Programmazione Avanzata

Il modulo inspect

1

Il modulo inspect

- Il modulo inspect fornisce diverse utili funzioni per aiutare a ottenere informazioni riguardanti oggetti ”vivi” come moduli, classi, metodi, funzioni, traceback, oggetti frame e oggetti codice.
- Per esempio, puo` essere d’aiuto per esaminare i contenuti di una classe, accedere al codice sorgente di un metodo, estrarre e formattare la lista di argomenti di una funzione, o ottenere tutte le informazioni necessarie per mostrare un traceback dettagliato.
- Ci sono quattro principali tipi di servizi forniti da questo modulo: effettuare il type checking, prelevare il codice sorgente, ispezionare classi e funzioni, esaminare lo stack dell’interprete.

2

Il metodo inspect.getmembers

inspect.**getmembers**(*object*[, *predicate*])

- Restituisce tutti i membri di un oggetto in una lista di coppie (name, value).
- Se viene fornito anche l'argomento opzionale *predicate*, questo viene invocato con l'oggetto value di ciascuna coppia e vengono incluse nella lista solo le coppie per le quali il predicato restituisce un valore true.

3

Predicati del modulo inspect

ismodule()	Return true if the object is a module.
isclass()	Return true if the object is a class, whether built-in or created in Python code.
ismethod()	Return true if the object is a bound method written in Python.
isfunction()	Return true if the object is a Python function, which includes functions created by a lambda expression.
isgenerator()	Return true if the object is a generator.
iscode()	Return true if the object is a code.
isbuiltin()	Return true if the object is a built-in function or a bound built-in method.
isabstract()	Return true if the object is an abstract base class.

4

Predicati del modulo inspect

```
'''Docstring del modulo'''

import inspect
def funzione():
  '''Docstring di funzione'''
  print ('Sono funzione')

class Base:
  '''Docstring di Base'''
  def __init__(self):
    self.var='Sono funzione'
  def funzione(self):
    print (self.var)
class Derivata(Base):
  def funzione(self):
    super().funzione()
    print ("Sono funzione di Derivata")
```

modulo.py

```
import modulo
import inspect
import sys
for k,v in inspect.getmembers(sys.modules["modulo"], inspect.isclass):
    print (k,v)
```

```
Base <class 'moduloinspect.Base'>
Derivata <class 'moduloinspect.Derivata'>
```


5

Il metodo inspect.getmodulename

inspect.**getmodulename**(*path*)

- Restituisce il nome del modulo (senza l'estensione) indicato da path (stringa che specifica l'intero path per arrivare al modulo) senza includere i nomi dei pacchetti. Viene controllata l'estensione con tutte le entrate di importlib.machinery.all_suffixed(). Se l'estensione corrisponde a una presente nella lista restituita da importlib.machinery.all_suffixed() (tale lista contiene le estensioni dei file importabili con import) allora viene restituita la componente finale della path senza l'estensione altrimenti viene restituito None.

6

Il metodo inspect.getdoc

inspect.**getdoc**(*object*)

- Restituisce la stringa di documentazione per un oggetto ripulita degli spazi di indentazione da cleandoc().
- Se per l'oggetto non e` fornita una stringa di documentazione e l'oggetto e` una classe, un metodo, una property o un descrittore allora la stringa di documentazione e` ottenuta dalla gerarchia.
- Restituisce None se la documentazione e` assente o se e` errata.

inspect.**getcomments**(*object*):

- Restituisce in una singola stringa le linee di commento che precedono il codice sorgente dell'oggetto (per una classe, metodo o funzione) o quelli in alto nel file sorgente (se l'oggetto e` un modulo).
- Se il codice sorgente non e` disponibile (ad esempio se l'oggetto e` definito in C o da shell) allora viene restituito None.

