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 quite easy to learn and used in several courses to teach the basics of programming (one of us had a introduction to programming during the secondary school with Python). Within our university cursus, we learned it while following the course of Artificial Intelligence because we had to implement our projects in Python. Thousands of packages (they are around 43000) are available in PyPI (the Python Package Index) so you can easily extend your projects with third party modules. Python is very portable and there is also a great community behind it. Therefore, this is really helpful when we need to find an answer to our questions. Another reason is that Google uses Python, their motto is by the way *"Python where we can, C++ where we must"*.

Python is a multi-paradigms language. It supports the following programming paradigms: object-oriented, procedural (and by extension, imperative), functional programming and last but not least, reflective (and by extension, meta-programming) [1, 2].

It appeared in February 1991 and takes some aspects of its philosophy from ABC [3]. Its syntax is inherited in particular from ABC and C languages [4]. 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 such as stated in *"The Zen of Python"* [5] and is interpreted. As in Java or C, the code can also be compiled.

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.

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 [6].

Python uses late binding [7] and duck typing is heavily used [8].

Everything is an object in Python, even classes. Furthermore, classes have a class (metaclass) [9].

Python has other interesting characteristics but we wanted to focus on the ones that seemed the most important to us.

Here is an exemple of working code in Python. It is the implementation of the Fibonacci sequence:

```
1 a, b = (1, 1)
2 while b < 10:
```

```
3     print 'a={0}, b={1} and a+b={2}'.format(a, b, (a+b))
4     a, b = (b, a + b)
```

We can run this program by executing the following command:

```
1 $ python fibonacci.py
```

And this is the output:

```
1 a=1, b=1 and a+b=2
2 a=1, b=2 and a+b=3
3 a=2, b=3 and a+b=5
4 a=3, b=5 and a+b=8
5 a=5, b=8 and a+b=13
```

This language can be used in various kind of application domains. Here are some examples [10]:

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 behavior.
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

We can easily use reflection in Python with built-in functions. Here are the main functions [11]:

- dir([object]): Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

- type(object): With one argument, return the type of an object. The return value is a type object.
- id(object): Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime.
- isinstance(object, classinfo): Return true if the object argument is an instance of the classinfo argument, or of a (direct, indirect or virtual) subclass thereof.
- callable(object): Return True if the object argument appears callable, False if not.
- getattr(object, name[, default]): Return the value of the named attribute of object. name must be a string.

It is also possible to use reflection with modules:

- The *inspect* module provides several useful functions to help get information about live objects [12].
- The *sys* module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter [13].

Here are some examples of such functions on the integer 42:

```
>>> type(42)
<type 'int'>
>>> isinstance(42, str)
False
>>> dir(42)
['__abs__', '__add__', '__and__', '__class__', '__cmp__',
    ..., '__truediv__', '__trunc__', '__xor__', 'bit_length',
    'conjugate', 'denominator', 'imag', 'numerator', 'real']
```

We can also create metaclasses: in Python, a metaclass can be a class, function or any object that supports calling an interface [14]. The metaclass by default is the "type" object.

The syntax to use metaclasses has been modified from Python 2 to Python 3 [15]. The syntax of the metaclass' creation is still the same [9]:

```
class MyMetaclass(type):
    """A metaclass"""
    def __new__(mcs, name, bases, dict):
        print("Class {} is created".format(name))
        return type.__new__(mcs, name, bases, dict)
```

Here is how to use a metaclass in Python 2:

```
class MyClass(object):
    __metaclass__ = MyMetaclass
    pass
```

If you run this code under Python 3, the hook is not recognized: Python will just add a ___metaclass___ attribute to the class but it will do nothing [16].

Here is how to use a metaclass in Python 3:

```
class MyClass(metaclass=MyMetaclass):
    pass
```

But if you want to have something which is compatible with both Python 2 and Python 3, you can use this:

```
MyClass = MyMetaclass('MyClass', (object, ), {})
```

We saw in the course that there are 2 parts of reflection: introspection which only looks at entities and intercession which can change the program behavior by manipulating the entities. Python performs the two.

Here is an example of intercession:

```
>>> class A:
...     def hello(self):
...         print "hey!"
...
>>> A().hello()
hey!
>>> def hello2(self):
...     print "Hello Sir."
...
>>> A().hello()
hey!
>>> A.hello = hello2
>>> A().hello()
Hello Sir.
```

We can also classify reflection according to what can be reflected: structural reflection and computational (behavioral) reflection. Python does both. It has a feature to add fields and methods to objects and classes. It can also modify the behavior of a method [17].

In Python, it is possible to change the class of an object at runtime [18]. Python MOP is about callables such as functions and instances with ___call___ method [19].

• What are the limitations of the reflective features provided by this language?

So far we do not have found any limitation in reflection in Python.

2.2 Applications of reflection

Python is a perfect language to easily build various type of applications. Reflection seems to be omnipresent in this language. It is completely integrated in the way of programming in Python.

