LSINF2335 - Programming paradigms

Report

A Comparison of Reflection & Meta-programming in Python and JavaScript

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, as both support reflection and meta-programming. We focused on these programming languages because of their popularity.

These two languages do not have the same purpose, so it could seem strange to compare them. On 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 most websites and is even used for running web 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 specificities.

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`.

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 marks the beginning of a comment. All characters after the # and up to the 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 tuples are 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 a class, function definitions or flow control. Blocks of code are denoted by indentation. The number of spaces in the indentation is variable, but all statements within the block must be indented of 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 valid :

```
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 skip 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 loops, 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. The first type in JavaScript is `Object`. In addition, there are six JavaScript primitive Data types :

Boolean `true` and `false`.

String `"This is a JS string"`

Number `42 or 3.14159`

Symbol A data type whose instances are unique and immutable.
(not yet standard, in ECMAScript 6 proposal).

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. reflection part 2.2.1 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 chose to create a filter method to get only the 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 (taken from wikipedia) :

```
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 classes 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."

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 provides 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'`).

In Python, everything is object and all objects are "first class". It means 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.

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 piece of code changes a method at runtime. This practice is called monkey-patching.

Python provides 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 keys in `__dict__` (dict, for dictionary) and on the method `__getattr__(object, name)` of the object, we can define (or change at runtime) these attributes. `__setattr__(object, name, value)` is the counter-part.

We can inspect an object with `dir(object)`, but we can also inspect a frame at 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)`.

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 programs that an attribute or method is intended to be private.

One leading underscore

A leading underscore is a 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.

```
>>> class MyClass:
...     def __init__(self, *args, **kwargs):
...         self._secret1 = 0 # One leading underscore
...         self.__secret2 = 0 # Two leading underscore
...
>>>
>>> a = MyClass()
>>> a._secret1
0
>>> a.__secret2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: MyClass instance has no attribute '__secret2'
>>> a._MyClass__secret2
0
```

Two leading underscore and two ending underscore

When a method is like `__this__`, it shouldn't be called. Because it means it's a method that Python calls, and the developer should not call it. There is always an operator or native function that 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.

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__')`.

```
>>> class ActingLikeAFunction:
...     def __call__(self):
...         print("I look like a function")
...
>>> a = ActingLikeAFunction()
>>> a()
I look like a function
```

Also, we can create a class by using "type" : `type('ClassName', (inherited_class_1, inherited_class_2, ...) {key: value,})`. Then, we can add attributes or methods to this class.

```
>>> MyClass = type('MyClass', (object,), {'x': 1, 'y':2})
>>> MyClass.method = lambda self : self.x
>>> a = MyClass()
>>> a.method()
1
>>> MyClass.method = lambda self : self.x + self .y # monkey-
    patching
>>> a.method()
3
>>> getattr(a, 'method').__call__() # Another way to call the
    method
3
>>> globals()['__builtins__'].getattr.__call__(globals()['a'], '
    method').__call__() # Another long way..
3
```

The Metaclass Hook

With all these tools, we can do meta-programming. However, this is not always easy and/or readable. Python introduces an easy-way to create Metaclass, wich can be seen as a class factory. First, let's see how to create a meta-class with and without the facilities of Python (syntactic sugar).

```
>>> class MyMetaClass(type):
...   pass
>>> type(MyMetaClass)
<class 'type'>
>>> FactoredClass = MyMetaClass('
    FactoredClass', (object,),
    {})
>>> type(FactoredClass)
<class '__main__.MyMetaClass'>
```

Listing 1: Without syntactic sugar

```
>>> class MyMetaClass(type):
...   pass
>>> type(MyMetaClass)
<class 'type'>
>>> class FactoredClass(object,
    metaclass=MyMetaClass):
...   pass
>>> type(FactoredClass)
<class '__main__.MyMetaClass'>
```

Listing 2: With syntactic sugar

Metaclasses are not that much useful and used in practice. A good example of usage of Metaclass in Python is the creation of a Singleton :

```
class Singleton(type):
    instance = None
    def __call__(cls, *args, **kw):
        if not cls.instance:
             cls.instance = super(Singleton, cls).__call__(*args, *
    *kw)
        return cls.instance
```

Every class that will be declared with the metaclass `Singleton` will only have one object. Even if we create multiple objects, they will all be the same object behind. This is done by overriding the `call()` method. If any instantiation exists, the metaclass `Singleton` will not create a new object but will bind the existing one to the supposed new one.

```
>>> class Database(metaclass=Singleton):
...   pass
>>> a = Database()
>>> b = Database()
>>> a
```

```
<__main__.Database object at 0x10a9ae5c0>
>>> b
<__main__.Database object at 0x10a9ae5c0>
```

