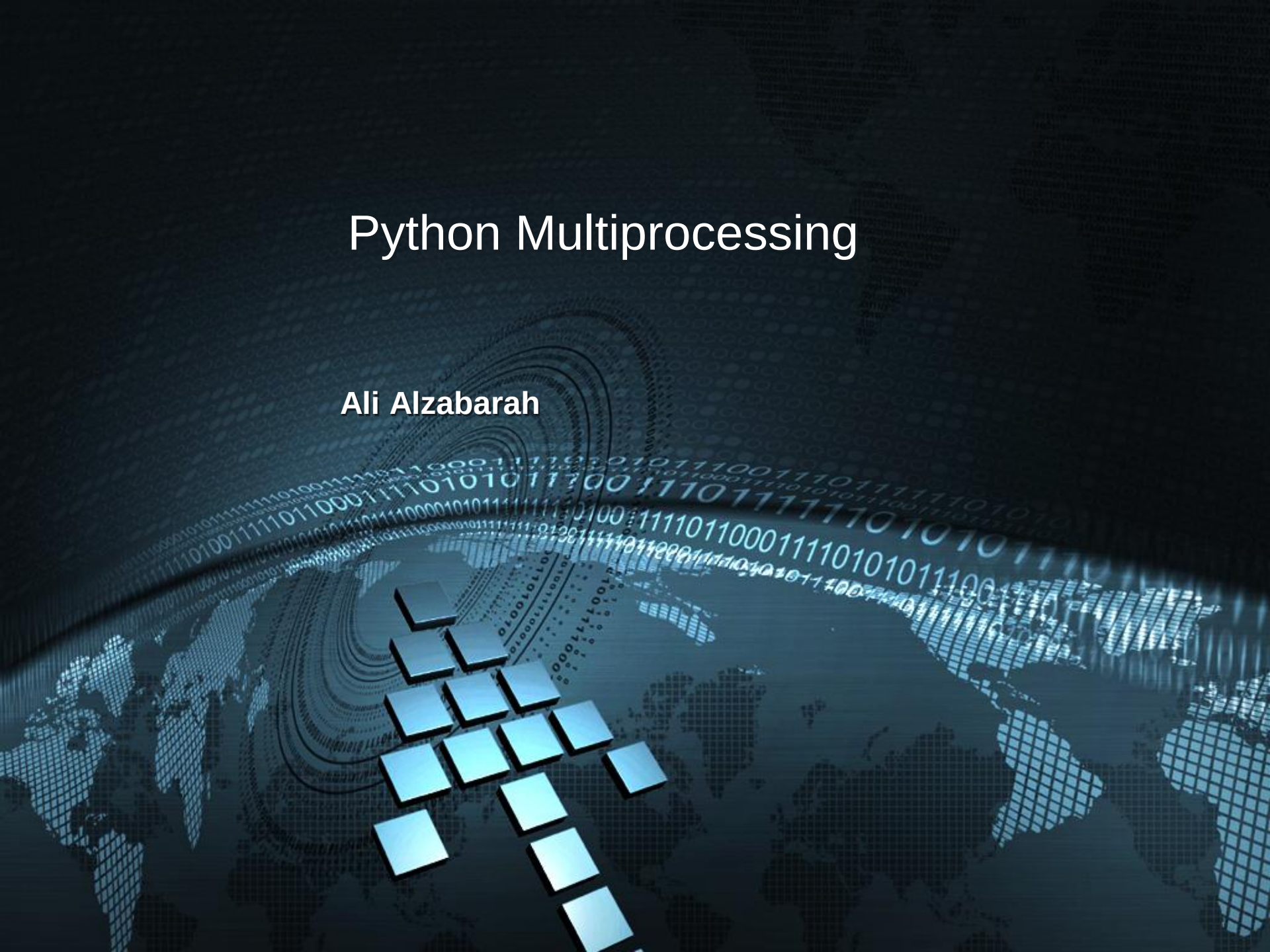
Python Multiprocessing

Ali Alzabarah



1. Introduction
2. Python is OOP language
3. Python and concurrency
4. Multiprocessing

Introduction

- Python is a great language and one of the widely used languages nowadays.
- Python has been an OOP language from day one.
- Python is intended to be easy to learn language.
- Python combines the power of scripting and OOP languages.
- From OO point of view, Python is not as powerful as Java.

