LSINF 2335

Programming Paradigms: Theory, Practice and Applications

Theme: Reflection & meta-programming

Individual Report 2013–2014

Capron Romain - 2140-08-00 - romain.capron@student.uclouvain.be
Marchal Antoine - 5462-08-00 - antoine.t.marchal@student.uclouvain.be

Python

Contents

1 Chosen Language

We have chosen Python which is a programming language that we both like. We learned to use it with the course of Artificial Intelligence, we had to implement our projects with Python. This language is intuitive and seems really complete. There is also a great community behind it and this is really helpful when we need to find an answer to our questions.

Python supports multiple programming paradigms: object-oriented [1] [2], functional [2] [3], meta-programming [4] and procedural [1] [2]. It appeared in February 1991 and takes some aspects of its philosophy from ABC [5]. Its syntax is inherited in particular from ABC and C languages [6]. Python's syntax allows us to quickly create things with smaller portions of code (e.g. we don't need to write long lines such as "*public static void main (String args[])*" in Java). Python's code has been designed to be easily readable [7] and is not compiled as in Java or C, it is interpreted. At the opposite of other programming languages, the indentation is really important because the success of the code's interpretation depends on it. Both spaces and tabulations are accepted for the indentation but mixing them can sometimes result in bugs. As many tools don't distinguish tabulations and spaces, it can be hard to debug. Python uses late binding [8] and duck typing is heavily used [9]. As Python is a dynamically typed language, the type is carried by the values, not by the variables. These ones hold references to objects and these references are passed to functions [10]. Everything is an object in Python (even classes). Furthermore, classes have a class (metaclass) [11]. Python has other interesting characteristics but we wanted to focus on the ones that seemed the most important.

Here is an exemple of Fibonacci:

```
a, b = (1, 1)
while b < 10:
    print 'a={0}, b={1} and a+b={2}'.format(a, b, (a+b))
    a, b = (b, a + b)
```

We can run this program by executing the following command:

```
$ python example.py
```

And this is the output:

```
a=1, b=1 and a+b=2
a=1, b=2 and a+b=3
a=2, b=3 and a+b=5
a=3, b=5 and a+b=8
```

```
a=5, b=8 and a+b=13
```

This language can be used in various kind of application domains. Here are some examples [12]:

1. Web and Internet development:
   - Frameworks such as Django and Pyramid
   - Micro-frameworks such as Flask and Bottle
   - Advanced content management systems such as Plone
2. Scientific and numeric:
   - SciPy is a collection of packages for mathematics, science, and engineering
   - Pandas is a data analysis and modeling library
3. Education: we learned programming with Java but Python seems to be more appropriate as it has a simpler syntax for a similar behaviour.
4. Software development: even big softwares are written in Python. For example, the well-known game *Sid Meier's Civilization IV* has been nearly completely implemented in Python.

2 Reflection and meta-programming

2.1 Reflective features

• What language features for dealing with reflection and meta-programming does the chosen language provide?
• What kinds of reflection and meta-programming features does that language offer?
• What is the MOP (meta-object protocol) for that language?
• What are the limitations of the reflective features provided by this language?
• Illustrate your explanations with working code fragments.

2.1.1 Write a Quine

To write a quine that makes use of reflection, we will divide our program in two parts:

1. a string definition

2. the program core

The string contains the code of the program core and the program core will print this string twice. The first time to print the string and the second time to print the program core. The output of this program must be its source code.

This is the quine we wrote:

```
_='_=%s; print _%%‘_‘’; print _%‘_‘
```

To explain it, we modified the _ symbols by variables a, b, c and d and we printed the output. This allows to understand directly that it has the same principle as a Matryoshka doll. The main doll is the source code that can be divided in two parts. In the program core and we can find inside (by printing) a smaller doll ; the string. And there is again a fourth doll in this string. The output of the program is a doll that has the same appearance than the original doll.

```
a='b=%s; print c%%‘d‘’; print a%‘a‘ #source code
b='b=%s; print c%%‘d‘’; print c%‘d‘ #output
```

It is reflective because:

1. The program is divided in two parts: string & core.

2. The core will print the string.

3. The string is also divided in two parts: string & core. Printed, it will produce a result that equals the source code. It is just like if the program was able to print itself.

2.2 Applications of reflection

• What are the typical applications that reflection could be used for in this language?

We need to know all details of methods associated with a class in order to enumerate these methods and print them. Several programming languages associate this structure with the type. In that case, the structure is defined by type and is already decided before run-time. Actually it is done most of the time during the compilation processus. For example, when you write a class in C++, you have to define all the details of this class. However, at run-time, the program itself is not aware of the structure of the class because

it does not have the ability the examine its inner structure while the program is running.
One could say it is useless to examine this structure at run-time because he could have done it before. When writing it or even during an eventual compilation processus. But beside that, reflection is not only about the fact of examining its own structure. It is also the ability to modify and maintain its inner structure at run-time.
A program made in a programming language that does not support reflection is not able to call twice the same method in order to produce two different results depending on the inner structure of the program. In the opposite corner, a reflective programming language can do it. And some of them, such as Python, can achieve it easily due to their simple syntax. [13]

• Can you give a working code example of such a typical problem that requires a reflective solution?

