OTools

Release 0.1.0

Jun 10, 2020

Contents:

OTools stands for Online Tools, which is a Python/Cython framework for developing multithread online systems.

This framework provides a simple way to deploy online systems by providing some concepts of multithread systems such as program scopes, shared memory, locks and threads in a more understandable and easier to implement way.

CHAPTER 1

OTools 101

There are basically 7 classes you need to understand in order to use the framework at its full potential: *Logger*, *Context*, *Dataframe*, *Service*, *Trigger*, *OTools* and *Watchdog*.

While explaining each one of them, we'll build and run a SimpleExample.

1.1 Logger

The *Logger* is the messaging core. It provides complete logging by showing from which context and module it has been called, making it easy to debug code.

You don't really need to worry constructing this module as the framework will handle it for you, all you have to understand here are the levels of logging.

It has 6 levels of logging, which were parsed into a class, in order to ease the use of it:

`LoggingLevel.`**`VERBOSE`**

- **Color**: white
- **Shows**: *VERBOSE*, *DEBUG*, *INFO*, *WARNING*, *ERROR* and *FATAL*

`LoggingLevel.`**`DEBUG`**

- **Color**: cyan
- **Shows**: *DEBUG*, *INFO*, *WARNING*, *ERROR* and *FATAL*

`LoggingLevel.`**`INFO`**

- **Color**: green
- **Shows**: *INFO*, *WARNING*, *ERROR* and *FATAL*

`LoggingLevel.`**`WARNING`**

- **Color**: bold yellow
- **Shows**: *WARNING*, *ERROR* and *FATAL*

`LoggingLevel.`**`ERROR`**

- **Color**: red
- **Shows**: *ERROR* and *FATAL*
- **Raises**: tries to raise any identified error on execution. If none found, doesn't raise anything.

`LoggingLevel.`**`FATAL`**

- **Color**: bold red
- **Shows**: *FATAL* only
- **Raises**: tries to raise any identified error on execution. If none found, raise FatalError.

1.2 Context

The *Context* is the core of communication between the modules of the framework. Objects of type *Service*, *Dataframe* and *Trigger* can be attached to it, and so it allows you to access them from your own running *Service* without worrying too much.

Your system can orchestrate one or many *Context* objects, some of them sharing *Dataframe* objects or whatever, it's up to your imagination.

The most important methods you need to know in order to make it work are the following:

`Context.`**`__init__`**(*level = LoggingLevel.INFO*, *name = "Unnamed"*)
: The constructor sets as default *INFO* as the logging level and *Unnamed* as the name of the context. Notice that there might not be two *Context* objects with the same name running on the same instance of *OTools*.

`Context.`**`__add__`**(*obj*)
: You can add only *Service*, *Dataframe* and *Trigger* objects to the context. Each of them will have different handling, but be aware that *Trigger* and *Service* objects can't have the same name on the same context. *Dataframe* objects also can't have the same name on same context.

Beginning our SimpleExample, we now construct the *Context* for this:

```
from otools import Context, LoggingLevel

context = Context(level = LoggingLevel.INFO, name = "MyContext")
```

1.3 Dataframe

The *Dataframe* is an object that's meant to work like shared memory space. You can set multiple values to different keys and get them from every *Service* attached to the same *Context* as the one containing the *Dataframe*.

Dataframe objects can be attached to multiple *Context* objects in order to share it among everything you need.

`Dataframe.`**`__init__`**(*name = "Dataframe"*)
: The constructor sets as default *Dataframe* as the name of the object. Remember that there can't be two *Dataframe* objects with the same name attached to the same *Context* object.

`Dataframe.`**`get`**(*key*, *blockReading=False*, *blockWriting=True*, *timeout=-1*)
: This function gets the value of the key *key* attached to this *Dataframe* object. If the key is not set, it logs an error message and returns *None*.

The *blockReading* and *blockWriting* flags are used to set if the locks for reading and writing of the *Dataframe* are acquired on this *get*. The *timeout* is how long it should wait for acquiring those locks, in case it needs to wait.

By default, on the *get* function, reading is not blocked but writing is. Timeout is infinite.

Note: all locks are released automatically by the framework.

Dataframe.**set**(*key*, *value*, *blockReading=True*, *blockWriting=True*, *timeout=-1*)
Similar to the function *get* on the parameters and explanation. The only different parameter here is *value*, which is the value you want to set to the key *key*.

Best practices:

- If you need a function to return a value for you to set it into a key, *DO NOT* do this:

```
dataframe.set('myKey', function(myArgs))
```

because you will lock your dataframe until your *function* returns. This happening, every other module that needs access to this dataframe will be locked too.

The best to do in this scenario is:

```
value = function(myArgs)
dataframe.set('myKey', value)
```

as this won't hold the dataframe until the function returns.

Continuing the example:

For the sake of simplicity, I'll construct just one *Dataframe* and attach it into the single *Context* we've built before:

```
from otools import Dataframe

dataframe = Dataframe(name = "MyDataframe")
context += dataframe
```

1.4 Service

A Service is a metaclass that shall encapsulate your own code in order to attach it into a *Context* object. As it encapsulates another class, it will make few methods available for it, in order to interact with the framework.

The class that will be encapsulated can implement few methods of its own that will interact with the framework. They're:

- *setup*: this method will set your class up by configuring your environment and everything you need to do before running it;
- *main*: this method will run once for every loop in the *Context* execution;
- *loop*: this method will run in loop in an exclusive thread, not depending on the *Context* execution loop;
- *finalize*: this method is the shutting down procedure, it says what your class should do before ending.