Python is OO language

- Python package is a collection of modules. Package is a directory contains modules.
  - **__init__.py file must exist in the directory to be considered a package.**
  - **__init__.py runs when the package is imported.**
- Python modules might contains zero or more classes.
  - **Its possible to have a module that has methods only.**

Python is OO language

- Python classes might contains zero ore more methods.
  - **Its possible to have class with no behavior and functionality.**
    - Class myClass: pass
- Python does not have access modifiers such as private, all class methods/attributes are public.
  - **i.e no private/protected methods. As Guido put it, “We are all adults”.**
    - Works around ? You bet !
      - But you still can access so called *private* methods and attributes if you know their names and the class they live in.

Python is OO language

- Lets see an example of a module called myModule.py :

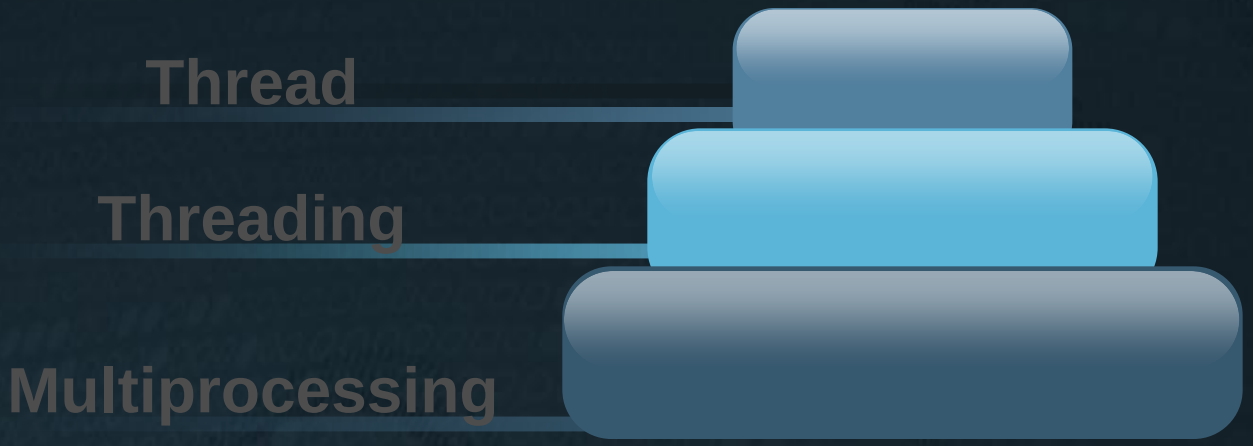
```
1  class myFirstClass:
2      myString = ""
3      def __init__(self):
4          self.myString = "OOP"
5
6      def getString(self):
7          return self.myString
8
9      def printString(self):
10         print self.myString
11
12 class mySecondClass(myFirstClass):
13     def __init__(self):
14         myFirstClass.__init__(self)
15
16     def getString(self):
17         return "anotherString"
18
19 myFirst = myFirstClass()
20 mySecond = mySecondClass()
21 myFirst.printString()
22 mySecond.printString()
23 print mySecond.getString()
```

Concurrency Introduction

- Since we are talking about concurrency, lets have some definitions.
  - **Thread : is a thread of execution in a program. Aka, lightweight process.**
  - **Process : an instance of a computer program that is being executed.**
  - **Thread share the memory and state of the parent, process share nothing.**
  - **Process use inter-process communication to communicate, thread do not.**

Python and Concurrency

- Python has three concurrency packages:



Python and Concurrency

- **Thread :**
  - Provides low-level primitives for working with multiple threads.
  - Python first implementation of threads, it is old.
  - Not included in Python 3 .
- **Threading :**
  - Construct higher-level threading interface on top of thread module.

