Vrije Universiteit Brussel

Principles of Object-Oriented Programming Languages

Assignement - Languages comparison : Python and Smalltalk

Bruno Rocha Pereira - 0529512

1 Introduction

This paper establishes a comparison between the two Object-Oriented programming languages: Python and Smalltalk. The implementation of smalltalk used for this paper is Pharo. However, the examples should work in most Smalltalk implementations. The main points of comparison that will be used here are the main concepts seen in the ”Principles of Object-Oriented Programming Languages” course as well as some specific language properties.

2 Closures

2.1 Python

In python, closures consist in inner functions that will have access to everything that is defined in the enclosing scope even when the outer function is deleted. It means that if we create a binding to the inner function, we will still be able to access it eventhough the outer one is not in the namespace anymore.

11.6 shows a simple of how a closure works in python.

Python implements an easy way to wrap closures: decorators. Decorators are a sort of syntactic sugar to easily do a preprocessing and/or a postprocessing around a closure.

2.2 Smalltalk

In Smalltalk, closures are present in the form of blocks. They can be used even if the method defining them have returned. Blocks can also access the outer functions' variables. However, compared to python, closures are not names and can't be bound and called later.

Even though a block can return something and thus escaping the outer function, they cannot do that when the function defining it has already return. This would lead to a runtime condition error caught by the virtual machine.

Exactly as in python, local variables can be used inside closures.

3 Delegation

3.1 Python

Delegation in python is preatty straightforward. We can simply bind a method from one class to another method where it delegates.

```
In [1]: class Counter:
   ...:     def count(self):
   ...:         print("Hello")
   ...:

In [2]: class PCounter:
   ...:     def __init__(self):
   ...:         self.counter=
      Counter()
   ...:         self.cout = self.
      counter.count
   ...:

In [3]: pc = PCounter()

In [4]: pc.cout
Out[4]: <bound method Counter.count
     of <__main__.Counter object at
     0x7fae981c7828>>

In [5]: pc.cout()
Hello
```

3.2 ST

In Smalltalk, delegation is even easier. We can simply call a method from another method which will then delegate to the first one.

4 Inheritance

Inheritance works pretty much in the same manner in Smalltalk as in python. In both language, any class inherits a base class, with the default base class being Object. This inheritance of the class Object is explicit in python2 and implicit in python3. It has to be explicit in Smalltalk. However, some differences in the inheritance mechanism exist.

4.1 Python

Python allows a mechanism called multiple inheritance. A class can inherit multiple superclasses and get all the attributes and methods of these superclass. The resolution of the problem of overlapping methods will be detailed later in this paper. However, in the case of multiple inheritance, a call to super can be ambiguous and has to be explicit or enthrusted to the method resolution order which uses depth-first, left-to-right. The method resolution order is dynamical and the super call is more performant than the one found in single-inheritance languages.[1]

Multiple inheritance can be very useful. Superclasses are in that case used as mixins that can stand alone.

```
In [19]: class First:
   ....:     def printer(self):
   ....:         print ("First")
   ....:

In [20]: class Second:
   ....:     def printer(self):
   ....:         print ("Second")
   ....:

In [21]: class Third(First, Second)
    :
   ....:     def __init__(self):
   ....:         super(Third,self
      ).printer()
   ....:

In [22]: a = Third()
First
```

[1] https://docs.python.org/3/tutorial/classes.html

4.2 Smalltalk

Smalltalk, on the other hand uses only simple inheritance. There is no built-in mechanism for multiple inheritance and therefore no way of doing a diamond inheritance. Nonetheless, multiple solutions have been proposed over time to simulate it, overriding for example the DoesNotUnderstand method.

5 Typing

Python and Smalltalk are both strongly and dynamically typed.Strongly typed means that the types of the variables do not suddenly change without any explicit cast. Dynamically typing means types are not specified at compile time. They both chose the same typing implementation. there's thus no particular observation to make in this point. However, an implementation of Smalltalk with strong typing has been proposed under the name "StrongTalk".

6 Mixins and traits

6.1 Python

Python uses mixins. Example : the socketserver module contains TCPServer and UDPServer as well as the ForkingMixin and ThreadingMixin mixins. Using a mixin with a class consists of creating a new class inheriting the class and the mixin. We can thus write a class ThreadingUDPServer: class ThreadingUDPServer(ThreadingMixIn, UDPServer): pass that will use threading to handle each connection to the UDP server.

