LSINF2335 Programming paradigms

Assignment

Comparing Python and JavaScript : Reflection and meta-programming

Group 1

Gerondal Thibault
Heraly Michaël

Academic year 2014 – 2015

Contents

1 Chosen Languages

1.1 What programming languages have you chosen to compare ?

We chose to compare Python & JavaScript, both support reflection and meta-programming. We focused on these programming languages because of their popularity.

These two languages don't have the same purpose, so it could seem strange to compare them. On the one hand, JavaScript was born in the 90s out of a need to make web pages dynamic . On the other hand, Python was conceived in the late 1980s as a programming language capable of exception handling.

Both languages have evolved in very different contexts. Python has always been maturely considered and debated during its development cycles; while JavaScript has long been considered as a language whose sole purpose was to boost internet pages. Therefore, it was much less standardized and the standard was not always respected. Microsoft and Netscape maintained separate languages for their browser. Given the competition between the two browsers, the JavaScript language largely developed in the context of war of performance and functionality.

Nowadays, JavaScript is widely used on every website and is even used for running website servers (with popular frameworks like Nodejs, Meteorjs, ...). Python has also become one of the most used languages for multiple purposes : from scripting up to making complete applications. Both of them are still growing, evolving and becoming increasingly used.

The large success of these languages is partly due to the various libraries simplifying programmers' life. Some libraries greatly benefit from meta-programming that allows them to easily extend the behavior of a language in order to make their use much easier and more natural.

It makes sense to compare Python and JavaScript, because they share the same programming paradigms : functional, object-oriented, and imperative. However, these languages both have their own specifics.

In the following sections, we are going to give you a brief introduction to the core syntax, semantics and concepts of those languages.

1.2 Python

Python is an imperative language where everything is object. There are a lot of built-in types, here is a non-exhaustive list of the most used ones: int, bool, str, float, long, tuple, list, dict, set. Since everything is object, there is a class for all of these types which is called type. There is also the special type (object) None which belongs to NoneType (which belongs to type). None is frequently used to represent the absence of a value, as when default arguments are not passed to a function. There are also two widely used built-in constants who are True

and False, that belong to the type bool. Assignments to None, True and False are illegal and raise a SyntaxError.

For example,

```
>>> type(3) #=> <class 'int'>
>>> type(True) #=> <class 'bool'>
>>> type('Hello world !') #=> <class 'str'>
>>> type([1,2,3]) #=> <class 'list'>
>>> type({'key':'value'}) #=> <class 'dict'>
>>> type((1,2,3)) #=> <class 'tuple'>
>>> type(tuple) #=> <class 'type'>
>>> type(type((1,2,3))) #=> <class 'type'>
```

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the physical line end are part of the comment and the Python interpreter ignores them. We use them here to show the expected result.

As you can see, 'list' and 'tuple' can store multiple values. The main difference between these two structures is that the tuple is immutable. This means that the tuple cannot be modified after it is created.

The syntax in Python has no braces to indicate blocks of code for class, function definitions or flow control. Blocks of code are denoted by line indentation. The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. Some developers use tab as indentation but it is a bad practice according to PEP8 (Style Guide for Python Code).

The following code shows an example of how blocks of code are defined in Python :

```
array = [1, 2, 3, 4, 5]
for number in array :
    print('Number {n} '.format(n=number), end='')
    if number % 2 == 0 :
        print('is an even number.')
    else :
        print('is an odd number.')
```

Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. Statements contained within the [ ], { } or ( ) brackets do not need to use the line continuation character.

For example, the following code is correct :

```
a = 5
c = a + \
    6
array = [1, 2, 3,
         4, 5]
```

Sometimes a statement is required syntactically, but no code needs to be executed. For this special case, the keyword 'pass' exists. It's a null operation, when it is executed, nothing happens.