2.1.2 Kinds of reflection and meta-programming features

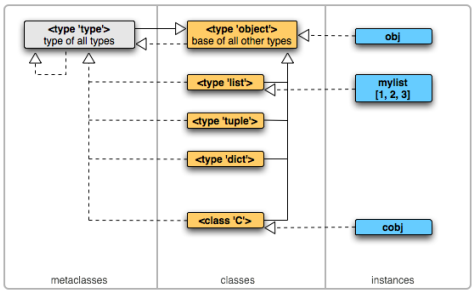
Introspection As we have seen, we can easily inspect and access representation of an object in Python with the method `dir()`. We can also inspect locals and globals symbol tables via `locals()` and `globals()`.

Intercession We have seen that we can add or replace an attribute (method or variable) in an object.

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'>
```



2.1.4 Limitations of the reflective features

Limitations of meta-programming in Python are located in everything related to the primary types and to the primary built-in functions. For example, we cannot add a method to the type "type", "object", "list", "dict", etc. We can redefine some of these primary attributes, but this might have no effect. For example, redefining "type" or "object" to "None" will have no impact on the creation of new classes or objects. This is a clear limitation of Python as we can alter the global system (and this is considered as a valid operation) but this does not have any impact.

2.2 JavaScript

2.2.1 Languages features for dealing with reflection and meta-programming

Different informations about the type of an object `custom` can be found in JavaScript :

- The primitive data type of an object can be found using the `typeof` operator (e.g. `typeof custom // "object"`)
- To access the function that created the object, the `custom.constructor` instruction can be used. (e.g. `custom.constructor // function CustomObject()`)
- The `instanceof` operator allows to check if an object is an instance of a particular constructor (or has this constructor in its prototype chain). Example:

  ```
  custom instanceof CustomObject // true
  custom instanceof Object // true
  ```

  N.B. objects created from primitive data type are considered as instances of `Object` :

  ```
  new Boolean() instanceof Object; // true
  new String() instanceof Object; // true
  new Number() instanceof Object; // true
  ```

Undefined object In the case of an `undefined` object, only the primitive data type can be retrieved (i.e. `typeof undefined // "undefined"`), as the `Undefined` constructor is not defined, and an `undefined` object has no properties.

Object properties It is possible to get the own properties of the object `s` by calling `Object.getOwnPropertyNames(s)`.
Example:

```
function Song (artist, title) {
    this.title  = title;
    this.artist = artist;
    this.display = function() {
        return artist+" - "+title;
    }
}
var s = new Song("Mark Knopfler", "What It Is");
print(Object.getOwnPropertyNames(s));
// Array ["title", "artist", "display"]
```

There is no built-in feature to get the methods of an object (as of ECMAScript 5.1). But it can be achieved by the following function :

```
var getMethodNames = function(obj) {
    var methods = [];
    for (var i in obj) {
        if (obj[i] instanceof Function) {
            methods.push(i);
        }
```

```
    }
    return methods;
}
getMethodNames(s); // Array ["display"]
```

There are two ways to get the prototype of an object :

```
Object.getPrototypeOf(s); // standard way
s.__proto__;
```

The prototype properties of a constructor can be accessed this way :

```
Song.prototype.genre = "rock";
Song.prototype; // Object { genre: "rock", constructor: Song() }
```

It can also be retrieved from an object :

```
s.constructor.prototype; // Object { genre: "rock", constructor:
    Song() }
```

So now, the object `s` can access the field `genre` :

```
s.genre; // "rock"
```

JavaScript proceeds method lookup in the following order :

1. in the object itself;
2. following the prototype chain.

JavaScript allows to define object properties reflectively :

```
Object.getOwnPropertyNames(s);
// Array ["title", "artist", "display"]
s.hasOwnProperty("album"); // false
s.album = "Sailing to Philadelphia";
Object.getOwnPropertyNames(s);
// Array ["title", "artist", "display", "album"]
s.hasOwnProperty("album"); // true
s.album; // "Sailing to Philadelphia"
```

It is also possible to set object properties :

```
s.title = "Why Worry";
s.album = "Brother In Arms";
```

The `bind` method creates a new function which, when called, has its `this` keyword set to the provided value.
Example (taken from the MDN) :

