Advanced Programming

Federico Bruzzone

4 ottobre 2022

Indice

1

1.1 Informazioni generali

Scopo del corso

- Scoprire il concetto di separazione dei compiti;
- Imparare a programmare decomponendo le funzionalità del SW;
- Imparare ad ottimizzare il SW separandone le funzionalità;

Materiale di riferimento

- i licidi del corso;
- Ira R. Forman and Note B. Forman. Java Reflection in Action Manning Publications, October 2004;
- Ramnivas Laddad. AspectJ in Action: Pratical Aspect-Oriented Programming. Manning Pubblications Company, 2003;

1.1.1 How to use Python

We are condidering Python 3+

- version > 3 is incompatible with previus version;
- version 2.7 is the current version.

A python program can be:

- edited in the python shell and executed step-by-step by the shell;
- edited and run through the iterpreter.

1.2 Overview of the Basic Concepts

1.2.1 Our first Python program

```
SUFFIXES = {1000: ['KB', 'MB', 'GB', 'TB', 'PB', 'EB', 'ZB', 'YB'],
            1024: ['KiB', 'MiB', 'GiB', 'TiB', 'PiB', 'EiB', 'ZiB', 'YiB']}
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
  ''' Convert a file size to human-readable form. '''
  if size < 0:
    raise ValueError('number must be non-negative')
  multiple = 1024 if a_kilobyte_is_1024_bytes else 1000
```

```python
    for suffix in SUFFIX[multiple]:
        size /= multiple
        if size < multiple:
            return '{0:.1f} {1}'.format(size, suffix)
        raise ValueError('number too large')

if __name__ == '__main__':
    print(approximate_size(1000000000000, False))
    print(approximate_size(1000000000000))
```

Listing 1: humanize.py

1.2.2 Declaring function

Python has function

- no header files à la C/C++;
- no interface/implementation à la Java.

```python
def approximate_size(size, a_kilobyte_is_1024_bytes=True):
```

1. **def**: function definition keyword;
2. **approximate_size**: function name;
3. **a_kilobyte_is_1024_bytes**: comma separate argument list;
4. =**True**: default value.

Python has function

- no return type, it always return a value (**None** as a default);
- no parameter types, the interpreter figures out the parameter type.

1.2.3 Calling Functions

Look at the bottom of the *humanize.py* program

```python
1 if __name__ == '__main__':
2     print(approximate_size(1000000000000, False))
3     print(approximate_size(1000000000000))
```

2 in this call to **approximate_size()**, the **a_kilobyte_is_1024_bytes** parameter will be **False** since you explicitly pass it to the function;

3 in this row we call **approximate_size()** with only a value, the parameter **a_kilobyte_is_1024_bytes** will be **True** as defined in the function declaration.

Value can be passed by name as in:

```
def approximate_size(a_kilobyte_is_1024_bytes=True, size=1000000000000)
```

Parameters' order is not relevant

1.2.4 Writing readable code

Documentation Strings A python function can be documented by a documentation string (docstring for short).

''' Convert a file size to human-readable form. '''

Triple quotes delimit a single multi-string

- if it immediatly follows the function's declaration it is the doc-string associated to the function;
- docstrings can be retrieved at run-time (they are attributes).

Case-Sensitive All names in Python are case-sensitive

1.2.5 Everything is an object

Everything in Python is an object, functions included

- **import** can be used to load python programs in the system as modules;
- the dot-notation gives access to the the public functionality of the imported modules;
- the dot-notation can be used to access the attributes (e.g., the **__doc__**)
- **humanizeåpproximate_size.__doc__** gives access to the docstring of the **approximate_size()** function; the docstring is stored as an attribute.

1.2.6 Everything is an object (Cont'd)

In python is an object, better, is a first-calss object

- everything can be assigned to a variable or passed as an argument

```
h1 = humanize.approximate_size(9128)
h2 = humanize.approximate_size
```

- **h1** contains the string calculated by **approximate_size(9128**;
- **h2** contains the "function" object **approximate_size()**, the result is not calculated yet;
- to simplify the concept: **h2** can be considered as a new name of (alias to) **approximate_size**.

1.2.7 Indenting code

No explicit block delimiters

- the only delimiter is a column (':') and the code indentation;
- code blocks (e.g., functions, if statements, loops, ...) are defined by their indentation;
- white spaces and tabs are relevant: use them consistently;
- indentation is checked by the compiler.

1.2.8 Exceptions

Exceptions are Anomaly Situations

- C encourages the use of return codes which you check;
- Python encourages the use of exceptions which you handles.