6.2 Smalltalk

Pharo has traits. The problem with mixins is that they are

Let's define a trait:

```
TAnnouncer>>announce:
    anAnnouncement
    | announcement |
    announcement := anAnnouncement
        asAnnouncement.
    self subscriptions ifNil: [ ^
        announcement ].
    self subscriptions
        keysAndValuesDo: [ :class :
        actions|
        (class handles: announcement
            )
            ifTrue: [ actions
                valueWithArguments: (
                Array with:
                announcement) ] ].
    ^ announcement
TAnnouncer>>on: anAnnouncementClass
     do: aValuable
    ^ self subscribe:
        anAnnouncementClass do:
        aValuable
```

Once this trait has been created, we can apply it to a class this way :

```
Object subclass: #BulletinBoard
    uses: TAnnouncer
    instanceVariableNames: '
        subscriptions'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BulletinBoard'
```

However, methods from the trait can be excluded by using

```
uses: TAnnouncer - {#
    methodIDoNotWant}
```

Methods from the traits might however overlap with the class methods, with the superclass or with each others. There are several solutions to these situations. The first one is to exclude the trait methods. The second one is to redefine locally one of the overlapping methods. In the case of the superclass, the method from the trait simply overrides the superclass' one.

The fact that Smalltalk uses traits instead of mixins allows it to handle conflicts happening when two traits applied have common name methods while mixins do not allow that. Python does [2]

7 Reflection

7.1 Python

In python, we can simply inspect data at runtime. Several tools can be used to achieve that. The first tool that I'm going to detail is the keyword module.

```
In [1]: import keyword
In [2]:  keyword.kwlist
```

[2] http://pharo.gemtalksystems.com/book/LanguageAndLibraries/Traits/

```
Out[2]: ['and', 'assert', 'break',
    'class', 'continue', 'def', '
    del', 'elif', 'else', 'except',
     'exec', 'finally', 'for', '
    from', 'global', 'if', 'import'
    , 'in', 'is', 'lambda', 'not',
    'or', 'pass', 'print', 'raise',
     'return', 'try', 'while', '
    yield']
```

As we can see, the keyword module allows us to get the list of the keywords defined in the language. This list can also be modified on the fly (with append() for example).

Another tool available for inspection is the dir() function. It displays all the available attributes of the object given as parameter.

Modules imported can be overwritten at runtime. 11.4

However, in python some built-in functions (like print, datetime,..) cannot be changed (they can still be overwritten) since they are written in C. The list of built-in functions can be found using dir(__builtins__) 11.5 shows an example of trying to change a built-in function.

7.2 Smalltalk

Smalltalk is a reflective programming language. That means programs can inspect their own execution and structure. The metaclasses can be reified to objects and then queried or inspected. These reified metaobjects can also be modified and these changes will be reported back to the execution system.

In addition Pharo/Smalltalk includes the Inspector. The inspector allows us to have a look at an object, change the instance variables and send methods to it.

8 Concurrency

8.1 Python

In the concurrency domain, python offers two ways of doing it: using Processes or using Threads. They both have a lot in common since under the hood, they both use the same syscall (clone()) [3]. However, they are still different from each other. Threads and Processes do not share the same memory. Subrocesses spawned by a same process do not share the same memory. Meanwhile, Threads do not only share memory, they also share descriptors, the filesystem context, and signal handling. Moreover, Threads use a lot less memory then Processes since they do not have to reacreact a whole context when spawned. Threads look then a lot more appealing than Processes. It is actually a lot more risky as well and necessitates to use more concepts. Indeed, since Threads share memory, situations like race conditions and deadlocks can happen. Therefore, Threads and Processes both have pros and cons.

11.2 shows how to simply use threads without shared ressources. If we want to lock a part of the code (in order to display words instead of letters for example), we can instantiate an RLock() and put the code we want to lock in a:

```
with lock:
```

The other solution is to use Processes. 11.3 shows an example of using processes and workers. This example simply prints "worker" each time a process is spawned.

[3] https://migrateup.com/python-concurrency-story-pt1/

Since there is no shared memory, we don't need to use semaphores.

8.2 Smalltalk

In order to execute a program concurrently in SmallTalk, a block has to be created, and the method fork has to be called on it. The folling code will spawn a thread that will write the numbers 1 to 10 in the transcript.

```
[ 1 to: 10 do: [ :i | Transcript
    show: i printString ; cr ] ]
    fork
```

To create fake concurrent processes, we can use as shown in 11.8

Another way of creating concurrency is to use the newProcess method. This will create a new process, in the "suspended" state. This means the block will not be sent to the process scheduler. In order to execute the block, the method "resume" has to be called on the process. Different specific methods can be called on processes : "suspend" ,"resume","terminate".

There is of course no point in spawning a single thread since there is no concurrency. In otder to schedule the different threads/processes, SmallTalk uses the ProcessorScheduler. The scheduler is based on a system of priority - an integer from 1 to 100 - associated with the processes.

In Smalltalk, synchronized data can be defined using an instance of the class "Semaphore".