This is the Reflect.py file that will define the class *reflect* :

```
'''Demonstration class for Python reflection'''
class reflect(object):
    '''Documentation of this reflect class'''
    def __init__(self):
        '''Constructor'''
        self.firstname = "Romain"
        self.surname = "Capron"

    def message(self):
        print   "My firstname is", self.firstname,"and my
    surname is", self.surname

def welcome():
    print "Welcome my friend!"
```

This is the main.py file that will be a reflection demonstration on the class *reflect* :

```
from Reflect import reflect
from Reflect import welcome
if __name__ == '__main__':
    i = reflect()

    print i.__class__                          #1
    print i.__dict__                           #2
```

```
8     print i.__doc__                                #3
9     print i.__sizeof__()                           #4
10
11    print i.__getattribute__('firstname')          #5
12    i.__setattr__('firstname', 'Antoine')
13    i.__setattr__('surname', 'Marchal')
14    print i.firstname                              #6
15
16    welcome()                                      #7
17    i.hello = welcome;
18    i.hello()                                      #8
19
20    i.message()                                    #9
21    i.message = welcome
22    i.message()                                    #10
```

This is the output printed by main.py :

```
1  <class 'Reflect.reflect'>
2  {'surname': 'Capron', 'firstname': 'Romain'}
3  Documentation of this reflect class
4  32
5  Romain
6  Antoine
7  Welcome my friend!
8  Welcome my friend!
9  My firstname is Antoine and my surname is Marchal
10 Welcome my friend!
```

Note that each line of this last file corresponds to the red comment number (#X) in main.py.

The four first prints are applied on an instance i of the class reflect from the file Reflect.py.
In the first print (#1), we use .___class___ that refers to the class to which a class instance belongs. In this case the program asks to the instance i to print its own class.
In the second print (#2), .___dict___ is a descriptor object that returns the internal dictionary of attributes for a specific instance. In this case the program asks to the instance i to print its internal dictionary of attributes.
In the third print (#3), .___doc___ is the string of the class documentation. The class documentation of the file Reflect.py is the red comment line between the "' symbols. This print straightforwardly asks i to print a part of

the file from the class which it is an instance.
In the fourth print (#4), .___sizeof___ returns the size of an object in bytes. The instance i prints its own size in bytes.

The fifth and sixth prints use accessors and mutators on the internal dictionary of attributes d of the instance i. In the fifth print (#5), we use .___getattribute___('firstname') that will get the attribute value of the attribute called "firstname" in d. Thus, it prints Romain. Before the sixth print (#6), we modify the attribute values of the attributes "firstname" and "surname" to Antoine and Marchal respectively. If we now print the firstname of the instance i (#6), Antoine will be printed instead of Romain.

With #7, we call the method *welcome()* that simply prints the string "Welcome my friend!". Then, we link an attribute called *hello* of the instance i to that method *welcome()*. Python use dynamic typing. This attribute will be defined as a method because the behaviour of *welcome()* is the one of a method. This implies that calling *i.hello()* (#8) will actually call *welcome()* and thus just as in #7 print the string "Welcome my friend!".

With #9, we call the method message() that has been created at the creation of the instance i. The behaviour of this method on an instance j is to print the string *("My firstname is" + firstname + "and my surname is" + surname)* where the variables firstname and surname are $j.\_\_\_getattribute\_\_\_('firstname')$ and j.___getattribute___('surname') respectively. Then, similarly to what we did before #8 we linked this attribute to the method welcome. This implies that calling *i.message()* again (#10) will actually call *welcome()* and thus just as in #8 print the string "Welcome my friend!".

These four previous prints showed that it is possible and easy in Python to define are redefine dynamically typed attributes of an instance. NEED TO ADD WHY IT IS REFLEXION.

• Does there exist a “killer-app” for this language that has been implemented with reflection?

2.3 Comparison with other languages

• How does this language compare to Smalltalk, Java or Ruby from the point of view of the reflective features it supports, the kinds of reflection it offers, or its MOP?
• What can this language learn from those languages?

• Does it offer some specific reflective features that you do not have in either Smalltalk, Ruby or Java? (Can Smalltalk/Java learn something from reflection in this language?)
• Does it offer some powerful native (non-reflective) features that allow you to express things for which you would need reflection in other languages (like Smalltalk, Ruby or Java)?

3 Conclusion

In conclusion, how good does this language score as a reflective language?
o Does it provide a very rich, well-structured and well-supported set of reflective features that are supported by the programming environment as well?
o Are there only a few ad-hoc reflective features that are not well supported by the environment?
o What can other (reflective) languages learn from this language?
o What can this language learn from how reflection is dealt with in other languages?

References

[1] Alex Martelli. *Python in a Nutshell*. O'Reilly Media, Inc., 2006.

[2] A. M. Kuchling. Functional Programming HOWTO - Python v2.7.6 documentation. https://docs.python.org/2.7/howto/functional.html.

[3] Python. http://www.devtome.com/doku.php?id=python.

[4] Python and Meta-Programming | mihai.ibanescu.net. http://mihai.ibanescu.net/python-and-meta-programming).

[5] Why was Python created in the first place? https://docs.python.org/2/faq/general.html#why-was-python-created-in-the-first-place.

[6] Wikipedia – History of Python. http://en.wikipedia.org/wiki/History_of_Python.

[7] PEP 20 – The Zen of Python. http://legacy.python.org/dev/peps/pep-0020/.

[8] PEP 289 – Generator Expressions. http://legacy.python.org/dev/peps/pep-0289/.

[9] Wikipedia – Duck typing. http://en.wikipedia.org/wiki/Duck_typing#In_Python.

[10] Wikipedia – Python syntax and semantics. http://en.wikipedia.org/wiki/Python_syntax_and_semantics.

[11] 3. Data model – Python v2.7.6 documentation. https://docs.python.org/2/reference/datamodel.html.

[12] Applications for Python | Python.org. https://www.python.org/about/apps/.

[13] Reflection in Python | Assembleforce.com. http://www.assembleforce.com/2012-08/reflection-in-python.h.