Most of the time, 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 it could have be done 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. [20]

This is Reflect.py defining the class *reflect* that we will use to demonstrate reflective abilities of Python:

```
1  '''Demonstration class for Python reflection'''
2  class reflect(object):
3      '''Documentation of this reflect class'''
4      def __init__(self):
5          '''Constructor'''
6          self.firstname = "Romain"
7          self.surname = "Capron"
8
9      def message(self):
10         print   "My firstname is", self.firstname,"and my
       surname is", self.surname
11
12 def welcome():
13     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
    print i.__doc__                            #3
    print i.__sizeof__()                       #4

    print i.__getattribute__('firstname')      #5
    i.__setattr__('firstname', 'Antoine')
    i.__setattr__('surname', 'Marchal')
    print i.firstname                          #6

    welcome()                                  #7
    i.hello = welcome;
    i.hello()                                  #8

    i.message()                                #9
    i.message = welcome
    i.message()                                #10
```

This is the output printed by main.py:

```
<class 'Reflect.reflect'>
{'surname': 'Capron', 'firstname': 'Romain'}
Documentation of this reflect class
32
Romain
Antoine
Welcome my friend!
Welcome my friend!
My firstname is Antoine and my surname is Marchal
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. As explained in 2.1 this is called intercession and thus is a simple and clear example of reflection in Python.

2.2.1 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.

2.2.2 Python's killer-app : YouTube

The creator of Python, Guido van Rossum works for Google. Python is used by Google for its search engine, YouTube and other smaller Google products.

In 2006, Guido van Rossum said via the python developers mail list: *"And I just found out (after everyone else probably :-) that YouTube is almost entirely written in Python. (And now I can rub shoulders with the developers since they're all Googlers now... :-)"*

We do not have the code of these different Google products BUT if we look for this on the internet (using Google search engine of course), and if we draw aside video on YouTube about snakes, we can find the Python API in the Google developers website. For example, we can see that they use metaclasses, they even have modules called reflection (for example in the package protobuf). When loading a new message, the GeneratedProtocolMessageType metaclass uses intercession on the specified descriptors in order to create all the Python methods developers need to work with each message type and adds them to the relevant classes. [21, 22]

This is an example from the Google tutorial showing how to use reflection:

```
class Person(message.Message):
    __metaclass__ = reflection.GeneratedProtocolMessageType

class PhoneNumber(message.Message):
    __metaclass__ = reflection.GeneratedProtocolMessageType
    DESCRIPTOR = _PERSON_PHONENUMBER
    DESCRIPTOR = _PERSON

class AddressBook(message.Message):
    __metaclass__ = reflection.GeneratedProtocolMessageType
    DESCRIPTOR = _ADDRESSBOOK
```

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?
It seems that both languages allows a lot of possibilites in meta-programming but it is less common in Python than in Ruby. It is possible to realize the same things in Python or in Ruby.

• 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)?

SOIT ON FAIT QUESTION PAR QUESTION, SOIT ON FAIT PAR LANGAGE.
JE FAIS DES COPIE-COLLE DE PARTIES IMPORTANTES.

2.3.1 Python versus SmallTalk

Can change the class of an object at runtime. Related to *become* of Smalltalk

2.3.2 Python versus Ruby

0
"Both Python and Ruby have strong support for metaprogramming. Many of Rails' most-touted features use metaprogramming. In a general sense all of these things can be achieved in Python, though the techniques are usually

quite different."
1
"You can manipulate Python classes on run-time, but not basic classes such as int or basestring. While in Ruby, you can manipulate everything, including replacing/adding methods inside Integer or String."
2
"Python's reflection is quite good. The list of all living objects, however, does not exist."

2.3.3 Python versus Java

3 Conclusion

As explained before, Python is a perfect language to easily build various type of applications. Reflection seems to be omnipresent in this language. It is completely integrated in the way of programming in Python. Using reflection is done naturally and most of the people do it without even knowing the concept of reflection. This is for us the proof that reflection is successfully and fully integrated in Python.
ADD CONCLUSION ABOUT VERSUS WITH OTHER LANGUAGES
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] Why was Python created in the first place? https://docs.python.org/2/faq/general.html#why-was-python-created-in-the-first-place.

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

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

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

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

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

[9] Python documentation: 3. Data model. https://docs.python.org/2/reference/datamodel.html.

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

[11] Python documentation: built-in functions. https://docs.python.org/2/library/functions.html.

[12] Python documentation: the inspect module. https://docs.python.org/2/library/inspect.html.

[13] Python documentation: the sys module. https://docs.python.org/2/library/sys.html.

[14] How Metaclasses work technically in Python 2 and 3. http://www.pythoncentral.io/how-metaclasses-work-technically-in-python-2-and-3/.

[15] Python 2 and 3: Metaclasses. http://mikewatkins.ca/2008/11/29/python-2-and-3-metaclasses/.

[16] Metaclasses in Python 3.0 [1 of 2]. http://www.artima.com/weblogs/viewpost.jsp?thread=236234.

[17] Sheila Mendez, Francisco Ortin, Miguel Garcia, and Vicente Garcıa-Dıaz. Computational reflection in order to support context-awareness in a robotics framework. 2011.

[18] Programming Languages. http://www.cs.jhu.edu/~scott/pl/lectures/dynamic-lang-study.shtml.

[19] Presentation: what we need to know about Python. http://www.russel.org.uk/Presentations/skillsMatter_Python_inTheBrain_2011-08-02.pdf.

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

[21] Google references for developers: reflection. https://developers.google.com/protocol-buffers/docs/reference/python/index?hl=i.

[22] Google tutorials for developers: protocol buffers with reflection. https://developers.google.com/protocol-buffers/docs/pythontutorial.