7

Predicati del modulo inspect

```
'''Docstring del modulo'''


import inspect
def funzione():
  '''Docstring di funzione'''
  print ('Sono funzione')

class Base:
  '''Docstring di Base'''
  def __init__(self):
    self.var='Sono funzione'
  def funzione(self):
    print (self.var)
class Derivata(Base):
  def funzione(self):
    super().funzione()
    print ("Sono funzione di Derivata")
```

modulo.py

```
#script da eseguire
import modulo
import inspect
import sys
print(inspect.getdoc(modulo))
print(inspect.getdoc(modulo.Base))
print(inspect.getdoc(modulo.funzione))
print(inspect.getcomments(sys.modules[__name__]))
```

```
Docstring del modulo
Docstring di Base
Docstring di funzione
#script da eseguire
```


8

Il metodo inspect.getfullargs

inspect.**getfullargspec**(*func*)

- restituisce i nomi e i valori di default dei parametri di una funzione Python. Restituisce una nametuple
  FullArgSpec(args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations)
- *args* è una lista dei nomi dei parametri posizionali
- *varargs* è il nome del parametro * o None in assenza del parametro preceduto da *
- *kwonlyargs* è una lista di parametri keyword-only nell'ordine in cui sono dichiarati.
- *varkw* è il nome del parametro ** o None in assenza del parametro preceduto da **
- *defaults* è una n-upla di argomenti di default corrispondentti agli ultimi n argomenti posizionali
- *kwonlydefaults* è un dizionario che mappa i nomi dei parametri da *kwonlyargs* ai valori di default usati se nessun argomento viene fornito.
- *annotations* è un dizionario che mappa i nomi dei parametri alle annotazioni. La chiave speciale "return" è usata per l'annotazione del valore di return.

9

Il metodo inspect.getfullargs: un esempio

```
def strictly_typed(function):
   annotations = function.__annotations__
   arg_spec = inspect.getfullargspec(function)
   assert "return" in annotations, "missing type for return value"
   for arg in arg_spec.args + arg_spec.kwonlyargs:
      assert arg in annotations, ("missing type for parameter ‘” + arg + "'")
   @functools.wraps(function)
   def wrapper(*args, **kwargs):
      for name, arg in (list(zip(arg_spec.args, args)) + list(kwargs.items())):
         assert isinstance(arg, annotations[name]), (
               "expected argument '{0}' of {1} got {2}".format(
               name, annotations[name], type(arg)))
      result = function(*args, **kwargs)
      assert isinstance(result, annotations["return"]), (
               "expected return of {0} got {1}".format(
               annotations["return"], type(result)))
      return result
   return wrapper
```


10

Il metodo inspect.getfullargs: un esempio

- La funzione wrapper nel decoratore strictly_typed comincia iterando su ogni coppia nome-argomento.
- Siccome zip restituisce un iteratore e dictionary.items() restituisce una view non li possiamo concatenare direttamente e per questo li convertiamo in liste.
- Se un qualsiasi argomento con cui è invocata la funzione ha un tipo diverso da quello specificato dall'annotazione si ha AssertionError; in caso contrario viene invocata la funzione originaria è viene effettuato un controllo sul tipo del valore restituito e se è del tipo giusto viene restituito.

```
@functools.wraps(function)
  def wrapper(*args, **kwargs):
    for name, arg in (list(zip(arg_spec.args, args)) + list(kwargs.items())):
      assert isinstance(arg, annotations[name]), (
          "expected argument '{0}' of {1} got {2}".format(
          name, annotations[name], type(arg)))
    result = function(*args, **kwargs)
    assert isinstance(result, annotations["return"]), (
          "expected return of {0} got {1}".format(
          annotations["return"], type(result)))
    return result
  return wrapper
```


11

Il metodo inspect.getfullargs: un esempio

```
@strictly_typed
def range_of_floats(a:int,b:int,c:int) -> types.GeneratorType:
   return (float(x) for x in range(a,b,c))

for x in range_of_floats(1,8,'a'):
   print(x)
```

```
Traceback (most recent call last):
  File "/Users/adb/Documents/strictly_typed_decorator.py", line 31, in <module>
    for x in range_of_floats(1,8,'a'):
  File "/Users/adb/Documents/strictly_typed_decorator.py", line 16, in wrapper
    assert isinstance(arg, annotations[name]), (
AssertionError: expected argument 'c' of <class 'int'> got <class 'str'>
```