The way these methods interact with the framework will get clearer when we get to our orchestrator, *OTools*.

Methods that will be available for the class after encapsulation are the following:

Service.**MSG_VERBOSE**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.VERBOSE*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**MSG_DEBUG**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.DEBUG*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**MSG_INFO**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.INFO*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**MSG_WARNING**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.WARNING*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**MSG_ERROR**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.ERROR*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**MSG_FATAL**(*message*, *moduleName="Unknown"*, *contextName="Unknown"*, **args*, ***kws*)
Log a message with level *LoggingLevel.FATAL*. *moduleName* and *contextName* will be filled by other modules on the framework.

Service.**getContext**()
Returns the *Context* object to which your *Service* is attached.

Service.**deactivate**()
Shuts this *Service* down without interfering on the *Context*.

Service.**reset**()
This method is used by the *Watchdog* module. It resets this single *Service* by re-creating it over itself.

Service.**terminateContext**()
This method shuts everything down on the *Context*. If this is the only *Context* running, this will shutdown the framework.

Service.**getService**(*serviceName*)
This will return the *Service* object identified by the name *serviceName* attached to the *Context*.

Service.**getDataframe**(*dataframeName*)
This will return the *Dataframe* object identified by the name *dataframeName* attached to the *Context*.

Besides those methods that will be available for your class, the *Service* object also has few of their own:

Service.**setup**()
This will run the *setup* method of your own class.

Service.**main**()
This will run the *main* method of your own class.

Service.**loop**()
This will feed the *Watchdog* and run one iteration of the *loop* method of your own class.

Service.**finalize**()
This will run the *finalize* method of your own class and deactivate this.

Continuing the example:

For the sake of simplicity, I'll construct just two *Service* and attach them into the single *Context* we've built before:

```
from otools import Service
from time import sleep

class MyCode ():
   # All methods here are optional
   def setup (self):
      # Do whatever you need here
      pass
   def main (self):
      # I'll just print something
      self.MSG_INFO("Hey, I'm being executed!")
      sleep(2)
   def loop (self):
      # I'll print here using WARNING level
      self.MSG_WARNING("A warning log here!")
      sleep(1)
   def finalize (self):
      # Stop!
      self.MSG_INFO("Shutting down...")
      pass

class MySecondCode ():
   def setup(self):
      self.loopCounter = 0
   def main(self):
      self.loopCounter += 1
      if self.loopCounter >= 5:
         self.MSG_INFO("I'm shutting everything down!")
         self.terminateContext()

context += Service(MyCode)
context += Service(MySecondCode)
```

Notice that, when this runs, for every *Context* execution loop, *MyCode* will log an *INFO* message and *MySecondCode* will count the number of loops. If the *loopCounter* is equal or greater than 5, *MySecondCode* will terminate the *Context*. As this is the only *Context* running, it will shut the whole software down.

Meanwhile *MyCode* will, in an exclusive thread, print *WARNING* messages until the *Context* shuts down.

1.5 Trigger

Trigger is a class for implement triggering actions on the framework. After being constructed, *Service* and *TriggerCondition* objects are allowed in order to configure this.

For interacting with *Trigger* objects, these are the methods you should know:

`Trigger.__init__`(*name = "Trigger", triggerType = 'or'*)
: When constructing this type of object, as it's similar to *Service* objects, a *name* will be assigned and it must not conflict with any *Service* names attached to the same *Context*. The *triggerType* allowed options are:

  - *and* : all conditions must trigger;
  - *or* : any of the conditions must trigger;
  - *xor* : make a XOR on the conditions, in the order they were attached.

`Trigger.__add__`(*a*)
: As previously said, only *Service* and *TriggerCondition* objects are allowed to be attached to *Trigger*

objects.

All *Service* objects will join the execution stack of the *Trigger*. Notice that *loop* methods will be ignored on *Trigger* stack executions.

All *TriggerCondition* objects will join the list of conditions it must attend according to the *triggerType* in order to run the execution stack.

A *TriggerCondition* is a class for encapsulating other classes in order to identify it as a condition not as anything else. It has only one important method, which is:

`TriggerCondition.main()`
This method will run the *main* method of your encapsulated class and the answer (*True* or *False*) will say whether this *TriggerCondition* will trigger or not.

No *Trigger* objects will be used on our SimpleExample, but here's a code snippet in order to get things clearer:

```
from otools import Service, Context, OTools, Trigger, TriggerCondition, Dataframe
from time import sleep

class Increment ():
   def main(self):
      df = self.getDataframe('counter')
      if df.get('counter') == None:
         df.set('counter', 0)
      df.set('counter', df.get('counter') + 1)
      sleep(1)

class Shutdown ():
   def main(self):
      self.terminateContext()

class MyCondition ():
   def main(self):
      df = self.getDataframe('counter')
      if df.get('counter') >= 5:
         return True
      else:
         return False

ctx = Context()

trigger = Trigger()
trigger += TriggerCondition(MyCondition)
trigger += Service(Shutdown)

df = Dataframe('counter')

ctx += Service(Increment)
ctx += trigger
ctx += df

main = OTools()
main += ctx
main.setup()
main.run()
```

1.6 Watchdog

1.7 OTools

Python Module Index

Index

Symbols

C

D

E

F

G

I

L

M

O

R

S

T

V

W