


Gestione del codice

Introduzione

- I linguaggi dinamici mettono a disposizione meccanismi di gestione avanzata del codice
 - Permettono di adattare a runtime il comportamento del programma senza bisogno di ricompilazione
- *Veloce e flessibile adattamento a situazioni differenti*
- *Capacità di (auto-)analisi del codice a tempo di esecuzione*
 - *Generazione e/o modifica del codice a tempo di esecuzione*
 - *Gestione dinamica di funzioni speciali*
 - *Intercettazione e gestione di errori a run time*
- 
- ```
graph LR; MP[Metaprogramming] --> A[Capacità di (auto-)analisi del codice a tempo di esecuzione]; MP --> B[Generazione e/o modifica del codice a tempo di esecuzione];
```

# Metaprogramming

# Metaprogramming

- **Metaprogramming:** scrittura di un programma (metaprogramma) in grado di generare, analizzare e/o modificare altri programmi o (parti di) se stesso a run time
- **Obiettivo:** produrre programmi estremamente flessibili, in grado di reagire autonomamente rispetto a situazioni/configurazione che cambiano a run time, **senza intervento umano (scrittura di nuovo codice) né ricompilazione completa**
  - **Riuso e riduzione tempo di sviluppo del codice**
- **Implicazioni:** un metaprogramma deve poter effettuare a tempo di esecuzione alcune operazioni che in altri linguaggi sono effettuate solo a tempo di compilazione (es. *compilatore in ambiente runtime, type checking dinamico, ...*)

# Reflection

- Forma **più usata** di metaprogramming: **Reflection**
- Comune nei linguaggi dinamici, usato ma **limitatamente ad alcuni aspetti** in altri linguaggi – es. Java
- Capacità di un programma **di eseguire elaborazioni (analizzare e/o modificare) la propria struttura e il proprio comportamento a runtime**
- Es. Java: un programma in esecuzione può esaminare le classi da cui è costituito, i nomi e le signature dei loro metodi
  - Package `java.lang.reflect`"- dedicato agli strumenti che consentono a un programma di estrarre informazioni sulla propria struttura

# Ambiente run time

- La reflection viene realizzata tramite un insieme di **meccanismi suppletivi all'ambiente run time**
- L'ambiente run time deve essere in grado di:
  - 1) **Analizzare** costrutti di codice (classi, metodi) a run time (*introspection*)
  - 2) **Convertire** una stringa contenente il nome simbolico di un oggetto in un riferimento all'oggetto a run time
  - 3) **Valutare** a run time una espressione (stringa) come se fosse una sequenza di nuovo codice
- *Non tutti i linguaggi offrono tutti i meccanismi*

# Introspection

- Capacità di un linguaggio (orientato agli oggetti) di determinare tipo e proprietà di un oggetto (o di una classe) a tempo di esecuzione
  - Basato sul type checking dinamico
- Utilizzata principalmente per ispezionare classi, attributi e metodi senza bisogno di conoscerne il nome a tempo di compilazione (hard-coded nel programma)
  - Possibilità di istanziare oggetti e invocare metodi
- Es. di processo
  - Si individua il tipo di una data classe
  - Si individua la lista di metodi della classe
  - Si eseguono i metodi

# Introspection in Python

Meccanismi di auto-analisi (reflection) sulle classi

– Simili a quanto visto per Java

Accesso a classi e loro metodi senza avere nomi di entità hard-coded nel codice

– Flessibilità, manutenibilità, riuso

Strumenti:

`__dict__`      #attributo speciale di classi/moduli  
                  #Python → namespace come  
                  dizionario

`locals(), globals(), getattr()`    #funzioni built-in

`import inspect`

`inspect.isfunction(i)`      #verifica se i è un metodo



# dict

Es.

```
>>> "2".__class__
```

```
<type 'str'>
```

```
>>> "2".__class__.__dict__
```

```
dict_proxy({'upper': <method 'upper' of 'str'
objects>, '__format__': <method '__format__' of
'str' objects>, '__getslice__': <slot wrapper
'__getslice__' of 'str' objects>, 'startswith':
<method 'startswith' of 'str' objects>, 'lstrip':
<method 'lstrip' of 'str' objects>, 'capitalize':
<method 'capitalize' of 'str' objects>, '__str__':
<slot wrapper '__str__' of 'str' objects>, ...
```