```
this.x = 9;
var module = {
    x: 81,
    getX: function() { return this.x; }
};

module.getX(); // 81

var getX = module.getX;
getX(); // 9, because in this case, "this" refers to the global
    object

// Create a new function with 'this' bound to module
var boundGetX = getX.bind(module);
boundGetX(); // 81
```

As in JavaScript the value of this is determined by how a function is called, the `bind` method is useful to set the value of a function's `this`, regardless of how it is called. It is particularly useful when making AJAX requests.

The ECMAScript 6 proposal brings the `Proxy` object. It allows to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc). But it is not yet implemented in web browsers as of this writing.
Example (taken from the MDN) :

```
var handler = {
    // Return 37 if 'name' is not in target
    get: function(target, name){
        return name in target?
            target[name] :
            37;
    }
};

var p = new Proxy({}, handler); // new Proxy(target, handler)
p.a = 1;
p.b = undefined;

console.log(p.a, p.b); // 1, undefined
console.log('c' in p, p.c); // false, 37
```

2.2.2 Kinds of reflection and meta-programming features

JavaScript allows different kinds of reflection features :

- Introspection : looking at the reified entities.
- Intercession : changing the program behaviour by manipulating the reified entities.

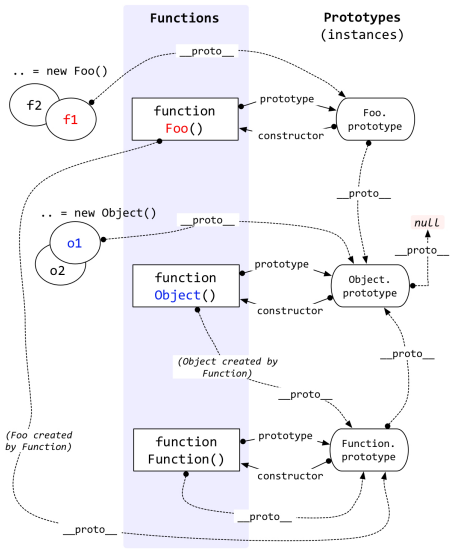
As JavaScript is an interpreted language, it proceeds runtime reflection. That means that the system can be adapted at runtime, once it has been created and run.

2.2.3 Meta-object protocol

The primitive data types in JavaScript are of type object (i.e. `Object`, `Boolean`, `String`, `Number`, and `Symbol`). Everything is object in JavaScript, even objects' prototypes. Note that Functions are also objects.

Every object has a prototype, a prototype is an object (dictionary) that binds method names to actual code. We use the property `__proto__` to access the prototype of an object.

The `prototype` property corresponds to the object (called *prototype object*) that was used to build the actual object.



When the `f1` object is created (with the `new` keyword), its *internal* `__proto__` refers to the prototype of the `Foo` function. Therefore, `Foo.prototype` *inherits* from the prototype of `Object`. The `__proto__` property can be used to provide *inheritance* of properties.

```
>>> function Custom() { console.log('hello'); }
undefined
>>> Custom.hasOwnProperty('constructor')
false
>>> Custom.prototype.hasOwnProperty('constructor')
true
```

2.2.4 Limitations of the reflective features

In JavaScript, object properties can be deleted, but not the objects themselves. Contrarily to Ruby, there is no way to access instances of a constructor in JavaScript.

3 Comparing the languages

3.1 Reflective features

Both Javascript and Python, are dynamically typed (duck-typed) languages. In both languages, everything is object. They both offer the same kind of reflective features but they don't rely on the same mechanism. Javascript is prototype-based and Python is class-based. In Javascript, you can inspect an object by calling `Object.getOwnPropertyNames(objectToAnalyse)`. This will only show the properties that the object (dictionary) contains. As many methods are defined in Prototypes in JavaScript, you will have to inspect the attribute `__proto__` of the object to retrieve other available methods of the object. But that will not be enough to find all other available methods as JavaScript rely on Prototype-chaining to find method name. So, you will have to explore the attribute `__proto__` of the `__proto__` object, and so on until there is no more `__proto__` defined. Python rely on classes, by calling `dir(objectToAnalyse)`, all attributes that the object inherits are retrieved by using the keys in the `__dict__` attribute of the object.

3.2 Kinds of reflection

Same kinds of reflection (introspection & intercession).

3.3 Meta-object Protocol

Tycale

3.4 Difference of reflection and meta-programming

JavaScript has a better API for reflection and meta-programming than Python.

4 Conclusion

5 Bibliography

Bibliography