12

Altri metodi per accedere al codice sorgente

- inspect.**getfile**(*object*): restituisce il nome del file in cui un oggetto e` stato definito. Lancia TypeError se l'oggetto e` un modulo, una classe o una funzione built-in.
- inspect.**getmodule**(*object*): prova ad indovinare in quale modulo un oggetto e` stato definite. Restituisce None se non riesce ad individuare il modulo.
- inspect.**getsourcefile**(*object*): restituisce il nome del file sorgente Python nel quale un oggetto e` stato definite oppure None se non c'e` modo di individuare la sorgente. Lancia TypeErrror se l'oggetto se l'oggetto e` un modulo, una classe o una funzione built-in.
- inspect.**getsource**(`object`): restituisce in una singola stringa il testo del codice sorgente di un oggetto. L'argomento puo` essere un modulo, una classe, un metodo, una funzione, un traceback, un frame, o un oggetto code. Viene lanciato OSError se il codice sorgente non puo` essere recuperato.
- ecc.

13

Il metodo inspect.signature

inspect.**signature**(`callable`, `*`, `follow_wrapped=True`, `globals=None`, `locals=None`, `eval_str=False`)

- Restituisce un oggetto Signature per *callable*
- Gli argomenti `follow_wrapped`, `globals`, `locals`, `eval_str` sono passati a inspect.get_signature per risolvere le annotazioni nel caso in cui si utilizzi `from __future__ import annotations`
- L'oggetto Signature rappresenta una chiamata di un callable e la sua annotazione per il return.

```
>>> from inspect import signature
>>> def foo(a, *, b:int, **kwargs):
...     pass

>>> sig = signature(foo)

>>> str(sig)
'(a, *, b:int, **kwargs)'

>>> str(sig.parameters['b'])
'b:int'

>>> sig.parameters['b'].annotation
<class 'int'>
```


14

La classe inspect.Signature

class inspect.**Signature**(*parameters=None*, *, *return_annotation=Signature.empty*)

- Per ogni parametro accettato dalla funzione l'oggetto Signature memorizza un oggetto Parameter nella sua collezioni di parametri **parameters** che è un mapping ordinato di coppie (nome parametro, oggetto Parameter). L'ordine è quello in cui sono definiti.
- Per settare **parameters** viene utilizzato l'argomento opzionale *parameters* che è una sequenza di oggetti parameter che viene validata verificando che non vi siano nomi di parametri duplicati, che i parametri siano nell'ordine corretto, cioe` prima quelli solo posizionali e poi quelli che possono essere considerati posizionali o keyword, e che i parametri con valori di default seguano quelli senza valori di default.
- L'argomento *return_annotation* è l'annotazione per il valore di return. Se il callable non ha l'annotazione per il return allora l'argomento e quindi l'attributo **return_annotation** della signature è settato a Signature.empty.

15

I metodi bind e bind_partial di Signature

bind(**args*, ***kwargs*)

- Crea un mapping dagli argomenti posizionali o keyword ai parametri.
- Se *args e **kwargs corrispondono ai parametri nella signature su cui e ` invocato, bind restituisce un'istanza di inspect.**BoundArguments** che mantiene l'associazione tra parametri e argomenti. In caso contrario lancia TypeError.

bind_partial(**args*, ***kwargs*)

- Funziona allo stesso modo di Signature.bind() ma permette di omettere alcuni argomenti.
- Se *args e **kwargs corrispondono alla signature su cui e ` invocato, bind restituisce un'istanza di inspect.**BoundArguments** che mantiene l'associazione tra parametri. In caso contrario lancia TypeError

16

I metodi bind e bind_partial: un esempio

```
import inspect
def bind_arguments(func,*args) -> inspect.BoundArguments:
      """ Controlla se gli argomenti args dati rispettano la
signature"""
      try:
         return inspect.signature(func).bind(*args)
      except TypeError as e:
         print("argomenti non corrispondenti alla signature")


def f(x):
   return x+4

print(bind_arguments(f,3))
print(bind_arguments(f,4,3))
```

```
<BoundArguments (x=3)>
argomenti non corrispondenti alla signature
None
```