# locals(), globals()

Funzioni built-in per accedere ai namespace

– locals()

Accesso al *namespace locale*: funzioni, classi e blocchi di codice

– globals()

Accesso al *namespace globale*: a livello di modulo

Entrambe restituiscono un dictionary avente come chiavi gli identificatori e come valori i valori delle entità associate

# locals(), globals()

```
>>> x = 1
>>> def foo(arg):
... y = 2
... print(locals())
... print(globals())
```

```
>>> foo(7)
{'y': 2, 'arg': 7}
```

```
{'__spec__': None, '__builtins__': <module 'builtins' (built-in)>,
'__package__': None, '__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__name__': '__main__', 'foo': <function foo at 0x7f0210b2a9d8>,
'__doc__': None, 'x': 1}
```

locals()

globals()

NOTA: diverso da dir() che restituisce solo la lista di identificatori

# locals(), globals()

Entità accessibili attraverso i costrutti disponibili sui dictionary

```
>>> for k,v in globals().items():
... print k, "=", v
```

Restituisce un riferimento alla funzione foo()

```
>>> globals()['foo']
<function foo at 0x7f0210b2a9d8>
```

```
>>> globals()['foo'](7)
```

Invoca la funzione foo()

# getattr(): recupero di attributi e metodi di una classe

getattr(instance, name)

Accede all'entità di nome **name** (*stringa*) di **instance** (*istanza di classe*)

- Se **name** è il nome di un **attributo**, restituisce il **valore**
- Se **name** è il nome di un **metodo**, restituisce il **riferimento al metodo**

*Es. In un modulo che definisce una classe MyClass(), avente un costruttore ed un metodo f() che non prevedono parametri in ingresso, come si può invocare il metodo f() senza usare nomi hard-coded (ma sotto forma di stringa?)*

# **getattr(): recupero di attributi e metodi di una classe**

**Classe MyClass con costruttore e metodo f() che non prevedono parametri in ingresso:**

```
getattr(globals()['MyClass'](), 'f')()
```

# getattr(): recupero di attributi e metodi di una classe

Classe MyClass con costruttore ed metodo f() che non prevedono parametri in ingresso:

```
getattr(globals()['MyClass'](), 'f')()
```

`globals()['MyClass']` → riferimento a classe MyClass

`globals()['MyClass']()` → torna istanza di MyClass

`getattr(globals()['MyClass'](), 'f')` → restituisce il riferimento al metodo f

`getattr(globals()['MyClass'](), 'f')()` → invoca il metodo f

# Invocazione di un metodo di classe con e senza reflection

## # without reflection

```
obj = Foo()
obj.hello()
```

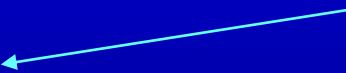
Chiamate tradizionali:  
classe Foo con  
metodo hello()



## # with reflection

```
class_name = "Foo"
method = "hello"
obj = globals()[class_name]()
getattr(obj, method)()
```

Utilizzo dei costrutti  
di reflection globals()  
e getattr()



#obj = istanza di classe  
#invocazione hello()



# Recupero dei metodi di una classe

- **Attributo speciale `obj.__dict__`**: dizionario che contiene tutti gli attributi di un oggetto `obj`
- **Se `object` è una classe, `__dict__` contiene tutti gli attributi e i metodi della classe**

```
>>> c = globals()['MyClass'] #riferimento a classe
```

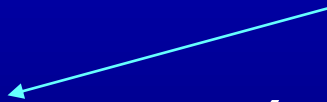
```
>>> c.__dict__
```

```
{ '__doc__': None, 'f': <function MyClass.f at
0x7f7b71201bf8>, ... }
```

```
>>> import inspect
```

```
>>> inspect.isfunction(getattr(globals()['MyClass'](
, 'f')))
```

True se passato un  
riferimento a funzione



→ESERCIZIO

[reflection/introspection/python/Person.py](#)

