Pattern Recognition
and Applications Lab

La Programmazione ad Oggetti in Python

Docente: Ambra Demontis

Anno Accademico: 2020 - 2021

University of Cagliari,
Italy

Department of Electrical and
Electronic Engineering

Corso di Laurea in Ingegneria Elettrica, Elettronica e Informatica

La Programmazione ad Oggetti in Python

In queste slide vedremo:

- Polimorfismo
- Duck Typing
- Classi Astratte

Polimorfismo

In Python, la stessa classe non può avere più metodi con lo stesso nome.

```
class CStudente:
  def __init__(self, nome, cognome):
     self.nome = nome
     self.cognome = cognome

  def __int__(self):
     self.nome = input("inserisci il nome ")
     self.cognome = input("inserisci il cognome ")
```

Non si può fare!

Polimorfismo

Classi diverse però possono avere funzioni con lo stesso nome…

```
class CAzienda(CCliente):
  def stampa_telefono(self):
     print("Il numero di telefono dell'azienda con ", self.idx, " è: ",
         self.numero_telefono)

class CPersona(CCliente):
  def stampa_telefono(self):
     print("Il numero di telefono della persona con ", self.idx, " è: ",
         self.numero_telefono)
```

Polimorfismo

```
class CCliente:
  def __init__(self, idx, numero_telefono):
     self.idx = idx
     self.numero_telefono = numero_telefono

oggetto_azienda = CAzienda("123", "070998899")
oggetto_persona = CPersona("422", "338786544")
```

Polimorfismo

Il codice che utilizza gli oggetti appartenenti a queste classi, per stampare il numero di telefono non ha bisogno di sapere se l'oggetto appartiene alla classe azienda o alla classe persona.

Dovrà semplicemente richiamare la funzione stampa_telefono dell'oggetto e il comportamento di questa funzione sarà differente a seconda della classe di apparenenza dell'oggetto.

Polimorfismo

oggetto_azienda.stampa_telefono()

oggetto_persona.stampa_telefono()

Stamperà:

Il numero di telefono dell'azienda con 123 è: 070998899

Il numero di telefono della persona con 123 è: 070998899

Polimorfismo

Il fatto che lo stesso codice di comporti in modi differente a seconda della sottoclasse di appartenenza viene detto **polimorfismo**.

Duck Typing

In Python, per poter scrivere un unico codice che può trattare diverse classi e che funziona in modo differente a seconda della classe di appartenenza dell'oggetto, non è necessario che queste classi siano figlie della stessa classe.

Basta che queste classi abbiano la stessa interfaccia!

Questa proprietà viene detta **duck typing**.

"Se cammina come un'anatra e nuota come un'anatra, allora è un'anatra."

(Se la classe ha l'interfaccia che mi serve, allora posso utilizzarla).

Duck Typing

Supponete di voler creare un programma di contabilità per un piccolo negozio che vende generi alimentari. Il gestore, nel periodo degli sconti, applica uno sconto differente in base al prodotto ma sempre uguale per lo stesso prodotto.

Duck Typing

Realizzate quindi il seguente codice:

```
class CQuadernoniPigna():
  prezzo = 3
  @classmethod
  def calcola_prezzo_scontato(cls):# 10 % sconto
     return cls.prezzo * 10 / 100

class CCaffePaulistaOro():
  prezzo = 5
  @classmethod
  def calcola_prezzo_scontato(cls): # sottrai sempre 2 euro al prezzo
     return cls.prezzo - 2
```

Il codice che utilizza queste classi potrà sfruttare il duck typing: diverse classi (non parenti) che espongono un metodo con la stessa interfaccia.

Le Classi Astratte

C’è però un problema!

Supponete che il codice per la gestione di questo programma diventi molto grande e complesso (con tante classi, tanti metodi ecc.)

Supponete anche che venga assunto un altro programmatore che dovrà darvi una mano creando nuove classi.

Come fa a sapere quali interfacce deve rispettare?

Le Classi Astratte

Una soluzione potrebbe essere fornirgli i diagrammi delle classi di tutto il codice…

Nella pratica questo spesso non è fattibile.