17

I metodi bind e bind_partial: un esempio

```
import inspect
def bind_partial_arguments(func,*args) ->
inspect.BoundArguments:
      """ Controlla se gli argomenti args dati rispettano la
signature"""
      try:
         return inspect.signature(func).bind_partial(*args)
      except TypeError as e:
         print("argomenti non corrispondenti alla signature")

def f(x,y):
   return x+y

print(bind_partial_arguments(f,3))
print(bind_partial_arguments(f,4,3))
```

```
<BoundArguments (x=3)>
<BoundArguments (x=4, y=3)>
```


18

Il metodo replace di Signature

Gli oggetti Signature sono immutable. Il metodo replace di inspect.Signature serve per creare una copia modificata di un oggetto Signature.

replace(**[, parameters][, return_annotation]*)

- Crea una nuova istanza di Signature basata sull'istanza di Signature su cui replace è invocata.
- È possibile passare parametri e/o return_annotation differenti per sovrascrivere quelli della signature di partenza. Per rimuovere return_annotation dalla signature occorre passare Signature.empty.

```
>>> def test(a, b):
...     pass
>>> sig = signature(test)
>>> new_sig = sig.replace(return_annotation="new return anno")
>>> str(new_sig)
"(a, b) -> 'new return anno'"
```


La classe inspect.Parameter

class inspect.**Parameter**(*name*, *kind*, ***, *default=Parameter.empty*, *annotation=Parameter.empty*)

- Gli oggetti Parameter sono immutable. Per modificare un oggetto Parameter si usa Parameter.replace() che crea una copia modificata dell'oggetto.

- Gli attributi di Parameter:
- **empty:** un marcatore speciale per specificare l'assenza di valori di default e annotazioni.
- **name:** il nome del parametro (stringa). Il nome deve essere un identificatore valido.
- **default:** il valore di default per il parametro. Se il parametro non ha valore di default questo attributo è settato a Parameter.empty.
- **annotation**: l'annotazione per il parametro. Se il parametro non ha annotazione, questo attributo è settato a Parameter.empty.
- **kind:** descrive come i valori degli argomenti vengono associati ai parametri

La classe inspect.Parameter

```
from inspect import  signature

def foo(a, b, *, c, d=10):
   pass

sig = signature(foo)
for param in sig.parameters.values():
    print('Parameter {}: {}'.format(param,param.kind))
```

import inspect

```
Parameter a: POSITIONAL_OR_KEYWORD
Parameter b: POSITIONAL_OR_KEYWORD
Parameter c: KEYWORD_ONLY
Parameter d=10: KEYWORD_ONLY
```


La classe inspect.Parameter

Possibili valori di Parameter.kind:

Name	Meaning
POSITIONAL_ONLY	Value must be supplied as a positional argument. Positional only parameters are those which appear before a `/` entry (if present) in a Python function definition.
POSITIONAL_OR_KEYWORD	Value may be supplied as either a keyword or positional argument (this is the standard binding behaviour for functions implemented in Python.)
VAR_POSITIONAL	A tuple of positional arguments that aren't bound to any other parameter. This corresponds to a `*args` parameter in a Python function definition.
KEYWORD_ONLY	Value must be supplied as a keyword argument. Keyword only parameters are those which appear after a `*` or `*args` entry in a Python function definition.
VAR_KEYWORD	A dict of keyword arguments that aren't bound to any other parameter. This corresponds to a `**kwargs` parameter in a Python function definition.

I metodi inspect.stack inspect.trace

inspect.**stack**()

- restituisce una lista di oggetti FrameInfo per lo stack del caller. La prima entrata nella lista rappresenta il caller; l'ultima entrata rappresenta la chiamata piu` esterna nello stack.

inspect.**trace**()

- restituisce una lista di oggetti FrameInfo per lo stack tra il frame corrente e quello in cui un'eccezione che si sta gestendo è stata lanciata. La prima entrata nella lista rappresenta il caller; l'ultima entrata rappresenta dove è stata lanciata l'eccezione.