For declaring a function, we use the keyword 'def':

```
def plusOne(x):
    return x+1
```

Then, we can call the function that way : plusOne(1)

Since Python is a functional language, we can manipulate function as object, for example :

```
>>> type(plusOne) #=> <class 'function'>
>>> a = plusOne
>>> a(1) #=> 2
```

For declaring conditions, we use keywords if, elif and else :

```
if condition1 :
    indentedStatementBlockForTrueCondition1
elif condition2 :
    indentedStatementBlockForFirstTrueCondition2
elif condition3 :
    indentedStatementBlockForFirstTrueCondition3
elif condition4 :
    indentedStatementBlockForFirstTrueCondition4
else:
    indentedStatementBlockForEachConditionFalse
```

For declaring loop, we use the keywords while or for :

```
while condition :
    <statement>
# or
for <var> in <list> : # e.g. for i in range(0, 3)
    <statement> # e.g. print(i)
```

Python is an Object-Oriented Programming language. Here is an example of how to create a class :

```
class Marionette:  # define Marionette class
    def dance(self): # define a method in the class
        print('I am dancing.')
```

Python supports multiple inheritance, for example :

```
class Wood:
    def burn(self):
        print('Wood object does burn well.')

class Pinocchio(Marionette, Wood): # inheritance from Marionette
    and Wood
    pass
```

Pinocchio will inherit the methods dance and burn.

Python also has some borrowed concepts from the functional world such as comprehension lists and lambda functions (also called anonymous functions) :

```
>>> [x for x in [1,2,3,4,5] if x % 2 == 1] #comprehension list
# => [1, 3, 5]
>>> list(filter(lambda x: x % 2 == 1, [1,2,3,4,5])) #lambda
    function
# => [1, 3, 5]
```

These two pieces of code have the same result. It will return an array with element which are odd from the list [1,2,3,4,5] . One of the most famous mottos in Python is "There should be one– and preferably only one –obvious way to do it." As we can see, we are able to do the same thing using two ways. In this case, the first piece of code is definitively better as it is easier to read.

1.3 JavaScript

JavaScript is a dynamically typed language. JavaScript objects consists of associative arrays. The names (keys) of object properties are strings. The properties and their values can be added, changed, or deleted at run-time. There are seven JavaScript primitive Data types : Object, Boolean, String, Number, Symbol (not yet standard, in ECMAScript 6 proposal).

Among which two special ones :

Undefined variables without a value (or setting a variable to undefined).

Null a variable whose value is **null** represents "nothing". To be consistent, the type of such a value should be Null. But the typeof a **null** value is Object. This is considered as a bug in JavaScript, but it is part of the ECMAScript language specification.

There are also built-in types (which are not primitive data types) : **Array**, **Function**, **RegExp**.

To declare a variable, the var keyword must be used. If the variable is not declared (using var) before being assigned, its scope is global.

```
function f() {
  var x = 42; // only accessible within the scope of the function
  y = 90; // global
}
```

In JavaScript, the declared variables are created before any code is executed. Undeclared variables do not exist until the code assigning to them is executed. It is recommended to always declare variables, regardless of whether they are in a function or in global scope.

Creating an object :

```
var myobj = {
  key1: value1,
  key2: value2,
```

```
  ...
};
```

Creating an array :

```
var myarray = [1, 2, 3, "element", {name: "Hello CoffeeScript!"}];
```

Defining a function : Functions delimit a block of statements. There are named functions, but also anonymous functions.

Named function	Anonymous function

`function name([param, [...]]) {`
`  [statements]`
`}`
	`var name =`
	`  function([param, [...]]) {`
	`  [statements]`
	`  }`

Define conditions :

```
if (condition1)
   statement1
else if (condition2)
   statement2
else if (condition3)
   statement3
...
else
   statementN
```

Define loops :

```
for ([initialization]; [condition]; [final-expression]) {
    [statements]
}
// or
while (condition) {
    [statements]
}
```

The **this** keyword has a different behaviour than in most other programming languages. Its value is determined by how a function is called.

If the enclosing function is an object method, **this** corresponds to the object the method is called on. This is also true for methods defined on the object's prototype chain. When a function is used as a constructor (with the **new** keyword), **this** is bound to new objects being constructed.