Python and Concurrency

- **Multiprocessing :**
  - Supports spawning process.
  - Offer local and remote concurrency
  - New in python 2.6.
  - Solves the issues in the threading module.
  - For more information and examples please download my previous presentation : http://www.cs.colorado.edu/~kena/classes/5828/s10/presentations/ali_alzabarah_se_presentati.pdf

Multiprocessing

- I’m not going to analysis all the abstraction that multiprocessing package provides. I will focus on Pool and Forking modules.
  - **Pool module represents a pool of worker processes.**
  - **Forking module is responsible for creating/starting/terminating a process object using either fork for *nix based systems or CreateProcess for Windows.**
- Some of the solution I will provide can not be implemented in python. Remember, python does not support interfaces.

Pool of worker

- It hides the complexity of concurrency programing.
  - **You do not have to worry about managing processes or shared data or stats between workers.**
  - **You do not have to worry about distributing the work between workers.**
  - **All you need to do is :**
    - Create pool of X processes
    - Assign a task to those processes.

Pool of worker

- Example :

```
1  from os import getpid
2  from multiprocessing import Pool
3
4  # function that return the square of a number
5  # and the process id
6  def square(x):
7      return "result is : %s and pid is : %s" % (str(x*x), getpid())
8
9  # create pool with 8 workers
10 myPool = Pool(processes=8)
11
12 # data to process
13 myData = range(16)
14
15 # assign a task to the processes in the pool
16 myResult = myPool.map(square,myData)
17
18 # print out the result
19 print myResult
20
```

Pool of worker

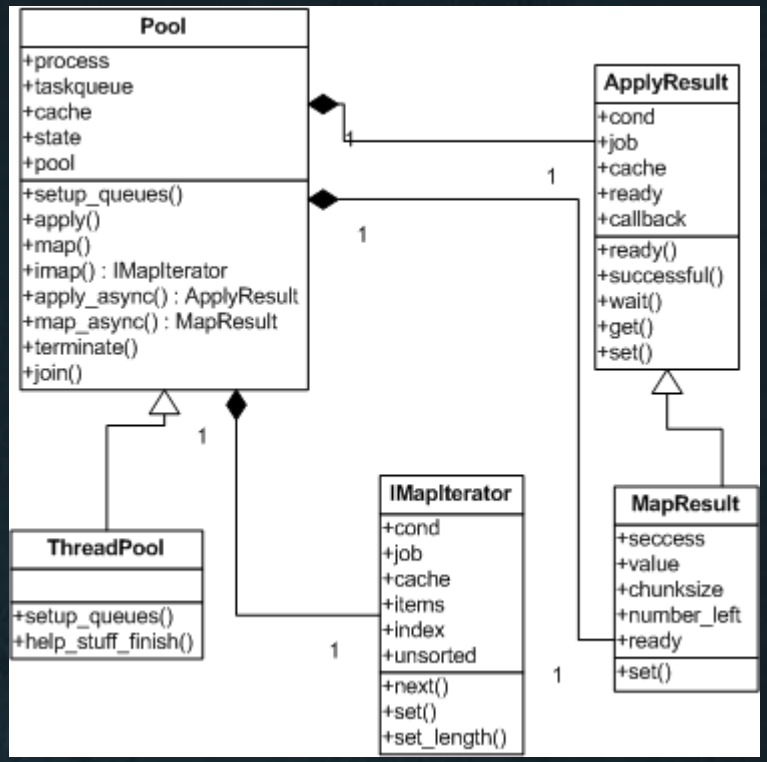
- The result :

```
['result is : 0 and pid is : 8363', 'result is : 1 and pid is : 8364',
'result is : 4 and pid is : 8365', 'result is : 9 and pid is : 8364',
'result is : 16 and pid is : 8367', 'result is : 25 and pid is : 8368',
'result is : 36 and pid is : 8369', 'result is : 49 and pid is : 8370',
'result is : 64 and pid is : 8368', 'result is : 81 and pid is : 8365',
'result is : 100 and pid is : 8366', 'result is : 121 and pid is : 8364',
'result is : 144 and pid is : 8367', 'result is : 169 and pid is : 8363',
'result is : 196 and pid is : 8366', 'result is : 225 and pid is : 8368']
```

Pool of worker

- Okay, That was easy and very good abstraction and encapsulation.
  - **That is a good hiding of the complex implementation.**
- Lets analyze the module to see how good the OO design and if the good abstraction is a result of a good design choices.

