità (depth-first) nella gerarchia delle superclassi:**
  - **Prima in C1 (e superclassi), poi in C2 (e superclassi), infine in C3 (e superclassi)**

Riferimento alla classe padre

class B(A):

A.__init__()

- **Il riferimento alla classe padre può avvenire tramite uso diretto del nome della superclasse**
- **Alternativa: funzione super([type [, obj]])**

class B(A):

super().__init__() equivale a super(B,self).__init__()

- **Restituisce un riferimento alla classe padre**
  - **type : classe corrente**
  - **obj: istanza della classe corrente**
  - **isinstance(obj, type) deve essere True**