Otherwise, if it is used in the global execution context (outside of any function), **this** refers to the global object.

The bind method can be used to set the value of a function's this regardless of how it's called (c.f. the reflection part for explanations).

Custom object types can be defined using the new keyword.

```
function Language (name) {
    this.name = name;
}
var coffee = new Language("CoffeeScript");
```

The object returned by the constructor function (Language in this case) becomes the result of the whole **new** expression.

New object properties can be defined dynamically :

```
coffee.specificity = "Exposes the good parts of JS";
coffee.syntax = function () {
    return "The syntax of "+this.name+" is beautiful!";
};
```

As of ECMAScript 5.1, there is no **class**. The ECMAScript 6 proposal introduces a **class** expression.

```
var Foo = class {
  constructor() {}
  bar() {
   return "Hello World!";
  }
};

var instance = new Foo();
instance.bar(); // "Hello World!"
```

1.4 Give an illustrative working code example of a typical program written in each of those languages

We choosed to create a filter method from scratch to filter only odd numbers of a list.

In python,

```
def filter(tab, cond):
    results = []
    for el in tab :
        if cond(el):
            results.append(el)
    return results

tab = [1,2,3,4,5]
cond = lambda x :  x % 2 == 0
print(filter(tab, cond)) # => [2,4]
```

In JavaScript,

```
var filter = function (tab, cond) {
  results = [];
  for (var i = 0; i < tab.length; i++) {
    if (cond(tab[i]))
      results.push(tab[i]);
  }
```

```
  return results;
}

var tab = [1, 2, 3, 4, 5];
var cond = function (num) {
  return num % 2 == 0;
}
console.log(filter(tab, cond)); // [2, 4]
```

2 Reflection and meta-programming

2.1 Python

2.1.1 Languages features for dealing with reflection and meta-programming

Python is a duck-typed language, it does not have a strong typing. The idea is that you do not need a type in order to invoke an existing method on an object. If a method is defined on it, you can invoke it.

For example :

```
class Duck:
    def quack(self):
        print('Quack, quack!')

class Person:
    def quack(self):
        print('I\'m Quackin\'!')

def in_the_forest(mallard):
    mallard.quack()


a = Duck()
b = Person()
in_the_forest(a) # => Quack, quack!
in_the_forest(b) # => I'm Quackin'!
```

Duck typing is useful for defining new class that can take place in an existing system without having to implement or inherit useless methods from another class. For example, in Python, if you have a library that asks a "File" object as parameter, you can pass a string via the "cStringIO" class. This class will mimic the behavior of the File class. The name of the concept refers to the duck test, attributed to James Whitcomb Riley, which may be phrased as follows: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."

Although Python is duck -typed, it is possible to check the type of a class via the built-in function 'type'. We can also check if a class has an attribute via the built-in function 'hasattr'. For example :

```
>>> type(a) # => <class '__main__.Duck'>
>>> hasattr(a, 'quack') # => True
```

These methods exist and are useful but the philosophy of Python and duck-typing is to avoid using them and just call the method. If there is a risk of failure, Python gives us an appropriate exception to handle this situation : AttributeError and TypeError. AttributeError is triggered when the object does not have the supposed method (e.g. in_the_forest(5)). TypeError is raised when an operation or function is applied to an object of inappropriate type (the associated value is a string giving details about the type of mismatch) (e.g. 5 + 'a').

As said before, in Python, everything is object and all objects are "first class". By “first class”, we mean that all objects that can be named in the language (e.g., integers, strings, functions, classes, modules, methods, etc.) have equal status. This definition was designed by Guido Van Rossum, creator of Python. This means that every object can be assigned to variables, placed in lists, stored in dictionaries, passed as arguments, and so on.

So function and methods are also first class, as we can check :

```
>>> type(in_the_forest)  # => <class 'function'>
>>> type(Person.quack)  # => <class 'function'>
```

This principle enables us to define and redefine functions and methods simply. For example, we can do the following hack :

```
def b(self):
    print('nope nope')
Person.quack = b
```

This practice is called monkey-patching. We can use it to change a function in run-time.