Pool of worker (class diagram)



Pool of worker (Analysis)

- Good Functions and attributes naming ?
  - No.
    - get() what ? help_stuff_finish !! Ready() !? isReady() is better.
- **Good module design ?**
  - No.
    - ApplyResult, MapResult and then ImapIterator.
      - » What is common about all those ? They are a result of a task.
      - » So why not “code to interface”. Why ImapIterator is left alone. It is a special form of result.

Pool of worker (Analysis)

- Oh, but python does not have interface. Okay, an abstract class. They do not have that either*.
  - **We could make a general class called “MultiProcessingResults” and make every result of a task inherit from it. We will take the common behavior and functionality and put it in that class. Every other class will inherit what is common and add more functionality if needed. (something is better than nothing)**

- Python 2.7 added module called ab that should provide the abstract class notion. However, I do not like the way they implement it. It is not part of the core language, it is just another module.

Pool of worker (Analysis)

- Weak cohesion and strong coupling :
  - **Pool constructor do the following :**
    - Call setup_queue() which is another function in the same class.
    - Create another task queue.
    - Create processes and name them.
    - Create a task handler.
    - Start and run the task handler ‘s thread.

Pool of worker (Analysis)

- Create a result handler.
- Start and run the result handler thread.

- Strong coupling :
  - **For example, imap method depends on many functions in the same class and other functions in another classes.**
    - Taskqueue.put(), get_tasks(), set_length() .. etc

Pool of worker (Analysis)

```
def __init__(self, processes=None, initializer=None, initargs=()):
    self._setup_queues()
    self._taskqueue = Queue.Queue()
    self._cache = {}
    self._state = RUN
    if processes is None:
        try:
            processes = cpu_count()
        except NotImplementedError:
            processes = 1
    self._pool = []
    for i in range(processes):
        w = self.Process(
            target=worker,
            args=(self._inqueue, self._outqueue, initializer, initargs)
            )
        self._pool.append(w)
        w.name = w.name.replace('Process', 'PoolWorker')
        w.daemon = True
        w.start()
    self._task_handler = threading.Thread(
        target=Pool._handle_tasks,
        args=(self._taskqueue, self._quick_put, self._outqueue, self._pool)
        )
    self._task_handler.daemon = True
    self._task_handler._state = RUN
    self._task_handler.start()
    self._result_handler = threading.Thread(
        target=Pool._handle_results,
        args=(self._outqueue, self._quick_get, self._cache)
        )
    self._result_handler.daemon = True
    self._result_handler._state = RUN
    self._result_handler.start()
    self._terminate = Finalize(
        self, self._terminate_pool,
        args=(self._taskqueue, self._inqueue, self._outqueue, self._pool,
              self._task_handler, self._result_handler, self._cache)
```

Pool of worker (Analysis)

- Easy to maintain code ? No.
  - **This is just an expected result of strong coupling and weak cohesion.**
- Delegation :
  - **Map and apply methods delegate to map_async and to apply_async. Both methods are in the same class. The different is :**
    - Only in the return type. {map,apply}_async returns {apply,map}_results object whereas map and apply return an array. Better solution ? Next slide

Pool of worker (Analysis)

- Why not have a class called Result and make all of those functions return “result” object. The client then can chose which result format he wants weather its iterator, string, array ..etc.
  - **Result objects can have a “getResult” function that have many forms depending on what the type that client want.**

Pool of worker (Analysis)