La classe inspect.FrameInfo

class inspect.**FrameInfo**
è una sorta di tupla con i seguenti campi:

- **frame:** l'oggetto frame a cui corrisponde il record.
- **filename:** nome del file associato al codice che è eseguito dal frame che corrisponde a questo record
- **lineno:** il numero di linea della linea corrente associata al codice che è eseguito dal frame corrispondente a questo record.
- **function**: Il nome della funzione che è stata eseguita dal frame a cui corrisponde questo record
- **code_context:** una lista di linee dal codice sorgente eseguito dal frame corrispondente a questo record.
- **index:** L'indice della linea corrente che è eseguita nella lista del code_context.
- **positions:** un oggetto dis.Positions contenente il numero della linea iniziale, quello della linea finale, lo spazio davanti la colonna iniziale e quello davanti la linea finale associate all'istruzione eseguita dal frame corrispondente a questo record.
- Il modulo dis supporta l'analisi del bytecode di CPython.
  class dis.**Positions** contiene i seguenti campi (alcuni sono None se l'informazione non è disponibile): **lineno, end_lineno,col_offset , end_col_offset**

I metodi inspect.**getgeneratorstate** e inspect.**getgeneratorlocals**

I seguenti metodi sono utili per determinare quando un generatore è in esecuzione o se è in attesa di cominciare l'esecuzione o di riprenderla, o se è gia` terminato.

inspect.**getgeneratorstate**(*generator*) restituisce lo stato di un generatore-iteratore..
Possibili stati:
•GEN_CREATED: In attesa di cominciare l'esecuzione.
•GEN_RUNNING: In esecuzione.
•GEN_SUSPENDED: Sospeso ad una espressione.
•GEN_CLOSED: Esecuzione completata.

inspect.**getgeneratorlocals**(*generator*) restituisce il mapping tra le variabili "in vita" nel generatore e i loro valori correnti, cioe` un dizionario di coppie (nome,valore). Cio` equivale ad invocare locals() dall'interno del generatore.
Se *generator* è un generatore senza un frame associato allora viene restituito un dizionario vuoto.
Viene lanciata TyperError se *generator* non è un oggetto generatore.

25

I metodi inspect.**getgeneratorstate e** inspect.**getgeneratorlocals: un esempio**

```
import inspect
def gen():
    for i in range(10):
        X = yield i
        print("questo e` X nel generatore dopo che riprende l'esecuzione:",X)
generatore = gen()
print("Questo e` lo stato di generatore:",
    inspect.getgeneratorstate(generatore))
print("Queste sono le variabili locali in generatore:",
    inspect.getgeneratorlocals(generatore))
print("invoco next(generatore)")
print("next(generatore) produce: ",next(generatore))
print("Questo e` lo stato di generatore:",
    inspect.getgeneratorstate(generatore))
print("Queste sono le variabili locali in generatore:",
    inspect.getgeneratorlocals(generatore))
print("invoco send(77)")
print("send(77) restituisce il valore fornito dall'espressione yield nella seconda iterazione:",generatore.send(77))
print("Questo e` lo stato di generatore:",
    inspect.getgeneratorstate(generatore))
print("Queste sono le variabili locali in generatore:",
    inspect.getgeneratorlocals(generatore))
```

```
Questo e` lo stato di generatore: GEN_CREATED
Queste sono le variabili locali in generatore: {}
invoco next(generatore)
next(generatore) produce:  0
Questo e` lo stato di generatore: GEN_SUSPENDED
Queste sono le variabili locali in generatore: {'i': 0}
invoco send(77)
questo e` X nel generatore dopo che riprende l'esecuzione: 77
send(77) restituisce il valore fornito dall'espressione yield nella
seconda iterazione: 1
Questo e` lo stato di generatore: GEN_SUSPENDED
Queste sono le variabili locali in generatore: {'i': 1, 'X': 77}
```


26

Il metodo inspect.getattr_static

- Funzionalita` per accedere agli attributi in modo statico.
- Sia getattr() che hasattr() possono innescare l'esecuzione del codice quando ricercano il valore di un attributo o ne verificano l'esistenza.
- Nel caso in cui si desidere un'introspezione passiva, come quando si vuole accedere alla documentazione, questo comportamento potrebbe non essere conveniente e si puo` ricorrrere ai seguenti metodi:
- inspect.**getattr_static**(*obj*, *attr*, *default=None*) recupera gli attributi senza innescare la ricerca dinamica attraverso __getattr__() o __getattribute__().
- Si noti che questa funzione potrebbe non essere in grado di recuperare tutti gli attributi recuperabili da getattr() (come quelli creati dinamicamente) ma potrebbe trovare attributi non recuperabili da getattr() (come descrittori che lanciano AttributeError). Questa funzione puo` restituire descrittori invece che membri dell'istanza.