Python gives us tools for introspecting class, the built-in function ‘dir’ gives the attributes (i.e. methods and class variable) of a class. For example :

```
>>> dir(Person)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
    '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
    '__hash__', '__init__', '__le__', '__lt__', '__module__',
    '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
    '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
    '__weakref__', 'quack']
```

The resulting list is not necessarily complete, and may be inaccurate. As this built-in function will rely on the method __getattr__() of the object, we can define (or change on runtime) this method.

We can inspect an object with dir, but we can also inspect a frame on the runtime. For doing so, we can use built-in function locals() and globals().

locals() returns a dictionary representing the current local symbol table.

globals() returns a dictionary representing the current global symbol table.

For example, we can call a function with globals()['function_name'](args).

2.1.2 Kinds of reflection and meta-programming features

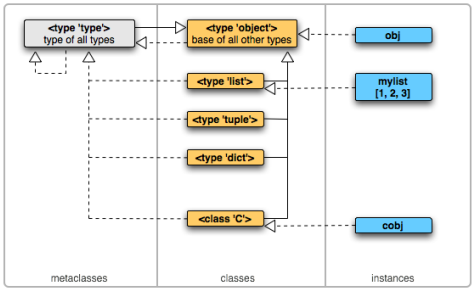
Introspection As we have seen, we can easily inspect and access representation of an object in python with the method dir(). We can also inspect locals and globals symbol tables via locals() and globals(). We have seen that we can do monkey-patching to add or replace an attribute in an object (remember that everything is object in python).

Intercession

2.1.3 Meta-object protocol

As seen before, in Python, everything is object and every object is first-class. All classes inherit from the class "type", even the class "type":

```
>>> type([1, 2, 3])
<class 'list'>
>>> type(type([1, 2, 3]))
<class 'type'>
>>> type(in_the_forest)
<type 'function'>
>>> type(type(in_the_forest))
<type 'type'>
>>> type(dir)
<type '
    builtin_function_or_method'>
>>> type(type(dir))
<type 'type'>
>>> type(type(type(type(type(dir
    )))))
<type 'type'>
```



In Python, there is no way to prevent a class to modify another. However, attribute's name can be used to prevent and alert other programmes that an attribute or method is intended to be private.

One leading underscore

A leading underscore indicate is weak "internal use" indicator. It it a little bit more than just a naming convention because when we import all attributes of a class through the wildcard "*" (e.g. `from X import *`), these "internal use" attributes will not be imported.

Two leading underscore

Two leading underscore is called "name mangling". It's a strong "internal use" indicator. Any identifier of the form `__spam` (at least two leading underscores) is textually replaced with `_classname__spam`, where classname is the current class name with leading underscores stripped. This "mangling rule" is designed mostly to avoid accidents; it still is possible for a determined soul to access or modify a variable that is considered private.

Two leading underscore and two ending underscore

When we see a method like __this__, don't call it. Because it means it's a method which python calls, not by you. There is always an operator or native function which calls these magic methods. Sometimes it's just a hook python calls in specific situations. For example __init__() is called when the object is created. __new__() is called to build the instance...

```
>>> class M:
...     def __init__(self, *args, **kwargs):
...         self._secret1 = 0
...         self.__secret2 = 0
...
>>>
>>> a = M()
>>> a._secret1
0
>>> a.__secret2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: M instance has no attribute '__secret2'
>>> a._M__secret2
0
```

Callable Object

Object in Python can be "callable" or not. A callable object is an object having a method named __call__(). In fact, X(arg1, arg2, ...) is a shorthand for X.__call__(arg1, arg2, ...). To know if an object is callable, we will use the method __getattr__('__call__').

2.1.4 Limitations of the reflective features

2.2 JavaScript

2.2.1 Languages features for dealing with reflection and meta-programming

2.2.2 Kinds of reflection and meta-programming features

2.2.3 Meta-object protocol

2.2.4 Limitations of the reflective features