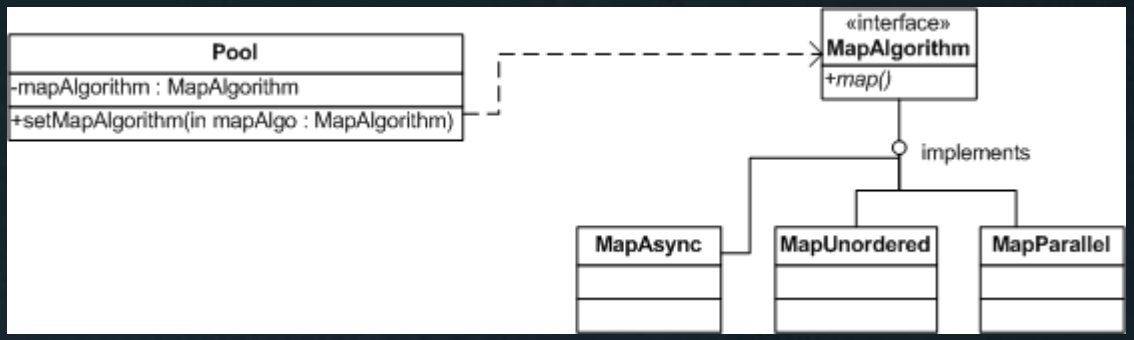
- I also noticed that Pool class has map, map_async, imap, imap_unordered. All of them takes a function and an iterable. We can use Strategy pattern here.
- We can have “mapAlgorithm” interface and have map/map_async … etc implement that interface. The client then can decide at run time which algorithm to use.

Pool of worker (Analysis)

- **We will then apply the following design patterns principles :**
  - Encapsulate what varies
  - Code to interface
  - Favor composition over inheritance.
- **Let’s see a possible diagram in the next slide.**

Pool of worker (class diagram better)

- Possible diagram for the suggested solution :



Pool of Workers (Conclusion)

- The module is very simple to use but poorly designed. It is not easy to maintain and will not respond well to future changes.
- I believe they focused on “functionality” of the module more than “design” choices. In another word, it does what they want to do in a simple and easy way. No complexity or consideration of future changes.

Pool of Workers (Conclusion)

- IMO, the module can greatly improved by
  - **using the strategy pattern.**
  - **Unifying the return type of the functions. Also, make them return one “result” object that has function that takes many forms. This will make the output a client choice and thus more flexibly.**

Forking Module

- Python works on many platforms. Hence, the forking module was designed to handle process creating/running/terminating on Windows/Unix based operating systems.
- Forking module is responsible for creating process either using fork system call (*nix) or CreateProcess (Windows)

