Wendelin - Open Source Big Data Platform

The Wendelin platform is part of an ongoing research project Nexedi is leading. The goal is to develop the technological framework for a open source Big Data platform "Made in France". Wendelin will integrate libraries widely used in the data science community for data collection, analysis computation and visualization. The research project also comprises development of first prototype applications in the automative and green energy sector as it's purpose is to provide a ready-to-use solution applicable in different industrial scenarios.

One Stack To Rule Them All

The Wendelin stack is written in 100% Python. On the base layer, SlapOS handles configuration, deployment and management of all components running on the Wendelin stack. Distributed storage is provided by NEO while ERP5 is used as platform to connect the various libraries, store data, provide a connecting user interface plus enable the creation of web-based Big Data applications up to integration of more complex business processes ("Convergence Ready"). At the heart of Wendelin is "Wendelin Core", component that will provide out-of-core computation capabilities allowing Wendelin based stacks to go beyond the limits imposed by available RAM in a cluster of machines. On top of this stack different libraries will be integrated - most importantly Scikit-Learn for machine learning and Juypter for the interactive development loved by many Python developers.

New features of Jupyter Kernel

In the blog post about the release of Wendelin 0.5it is said that Wendelin now runs in "cluster mode", which allows code to be executed by a cluster of Zope nodes. This means that, when you execute a code cell, one process in a server is selected to run your code, but if you run it a again a completely different process, in another different server, can be chosen to run your code. In this blog post you will find more details about our implementation that supports this distributed architecture and helps users create code that can run in any server at any time and produce the same output and other new features for quality of life improvements.

The environment object

One of the key concepts of our distributed code execution solution was the so called "environment object". It was designed to be able to store definitions that are hard to send between Python processes (like functions, classes, module setups and more complex objects) and allow each process to load them on demand as the users execute their code. Every notebook created in a Jupyter instance will now include a short demo and explanation of how the environment object works and why it was created.

[PICTURE OF ENVIRONMENT OBJECT EXPLANATION AND DEMO]

Under the hood, the environment implementation is complex, because of natural Python problems with sharing some objects among processes. The "environment" variable itself is a dumb object: just a simple class with "define" and "undefine" methods that do nothing. All the hard work is done by an AST (abstract syntax tree) processor that walks through the user's code before execution. It is capable of capturing any function definition as string (easily shared between different processes) and, if this function returns an instance of the "dict" class, it will be merged with the current code execution context. In addition, when there's an error in a setup function's code, execution is immediately stopped and, along with the error itself, a message tells the user that error came from one of his setup functions.

This tool provides a safe mechanism even for imports and modules that hold global settings, as the famous "matplotlib" module does: the environment setup of each user is executed just before his code and overrides the configuration in that given Zope process for the moment. When a user try to import a module in the "usual" way, it's automatically fixed by another of our pre-processing rules to use an environment definition and the user is given a proper warning about what happened. This way we also avoid unintentional interference by prior user's code execution.

Unsupported objects

Because the user context is saved in the ZODB, there are problems when the user tries to save variables that cannot be persisted inside it. In this situation the object is not stored in the context and code execution proceeds as expected. In the output the user will see a warning telling him which variable holds a reference to an unsupported object and a recommendation to use the environment object to load this object. Even objects inside containers (lists and dicts, i.e.) are

detected during context storage.