Raising Exceptions

- the **raise** statement is used to rise an exception as in:

  ```
  raise ValueError('number must be non-negative')
  ```

- syntax recalls function calls: **raise** statement followed by an exception name with an optional argument;
- exceptions are relized by classes.

No need to list the exceptions in the function declaration handling Exceptions

- an exception is handled by a **try** ... **except** block.

```
try:
  from lxml import etree
except ImportError:
  import xml.etree.ElementTree as etree
```

1.2.9 Running scripts

Look again, at the bottom of the *humanize.py* program:

```
if __name__ == '__main__':
  print(approximate_size(1000000000000, False))
  print(approximate_size(1000000000000))
```

Modules are Objects

- they have a built-in attribute **__name__**

The value of __name__ depends on how you call it

- if imported it contains the name of the file without path and extension.

2 Computational Reflection

2.1 Computational Reflection

2.1.1 A first definition

Computational reflection can be intuitively defined as:

"The activity done by a SW system to represent and manipulate its own structure and behavior"

The reflective activity is done analogously to the usual system activity

2.2 Reflection

2.2.1 Historical Overview

In the sisties

- Research field: artificial intelligence;
- First approaches to relection: intelligent behavior;

In the eighties

- Research filed: programming languages;
- Brian C. Smith, he introduces the reflection in Lisp (1982 and 1984), the reflective tower has been defined;
- Several reflective list-oriented languages have been defined (they exploit the quoting machanism);

In the meanwhile

- Research field: logic programming;
- the meta-programming takes place in PROLOG;

Between the eighties and the nineties

- Research fild: object-oriented programming languages;
- Pattie maes defines the computational reflection in OOPL (1987);
- Several people move from Lisp to OO:
  - P. Coite, ObjVLips (1987)
  - A. Yonezawa, ABCL-R (1988)
  - J. des Rivières e G. Kiczales MOP for CLOS (1991)
- SmallTalk is elected as the best reflective programming language

In te nineties

- Research field: typed and/or compiled object-oriented programming languages;
- Shigeru Chiba realizes OpenC++ (1993-1995), OpenJava (1999);

In the 1997

- Gregor Kiczales et al. defined the aspect-oriented programming and the story ends;

2.3 Computational Reflection

2.3.1 Reflection à la Pattie Maes

Pattie Maes has pioneered the filed

- a **computational system** is a system that can reason about and act on its applicative somain;
- a computational system is **causally connected** to its domain if and only if a change to its domain is reflected on it and vice versa;
- a **meta-system** is a computational system whose applicative domain in another computational system;
- **reflection** is the property of reasoning about and acting on itself;

therefore

- a **reflective system** is a meta system causally connected to itself;

2.3.2 Reflective system

From the definition, we can evince that a reflective system is:

- a software system logically layered into two or more levels respectively called base-level and meta-levels;
- the system running in a meta-level observes and manipulates the system running in the underlying level (reflective tower);

Characteristics

- the system running in the base-level is unaware of the existence and of the work of the systems running in the overlying levels;
- a meta-level system acts on a representation (called the system running in the underlying levels; and
- a system and its reification are causally connected and therefore, they are kept mutually consistent

2.3.3 Reflective system: Base- and Meta-levels

A meta-level system refies what it is implicit (e.g. mechanisms and structure) of the underlying base- or meta-level

2.3.4 How to Characterize a Reflective System

The reflective systems can be classified based on:

- what and when

What kind of reflective actions the system can carry out:

- structural and behavioral reflection;
- introspection (just to observe) and intercession (to alter)

When the meta-level entities exist:

- compile-time
- load-time; and
- run-time

2.3.5 Behavioral and structural reflection

The behavioral reflection allows the program of monitoring and manipulating its own computation, e.g.:

- to trap a method call and activating a different method instead;
- to monitor the object state;
- to create new objects, and so on

These activities can take place at run-time without a specific support

The structural reflection allows the program of inspecting and altering its own structure, e.g.:

- the code of a method can be modified or removed from the class;
- new methods and field can be added to a class, and so on;

These activities need a specific support by the execution environment (from the VM, RTE, ...) to be carried out at run-time

2.3.6 Reification

The base-level entities (referents) are reified into the metalevel, i.e., they have a representative into the meta-level

Such a representative, called reification, has to:

- support all the operations and have the same characteristics of the corresponding referent;
- be kept consistent to its referent (causal connection);

- be subjected to the manipulations of the meta-level entities to protect the base-level entities from potential inconsistency

Any change carried out on the reification has to be reflected on the corresponding referent.