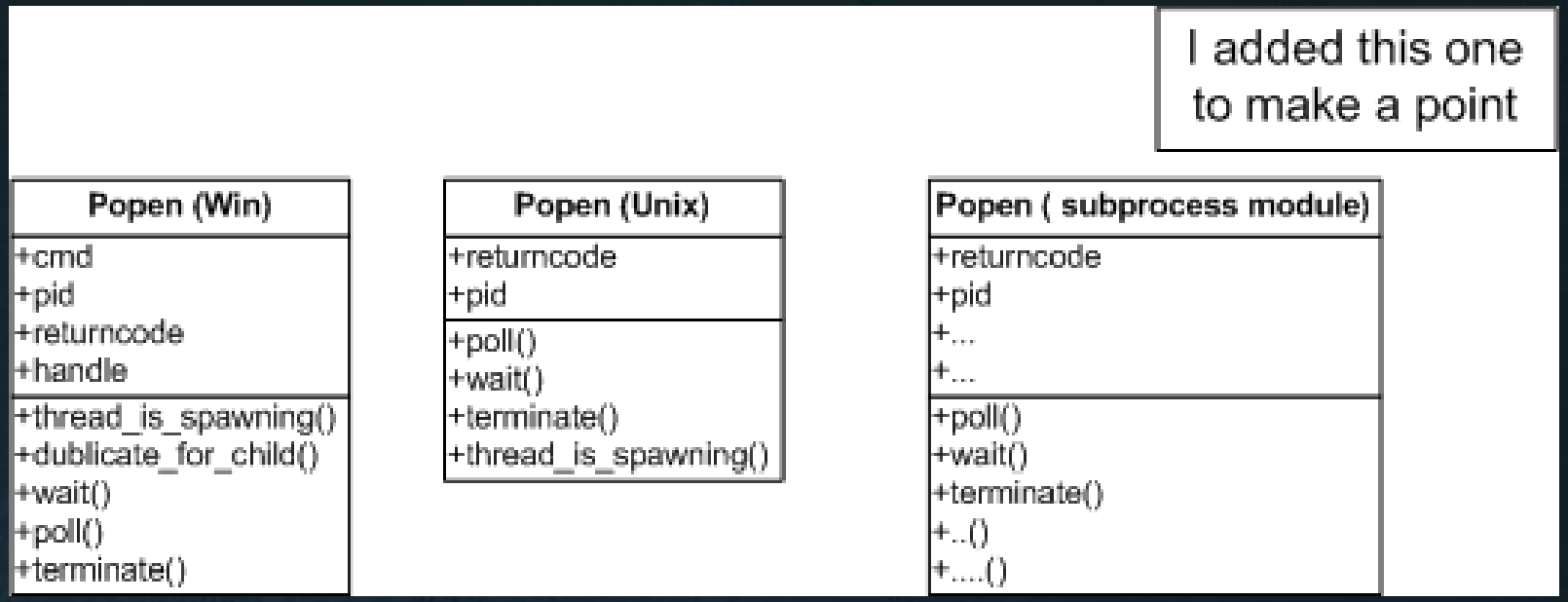
Forking Module

- The big picture of the module is one if statement that checks for the OS type. The module has two classes with the same name, Popen. One of the classes is inside if statement and the other one is inside else statement. When the class is imported, only one will be in the module namespace based on the OS type.

Forking Module

- The two classes do the same job and both of them are similar to another class called Popen in another module called subprocess.
- To start our analysis, I'm copying/pasting a note from a python developer in that module
  - **"We define a Popen class similar to the one from subprocess, but whose constructor takes a process object as its argument."**
    - Now, I'm not expecting a good OO design choices. Let's see.

Forking Module (Class Diagram)



Forking Module (Class Diagram)

- The previous slide clearly shows the problem :
  - **Three classes : same names, same functions name, same functionality but in different files.**
  - **Further analysis, results in these observations :**
    - Little or no code reuse.
    - No usage or little usage of any of the design pattern principles.
      - One would expect at least a heavy usage of delegation over inheritance to minimize repeated code.

Forking Module (Analysis)

- What are the common functionality those classes provides :
  - **Create a process**
  - **Wait for a process**
  - **Terminate a process**
  - **Store process id**
- What varies among those classes :
  - **In general, the Internal implementation. That is the system call used to create/terminate process.**

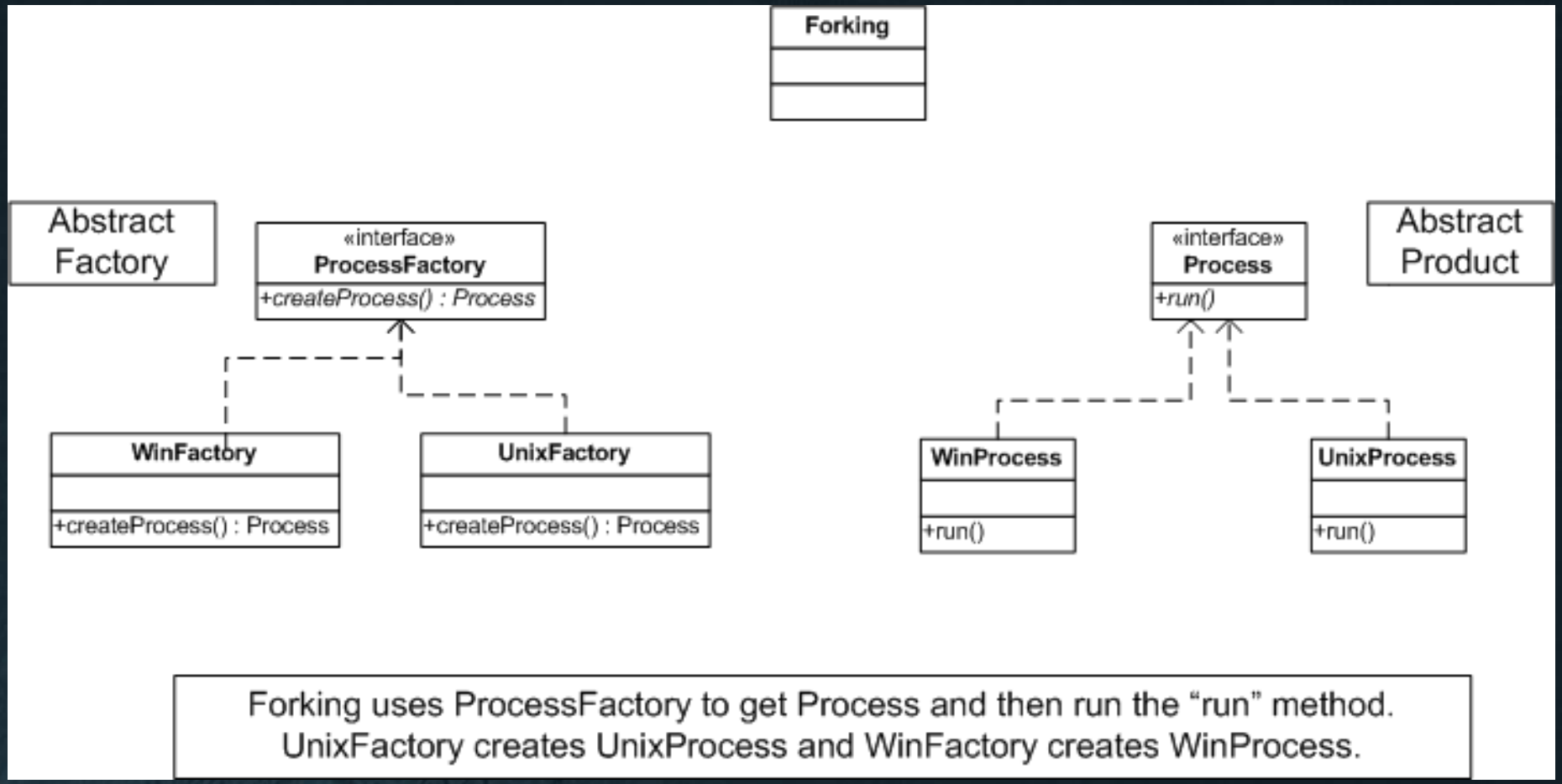
Forking Module (Analysis)