27

Il metodo inspect.getattr_static

```
import inspect
class foo_function:

        __slots__ = ['foo']
result = inspect.getattr_static(foo_function(), 'foo')
print(result)
result = getattr(foo_function(), 'foo')
print(result)
```

test_getattr_static.py

```
<member 'foo' of 'foo_function' objects>
Traceback (most recent call last):
  File "/Users/adb/Documents/didattica2/didattica/progAv2022/test_getattr_static.py", line 7, in
<module>
    result = getattr(foo_function(), 'foo')
AttributeError: 'foo_function' object has no attribute 'foo'
```


28

Il metodo inspect.getmembers_static

inspect.**getmembers_static**(*object*[, *predicate*])

- Restituisce tutti I membri di un oggetto in una lista di coppie (name,value) ordinate in base a name senza innescare la ricerca dinamica con __getattr__ o __getattribute__.
- Se viene fornito predicate allora vengono restituiti solo i membri che soddisfano iI predicato.
- getmembers_static potrebbe non ruscire a fornire tutti i membri forniti da getmembers (come gli attribute creati dinamicamente) e puo` trovare membri che getmembers non puo` trovare (come I descrittore che lanciano AttributeError). Inoltre in alcuni casi potrebbe resituire oggetti descriptor invece che istanze.

La classe inspect.Traceback

class inspect.**Traceback**
filename: Il nome del file associato al codice eseguito dal frame a cui corrisponde questo traceback.
lineno: il numero di linea della linea corrente associata al codice che è eseguito dal frame corrispondente a questo traceback.
function: Il nome della funzione che è stata eseguita dal frame a cui corrisponde questo traceback.
code_context: una lista di linee dal codice sorgente eseguito dal frame corrispondente a questo traceback.
index: L'indice della linea corrente che è eseguita nella lista del code_context.
positions: un oggetto dis.Positions contenente il numero della linea iniziale, quello della linea finale, lo spazio davanti la colonna iniziale e quello davanti la linea finale associate all'istruzione eseguita dal frame corrispondente a questo traceback.

La classe inspect.Traceback

inspect.**getframeinfo**(*frame*, *context=1*) Ottiene l'informazione riguardante un frame o di un oggetto traceback. Restituisce un oggetto traceback.
inspect.**getouterframes**(*frame*, *context=1*) Restituisce una lista di oggetti Frameinfo per *frame* e per tutti i frame piu` esterni le cui invocazioni hanno portato alla creazione di *frame.* Nella lista i frame sono ordinati a partire da *frame* fino ad arrivare al frame piu` esterno nello stack di *frame*.
inspect.**getinnerframes**(*traceback*, *context=1*) Restituisce una lista di oggetti Frameinfo per *il frame traceback* e per tutti i frame piu` interni le cui invocazioni sono una conseguenza di *frame.* Nella lista i frame sono ordinati a partire da *traceback*; l'ultima rappresenta il punto dove l'eccezione è stata lanciata.
inspect.**currentframe**() restituisce l'oggetto frame per il frame che effettua l'invocazione.

Questa funzione si basa sul supporto Python per i frame dello stack dell'interprete. Tale supporto esiste in CPython ma non è detto che esista in altre implementazioni, nel qual caso currentframe() restituisce None.

inspect.**stack**(*context=1*) restituisce una lista di oggetti Frameinfo per lo stack del frame che effettua l'invocazione. Nella lista i frame sono ordinati a partire dal frame che effettua l'invocazione fino ad arrivare al frame piu` esterno nello stack.