Smalltalk only implement one type of processes - instances of the class Process -, which can be compared to python's threads. They don't have their own memory like real OS processes have.[4]

[4] http://pillarhub.pharocloud.com/hub/Uko/concurrentProgrammingInPharo

9 Particularities

9.1 Python

Classes methods in python always have to have "self" as first parameter (It can actually be any word we want, but using 'self' is a convention to make it as explicit as possible). For example, when we use this simple code :

```
In [6]: class AClass:
    def __init__(self):
        self.attribute = 'value'
   ...:     def a_method(self):
   ...:         print(self.attribute
     )
In [7]: instance = AClass()
```

Doing this :

```
In [8]: instance.a_method()
value
```

actually is a shortcut for doing this:

```
In [10]: AClass.a_method(instance)
value
```

Using self (or the actual object) has in fact multiple advantages. One of them is the case of multiple inheritance. For example, if we define two classes with the same methods and a third class inheriting the first two classes, using self as parameter allows us to differentiate between the two methods with the same name. Example: 11.1

A built-in mechanism like this does not exist in Smalltalk and wouldn't be useful since multiple inheritance does not exist either.

10 Conclusion

Despite the fact that these two languages are both object-oriented and that in both of them everything is an object, they are very different from each other. Python seems to add more functionalities and be more complete while Pharo(Smalltalk) gets straight to the point. This is also probably due to the fact that Pharo is still under heavy development.

11 Code Listing

11.1 Using self as parameter in Python

```
In [11]: class Teacher:
   ....:     def work(self):
   ....:         print("I’m␣teaching")
   ....:

In [12]: class Student:
   ....:     def work(self):
   ....:         print("I’m␣studying")
   ....:

In [13]: class Assistant(Teacher,Student):
   ....:     def work(self):
   ....:         Teacher.work(self)
   ....:         print("␣and␣")
   ....:         Student.work(self)
   ....:

In [14]: assistant = Assistant()

In [15]: assistant.work()
I’m␣teaching
␣and
I’m studying
```

11.2 Using Threads in Python

```
class Displayer(Thread):


    def __init__(self, lettre):
        Thread.__init__(self)
        self.letter = letter

    def run(self):
        i = 0
        while i < 20:
            sys.stdout.write(self.letter)
            sys.stdout.flush()
            waitTime = 0.2
            waitTime += random.randint(1, 60) / 100
            time.sleep(waitTime)
            i += 1

thread_1 = Displayer("1")
thread_2 = Displayer("2")

thread_1.start()
thread_2.start()

thread_1.join()
thread_2.join()
```

11.3 Using Processes in Python

```
import multiprocessing

def worker():
    print ("Worker")
    return

if __name__ == '__main__':
    jobs = []
    for i in range(5):
        p = multiprocessing.Process(target=worker)
        jobs.append(p)
        p.start()
```

11.4 Overwriting module parts in Python

```
>>> import os
>>> os.path = 42
>>> os.path
42
>>> os.path.join()
Traceback (most recent call last):
  File '"<stdin>",␣line␣1,␣in␣<module>''
AttributeError:␣'int'␣object␣has␣no␣attribute␣'join'
```

11.5 Trying to modify a built-in function in Python

```
>>> print.__str__()
'<built-in␣function␣print>'
>>> print.__str__ = 42
Traceback (most recent call last):
  File '"<stdin>",␣line␣1,␣in␣<module>''
AttributeError:␣'builtin_function_or_method'␣object␣attribute␣'__str__'␣is␣
    read-only
```

11.6 Closure example in Python

```
In [6]: def gen_mult(n):
   ...:         print ("Gen")
   ...:         a = 3
   ...:         def n_times(x):
   ...:                 return x*n*a
   ...:         print ("Return")
   ...:         return n_times
   ...:

In [7]: mult_by_4 = gen_mult(4)
Gen
Return

In [8]: del gen_mult

In [9]: mult_by_12(8)
Out[9]: 96

In [10]: gen_mult
-------
NameError                             Traceback (most recent call last)
<ipython-input-14-15f3f6bdf7c6> in <module>()
----> 1 gen_mult

NameError: name 'gen_mult' is not defined
```

11.7 Decorator example in Python

```
In [7]: def decorate(func):
   ...:     print ("Decorate")
   ...:     def wrapper(*args, **kwargs):
   ...:         print ("Wrapper")
   ...:         a = list(args)
   ...:         a.reverse()
   ...:         print ("Args␣inverted␣:")
   ...:         print(a)
   ...:         return func(*args, **kwargs)
   ...:     return wrapper
```

```

In [8]: @decorate
   ...: def foobar(*args):
   ...:   print (",␣".join(args))

In [9]: foobar("A", "B", "C", "D")
Out[9]: Decorate
        Wrapper
        Args inverted :
        ['D', 'C', 'B', 'A']
        A, B, C, D
```

11.8 Concurrent processes in Pharo

```
[ 1 to: 10 do: [ :i |
  Transcript nextPutAll: i printString, '␣'.
  Processor yield ].
Transcript endEntry ] fork.

[ 101 to: 110 do: [ :i |
  Transcript nextPutAll: i printString, '␣'.
  Processor yield ].
Transcript endEntry ] fork
```