- What can be done to improve those three classes ?
  - **Use Abstract Factory design pattern. It will really help. Lets assume Python supports interfaces.**
    - ProcessFactory interface which has createProcess method.
    - WinFactory implements ProcessFactory. Its createProcess method return “WinProcess” object.

Forking Module (Analysis)

- Another interface called Process that has, for example a run method.
- WinProcess implements Process and override run method.
- Now we have Forking class that takes ProcessFactory at the construction time and call the createProcess function. We will then run the “run” method in Process object.

Forking Module (Better Class Diagram)



Summary

Design Pattern Principle	Forking	Pool
Code to interface	No	No
Encapsulate what varies	Little usage	Little usage
Classes are about behavior	Yes	Yes
Delegation over Inheritance	Little use	No
Coupling/Cohesion	Strong/weak	Strong/weak

Final Thoughts

- Multiprocessing module is a powerful addition to python. it introduced easy way to achieve true parallelism.
- Multiprocessing is poorly designed for different reasons :
  - **Python is not Java, no Interfaces.**
  - **Focus on functionality more than design patterns.**
  - **No/little design pattern usage.**
  - **Lots of violation to lots of design pattern principles.**

Final Thoughts

- **Open source project ! Lots of changes by different developers. Maybe not or little communication between developers resulted in 3 Popen classes.**
- Python needs to provides support to Interfaces/Abstract classes. It will greatly help it be better.
- Python modules are created to solve specific problem at hand instead of thinking about the language as a whole.

Final Thoughts

- Thread, Threading and multiprocessing modules could be combined in on module using *Facade pattern design*. i.e : concurrency package.

Python class analysis tips

- While analyzing I came across some functions that were helpful.
  - **To get all class methods**
    - Inspect.getmembers(classObj,predicate=inspect.ismethod)
    - Or use dir(classObj)
  - **To get class name out of class object**
    - classObj.__class__.__name__
  - **To get class attributes**
    - classObj.__dict__

References

- Python 2.6 documentation, http://docs.python.org/library/multiprocessing.html
- PyMOTW by Doug Hellmann, http://www.doughellmann.com/PyMOTW/multiprocessing/
- Learning Python by Mark Lutz
- Python class tutorial http://docs.python.org/tutorial/classes.html

Thank You!