# Valutazione dinamica di espressioni

- **Meccanismo di valutazione dinamica di espressioni:** capacità di considerare a runtime il contenuto di una stringa come *nuovo codice*
- *Nuovo codice leggibile da file o generabile come stringa a runtime*
- Si memorizza all'interno di una stringa **l'espressione** che si vuole **valutare/eseguire**
- **Espressioni:**
  - *Espressioni matematiche e logiche*
  - *Blocchi di codice*
  - *Funzioni*
- **Si esegue o si valuta (interpretazione con restituzione del risultato) il contenuto della stringa**

# Valutazione dinamica di espressioni

## Python

- “MyClass()” è la stringa che rappresenta la creazione di una istanza di classe MyClass

```
>>> str1="MyClass()"
```

```
>>> obj=eval(str1) ?
```

```
>>> obj=exec(str1) ?
```

```
>>> str2="obj.f()"
```

```
>>> eval(str2) ?
```

```
>>> exec(str2) ?
```

# Invocazione metodo di classe (Python)

**# traditional**

```
obj = Foo()
obj.hello()
```

Chiamate tradizionali:  
classe Foo con  
metodo hello()

**# with reflection**

```
class_name = "Foo"
method = "hello"
```

Reflection: globals()  
e getattr()

```
obj = globals()[class_name]() #obj = istanza di classe
getattr(obj, method)() #invocazione obj.hello()
```

**#dynamic string evaluation**

```
class_name = "Foo"
method = "hello"
```

String evaluation  
con eval()

```
obj=eval(class_name+"()") #obj = istanza di classe
eval("obj."+method+"()") #invocazione obj.hello()
```

# Valutazione dinamica di espressioni

- **Vantaggi:**

- È possibile costruire del **codice polimorfo**, in grado di modificarsi ed adattarsi a tempo di esecuzione
  - Es. configurazioni da file

- **Svantaggi:**

- Se le stringhe rappresentanti il codice non sono ben controllate (es. inserimento attraverso form o lettura da file di configurazione), è possibile incorrere in **rischi di sicurezza**

# Gestione dinamica delle funzioni

# Funzioni anonime

- Una **funzione anonima** è una funzione che **non è legata ad un nome e ad una signature**
- Utilizzate attraverso un riferimento o passaggio di parametri
- Uso tipico: contenere funzionalità di **uso a breve termine** in un punto specifico del programma per **specializzarne** il funzionamento
- Le funzioni anonime sono disponibili solo se il linguaggio supporta **funzioni come first-class entities** (es. non supportate in Java)
  - Funzioni assegnabili a variabili, passabili come argomenti e come valori di ritorno

# Uso in algoritmo di ordinamento

- Esempio d'uso: passata come argomento a una funzione di livello superiore per specializzarne il comportamento
- Es. Funzione di livello superiore = funzione di ordinamento per gestire diversi tipi di dato e ordinare secondo diversi criteri
  - In presenza di dati strutturati, cioè composti da più elementi (es. tuple, liste, o classi) la funzione di ordinamento può agire su campi diversi del dato
- Funzione *sorted(iterable [,key][, reverse])*
  - Ritorna la lista ordinata
  - key (opzionale) rappresenta una funzione che ritorna la “chiave” su cui fare l'ordinamento (il confronto)
  - reverse (opzionale) booleano (default False)



# Uso di funzione anonima

```
>>> sorted([5, 2, 3, 1, 4])
```

```
[1, 2, 3, 4, 5]
```

```
>>> student_tuples = [('john', 'A', 15), ('jane', 'B', 17),
 ('dave', 'B', 10)]
```

```
>>> sorted(student_tuples)
```

```
[('dave', 'B', 10), ('jane', 'B', 17), ('john', 'A', 15)]
```

Ordinamento di default avviene sul primo elemento della tupla

```
>>> sorted(student_tuples, reverse=True)
```

```
[('john', 'A', 15), ('jane', 'B', 17), ('dave', 'B', 10)]
```

- Uso del parametro `key` per definire l'ordinamento su un altro elemento
  - Passiamo una **funzione anonima** come parametro `key`

# Definizione di funzione anonima

- Funzioni anonime in Python
  - Identificate dalla parola chiave **lambda** (*non def*)
  - Accettano un **numero indefinito di parametri** e sono costituite da un'unica **espressione**

• Es.

