

Metaprogramming in Python

Introduzione

- Il Python mette a disposizione tutta una serie di meccanismi per il metaprogramming
- Introspezione di classi e metodi
- Esecuzione di codice generato dinamicamente
- Gestione degli errori a runtime
- Tali funzionalità sono presenti sotto forma di funzioni builtin e di moduli

Introspezione a livello di sistema

- E' possibile invocare funzionalità di introspezione già a livello di interprete da riga di comando
- L'interprete interattivo ci fornisce informazioni sui moduli installati e sulle loro caratteristiche
- Modulo di sistema **sys**
- Fornisce informazioni sull'interprete e sull'ambiente operativo
- **import sys**

Introspezione a livello di sistema

- Alcuni esempi di introspezione
 - Nome dell'eseguibile “interprete”: `sys.executable`
 - Versione dell'eseguibile “interprete”: `sys.version`
 - Nome della piattaforma:
`sys.platform`
 - Limiti di sistema:
`sys.maxint`, `sys.maxsize`, `sys.maxunicode`
 - Argomenti da linea di comando:
`sys.argv`
 - Percorso di ricerca dei moduli: `sys.path`

Introspezione a livello di sistema

- Una delle variabili esportate da `sys` permette di ottenere l'insieme dei nomi dei moduli visibili dall'interprete
- **`sys.modules`**: dizionario contenente coppie chiave-valore, in cui:
 - alla chiave corrisponde il nome completo del modulo
 - al valore corrisponde il percorso completo della versione compilata del modulo

Introspezione a livello di modulo

▪ ESEMPI:
Person.py

- La funzione **dir()** elenca i simboli di un simbolo generico
- Simbolo:
 - Nome modulo (oggetto di tipo classe)
 - Nome istanza oggetto
 - Variabile
 - Funzione
- Se invocata senza alcun argomento, **dir()** restituisce i simboli attualmente visibili all'interprete

Introspezione degli oggetti

- E' possibile estrarre informazioni specifiche dalle classi e dagli oggetti
 - Nome
 - Tipo di dato dell'oggetto
 - Identità
 - Metodi ed attributi
 - Relazioni con altre classi

Introspezione degli oggetti

▪ ESEMPI:
type.py
id.py

- Nome di un simbolo associato ad un oggetto:
simbolo privato `__name__`
 - `nome_simbolo.__name__`
- Tipo di un simbolo associato ad un oggetto:
funzione `type()`
 - `type(object)`
- Id (identità) di un oggetto: funzione `id()`
 - `id(object)`
- Confronto di identità: funzione `is()`
 - `obj_a is obj_b`
 - Restituisce True se i due oggetti hanno lo stesso id

Introspezione degli oggetti

▪ ESEMPI:
attr.py

- Controllo esistenza di uno specifico attributo in un oggetto: funzione **hasattr()**
 - **hasattr(object, attribute)**
 - Restituisce True se object ha l'attributo attribute
 - L'attributo può essere una variabile/funzione
- Recupero di uno specifico attributo da un oggetto: funzione **getattr()**
 - **getattr(object, attribute)**
 - Equivalente ad **object.attribute**
- Attribute può essere il contenuto di una variabile stringa; pertanto, gli attributi possono essere specificati dinamicamente!

- **Controllo invocabilità di un oggetto: funzione `callable()`**
 - `callable(object)`
 - Restituisce True se object è invocabile
- **Relazioni con altre classi:**
 - metodi `isinstance()` e `issubclass()`

Subroutine anonime

- E' possibile assegnare a variabili delle funzioni (che vengono perciò chiamate **subroutine anonime**)
- In Python, la funzioni anonime sono chiamate **funzioni lambda**
 - In onore ad **Alonzo Church** (1903-1995), ideatore del lambda-calcolo e precursore dei linguaggi funzionali

Subroutine anonime

▪ ESEMPI:
switch.py

- Una funzione anonima è definita tramite la parola chiave **lambda**
variabile = lambda lista_var : espressione
- Ad esempio, la seguente assegnazione associa alla variabile **f** la funzione anonima x^2
f = lambda x: x^2
- La variabile **f** punta all'oggetto “funzione anonima” rappresentato dall'espressione x^2
 - **type(f)** restituisce, conseguentemente, il tipo **function**

Closure

▪ ESEMPI:
closure1.py
closure2.py

- In Python, le closure (blocchi di codice cui sono associati una o più variabili) possono essere implementate in due modi diversi:
 - definendo una funzione che restituisce una funzione lambda
 - definendo una funzione che a sua volta definisce e restituisce un'altra funzione

Generazione dinamica di codice

▪ ESEMPI:
eval.py
exec.py
execfile.py

- Il Python mette a disposizione meccanismi per valutare espressioni e statement
 - **eval(expr)**: l'espressione viene analizzata sintatticamente e valutata
 - **exec(stmt)**: lo statement viene analizzato sintatticamente ed interpretato
 - **execfile(file)**: il file viene letto, analizzato sintatticamente ed interpretato
- E' possibile passare due ulteriori parametri: **globals** (un dizionario di variabili globali), e **locals** (un dizionario di variabili locali)
 - **eval(expr, globals, locals)**

Generazione dinamica di codice

▪ ESEMPI:
compile.py

- Se la stessa stringa deve essere eseguita più volte, conviene precompilarla
- **compile(source, filename, mode)**
 - Compila la stringa source (o, in alternativa, il file di nome filename)
 - Il parametro **mode** specifica l'identità dell'oggetto prodotto:
 - 'exec' → sequenza di statement
 - 'eval' → espressione
 - 'single' → un singolo statement interattivo
- Viene restituito un code object che può essere valutato con **eval()** o interpretato con **exec()**

Gestione degli errori a runtime

▪ ESEMPI:
exc1.py

- Il Python mette a disposizione il costrutto **try-except-else-finally** per gestire le eccezioni
- **try:**
 - blocco di codice
- **except *lista_eccezioni*:**
 - blocco di codice
- ...
- **else: # eseguito in assenza di eccezioni**
 - blocco di codice
- **finally: # eseguito sempre**
 - blocco di codice

Gestione degli errori a runtime

▪ ESEMPI:
exc2.py

- E' possibile ignorare alcune eccezioni associando la relativa lista di nomi allo statement nullo **pass**

try:

blocco di codice

except *lista_eccezioni*:

pass

Gestione degli errori a runtime

▪ ESEMPI:
exc3.py
Myerror.py

- Le eccezioni possono essere invocate dal programmatore tramite l'istruzione **raise**
raise NomeEccezione(Lista_Argomenti)
- L'elenco delle eccezioni è consultabile al seguente indirizzo:
<http://docs.python.org/library/exceptions.html>
- E' possibile definire nuove eccezioni estendendo la classe **Exception**