OOP: Object Oriented Programming

Riferimenti per questa parte:

1. Capitoli 7 e 8 del testo di riferimento
2. https://docs.python.org/3.6/tutorial/classes.html

OOP: Idea Principale

L'idea principale della programmazione orientate agli oggetti è di pensare agli **oggetti** come delle collezzioni sia di **dati** che di **metodi** che operano su quei dati.

Gli oggetti sono una caratteristica dominante di Python: ogni **oggetto** ha un **tipo** che definisce il tipo di **operazioni** che un programma può eseguire su quell'oggeto.

Abbiamo visto come definire nuove **funzioni**, ora vediamo come definire nuovi **tipi**

Esempio: Numeri Complessi

Tipo: Numero complesso

Dati: due numeri reali, la parte reale e quella immaginaria

Operazioni: somma, stampa a video, calcolo del coniugato…

La specifica delle operazioni possibili definisce una **interfaccia** tra i dati ed il resto del programma. Il modo (l'algoritmo usato e la sua implementazione) in cui viene calcolato il coniugato di un numero complesso viene nascosto al resto del programma (**ENCAPSULATION**)

Abstract Data Types

Un **Abstract Data Type (ADT)** è un insieme di oggetti e di operazioni sugli stessi oggetti

Operazioni ed oggetti sono raggruppati in modo che possano essere passati da una parte all'altra del programma

Due concetti chiave della programmazione sono:

1. **DECOMPOSITION**: viene usata per creare una struttura nel nostro programma
2. **ABSTRACTION**: si cerca di eliminare i dettagli insignificanti per concentrarsi sugli aspetti fondamentali di un problema

ESEMPIO: *pensare sempre ai numeri complessi, quando vengono passati ad una funzione*

Classi

Un Abstract Data Type (ADT) è un insieme di oggetti e di operazioni sugli stessi oggetti. In Python, si implementano nuovi ADT definendo nuoci classi

```
class NumeroComplesso:
    def __init__(self, real, imag):
        """Metodo costruttore, chiamato quando viene
           inizializzato un nuovo oggetto"""
        self.a = real
        self.b = imag

    def somma(self, c):
        """Somma al numero corrente
           il numero complesso c"""
        self.a = self.a + c.a
        self.b = self.b + c.b

    def __str__(self):
        """Ritorna una stringa che rappresenta il numero"""
        return str(self.a) + ' + ' + str(self.b) +'i'
```

Operazioni su Classi

Una classe supporta due tipi di operazioni:

1. **Istanziazione**: viene usata per creare una nuova **istanza** della classe, ovvero un nuovo oggetto del tipo definite dalla classe stessa. Quando viene creato un nuovo oggetto viene sempre richiamato il suo **metodo costruttore: `__init__`**
2. **Riferimento** ai suoi attributi: si usa la "dot notation" per acceddere ad attributi e metodi della classe

ATTENZIONE: l'oggetto associato all'espressione che precede il 'dot' viene implicitamente passato come primo parametro del metodo, e viene chiamato, per convenzione, sempre **`self`** (Vedi notebook)

Metodi con underscore __XX__

In Python esistono diversi metodi che possono essere definiti con un doppio underscore prima e dopo il nome del metodo, tipo **`__init__`**

ALTRI ESEMPI:

- **`__str__(self)`**: restiruisce una stringa, e viene per esempio chiamato in automatic quando un oggetto viene passatto alla funzione **`print()`**
- **`__add__(self, other)`**: viene utilizzato per fare l'OVERLOADING dell'operatore di addizione '+'
- **`__eq__(self, other)`**:

Inheritance

L'EREDITARIETÀ offre un meccanismo relativamente semplice per costruire gruppi di tipi (classi) collegati tra loro.

In pratica permette di costruire una **gerarchia di tipi**, in cui un dato tipo (**subclass**) può ereditare tutti gli attributi e metodi dal tipo da cui deriva (la sua **superclass**)

Per alcuni esempi e maggiori dettagli si rimanda al Capitolo 8 del libro di riferimento.

Object Oriented vs Functional Prog.

L’uso di ADT incoraggia l’analista programmatore a pensare più in termini di OGGETTI che di FUNZIONI.

Un programma diventa una collezione di TIPI invece che una collezione di FUNZIONI.

Esempio 1: Adder

- Una **classe** rappresenta dei "*dati con delle operazioni collegate*"

- Una **closure** rappresenta delle "*operazioni con dei dati collegati*"

ESEMPIO: vedi notebook

Esempio 2: Counter

- Una **classe** rappresenta dei "*dati con delle operazioni collegate*"

- Una **closure** rappresenta delle "*operazioni con dei dati collegati*"

ESEMPIO: vedi notebook

PARTE SECONDA:
HANDLING EXCEPTIONS

Handling Exceptions 1/4

Sino ad ora abbiamo visto le Exceptions come a degli errori gravi che causano un'interruzione "brutale" del nostro programma. Esempi dall prima lezione:

- **ValueError**: invalid literal for int() with base 10: '3.0'
- **AttributeError**: readonly attribute (`z.real = 3`)
- **NameError**: name 'x' is not defined (dopo `del x`)
- **TypeError**: unsupported operand type(s) for ** or pow(): 'str' and 'int'
- **TypeError**: 'int' object is not iterable

Handling Exceptions 2/4

Questi sono tutti esempi di eccezioni generate dal sistema per gestire ERRORI in parte prevedibili del programma.

- **ValueError**: invalid literal for int() with base 10: '3.0'
- **AttributeError**: readonly attribute (`z.real = 3`)
- **NameError**: name 'x' is not defined (dopo `del x`)
- **TypeError**: unsupported operand type(s) for ** or pow(): 'str' and 'int‘
- **TypeError**: 'int' object is not iterable

Handling Exceptions 3/4

Gli errori precedent sono generate dal comando

```
raise ValueError('Vostro msg errore')
```

E possono essere gestiti tramite i comandi

```
try:
        ChiamataFunzione()
except ValueError:
        print('decidete voi cosa fare, ma no crash!')
```

Handling Exceptions 4/4

Questi sono tutti esempi di eccezioni generate dal sistema per gestire ERRORI in parte prevedibili del programma. Gli errori precedent sono generate dal comando

```
raise ValueError('Vostro msg errore')
```

E possono essere gestiti tramite i comandi

```
try:
        ChiamataFunzione()
except ValueError:
        print('decidete voi cosa fare, ma no crash!')
```

LE ECCEZIONI SONO UN MODO PER SEMPLIFICARE IL FLUSSO DEL PROGRAMMA