**Riferimento a funzione** points to `dividi` in the first example.

**Keyword lambda** points to `lambda` in the first example.

**Parametri** points to `dividendo` in the first example and `x,y` in the second example.

**Espressione** points to `dividendo / 5` in the first example and `x**2+y` in the second example.

```
>>> dividi = lambda dividendo: dividendo / 5
>>> print(dividi(10))
2
>>> g = lambda x,y: x**2+y
>>> g(5,8)
33
```

# Uso di funzione anonima in sorted

```
>>> student_tuples = [('john', 'A', 15), ('jane', 'B', 17),
 ('dave', 'B', 10)]
```

#ordinate in base all'elemento di indice 2 (età)

```
>>> sorted(student_tuples, key=lambda x: x[2])
[('dave', 'B', 10), ('john', 'A', 15), ('jane', 'B', 17)]
```

La funzione **key** assume di ricevere in ingresso ogni elemento dell'iterable e di ritornare il valore su cui effettuare il confronto

Altro esempio

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> sorted(pairs, key=lambda pair: pair[1])
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

# Gestione degli errori a run time

# Eccezioni software

- Tutti i linguaggi dinamici prevedono un **meccanismo per intercettare e gestire situazioni anomale (errori) a run time**
- **Meccanismo delle eccezioni software**
- Reso popolare da Java, presente in C#, *Perl*, *Python*, *Ruby*, *PHP* (da *PHP5*), *Javascript*
- La maggior parte dei linguaggi adotta per la gestione degli errori un **approccio basato su classi** (Java, Python, ...)
- Perl adotta un approccio più semplificato basato sulle **istruzioni speciali (eval e die)**

→ESEMPI

[eccezioni/perl/eval2.pl](#)

# Approccio basato su classi

- Il **runtime environment (RE)** prevede la definizione di una classe madre rappresentante la **generica eccezione software**, che contiene (almeno):
  - un **identificatore dell'istanza** di eccezione
  - una **rappresentazione dello stack** corrente
- Ciascuna anomalia a runtime viene associata ad una **sottoclasse specifica** della classe madre
- Le eccezioni possono essere sollevate:
  - **Automaticamente**, in seguito ad anomalie provocate da istruzioni a run time (es. divisione per zero)
  - **Manualmente**, dal programmatore (**throw, raise**)
- In caso di eccezione, viene eseguito il **codice di gestione** (se *non presente, gestore di default: stampa stack ed esce*)

# Costrutto Try-Except-Finally

In Python, il costrutto **try-except-finally** è simile a quello **try-catch-finally** di Java:

**try:**

    <Blocco di codice>;

**except Eccezione  $e_1$ :**

    <Gestisci anomalia>;

...

**except Eccezione  $e_n$ :**

    <Gestisci anomalia>;

**except:**

    <Gestisci anomalia di una qualunque eccezione>;

**else:**

    <Codice eseguito in assenza di eccezioni>;

**finally:**

    <Codice eseguito alla fine del try-except, in ogni caso>;

Possibile anche gestire più eccezioni  
in un unico blocco  
espresse tra parentesi:  
**except ( $e_1, e_2, \dots, e_n$ ):**

# Gestione eccezioni

**while True:**

**try:**

**x = int(input("Please enter an integer number: "))**

**break**

**except ValueError:**

**print("Oops! That was not an integer ...")**

**print("Try again...")**

**else:**

**print("Correct number")**

ValueError: sollevata da int(x)  
se x non è un numero intero

→ESEMPI

[eccezioni/python/except.py](#)



# Sollevamento manuale di eccezioni

- **Python:** parola chiave **raise**  
*raise NameException(arg1,arg2)*  
*NameException* tipo di eccezione  
(*arg1,arg2*) lista opzionale di argomenti passati
- Per il **recupero degli argomenti**  
*except NameException as inst:*  
inst contiene l'istanza dell'eccezione sollevata  
inst.args è una tupla che contiene gli argomenti passati
- **import traceback**  
**traceback.print\_exc()** #stampa stack

→ESEMPI

[eccezioni/python/raise.py](#)