I diagrammi di classe generalmente vengono utilizzati per definire i prototipi.

E’ raro che vengano mantenuti aggiornati quando un progetto molto grande viene modificato!

Le Classi Astratte

Esiste però un meccanismo ad-hoc: le **classi astratte**.

Le classi astratte sono classi che non possono essere istanziate.

Servono a definire l'interfaccia che altre classi dovranno avere.

Le Classi Astratte

```
from abc import ABC, abstractmethod

class CProdotto(ABC):

  @classmethod
  @abstractmethod
  def calcola_prezzo_scontato(cls):
    pass
```

Il nuovo programmatore, quando quarda questa classe astratta sa che la nuova classe che deve implementare dovrà avere un metodo chiamato *calcola_prezzo_scontato* che non riceve nessun argomento.

Le Classi Astratte

Supponiamo di volere creare una classe astratta per il codice:

```
class CQuadernoniPigna(CProdotto):
  prezzo = 3
  @classmethod
  def calcola_prezzo_scontato(cls):# 10 % sconto
     return cls.prezzo * 10 / 100

class CCaffePaulistaOro(CProdotto):
  prezzo = 5
  @classmethod
  def calcola_prezzo_scontato(cls): # sottrai sempre 2 euro al prezzo
    return cls.prezzo - 2
```

Le Classi Astratte

Se dovesse comunque creare una classe che non ha quel metodo, quando un oggetto di quella classe verrà istanziato verrà sollevata un'eccezione. Es:

```
class CCacaoCraiAmaro(CProdotto):
  pass

oggetto_cacao = CCacaoCraiAmaro()
```

```
TypeError: Can't instantiate abstract class CCacao with abstract methods
calcola_prezzo_scontato
```

Esercizio sull'uso delle Classi Astratte

Supponete di voler creare le classi CRagioniere e CProgrammatore per un programma di gestione degli stipendi dei dipendenti.

Entrambe le classi dovranno avere un metodo che permette di calcolare lo stipendio dei dipendenti considerando che:

- Lo stipendio dei programmatori attualmente è di 1500 euro e nel mese di dicembre gli viene assegnata una premialità di 500 euro
- Lo stipendio dei ragionieri attualmente è di 1300 euro e nel mese di marzo gli viene assegnata una premialità del 30% dello stipendio.

Esercizio sull'uso delle Classi Astratte

```
class CProgrammatore(CDipendente):
  stipendio_base = 1500

  @classmethod
  def calcola_stipendio(cls, mese):
    if mese == 12:
      return cls.stipendio_base + 500
    else:
      return cls.stipendio_base
```

Dove CDipendente è la classe astratta che definiremo nelle prossime slide.

Esercizio sull'uso delle Classi Astratte

```
class CAnalista(CDipendente):
  stipendio_base = 1300

  @classmethod
  def calcola_stipendio(cls, mese):
    if mese == 3:
      return cls.stipendio_base + cls.stipendio_base * 30 / 100.
    else:
      return cls.stipendio_base
```

Dove CDipendente è la classe astratta che definiremo nelle prossime slide.

Esercizio sull'uso delle Classi Astratte

```
from abc import ABC, abstractmethod

class CDipendente(ABC):

  @classmethod
  @abstractmethod
  def calcola_stipendio(cls):
    pass
```

Esercizio sull'uso delle Classi Astratte

```
print("Lo stipendio di un programmatore a marzo è ",
    CProgrammatore.calcola_stipendio(3))
print("Lo stipendio di un programmatore a dicembre è ",
    CProgrammatore.calcola_stipendio(12))

print("Lo stipendio di un analista a marzo è ",
    CAnalista.calcola_stipendio(3))
print("Lo stipendio di un analista a dicembre è ",
    CAnalista.calcola_stipendio(12))
```

Esercizio sull’uso delle Classi Astratte

Stamperà:

Lo stipendio di un programmatore a marzo è 1500

Lo stipendio di un programmatore a dicembre è 2000

Lo stipendio di un analista a marzo è 1690.0

Lo stipendio di un analista a dicembre è 1300