core.ns Documentation

Release latest

Apr 30, 2020

Current core.NS documentation

Caution: core.NS is in constant state of development and improvement. While I am striving to keep backward compatibility the best I could, please pay close attention to "What's New" chapter of the documentation.

There are many Python Application Frameworks and when I am to introduce you another one, called core.NS. What is the **core.NS** and what it's place in the landscape of the Python packages ?

CHAPTER 1

What is **core.NS** ?

core.NS is a core application library, built around the idea of the Namespaces . This idea may be a bit unfamiliar to a hardcore developers, who grow in the world of "Object-Oriented Programming", procedural languages, globals and local procedures and variables, scopes and many other things, that you've spend your life around. Although, this idea will be very familiar to everyone, who've been around Unix OS. **core.NS** is a library, for creating and working with "Unix filesystem"-like namespace, created inside application. There will be */bin*, */dev*, */home* and other placeholders you are familiar with. But instead concept that's "everything is file", *core.NS* proposes concept "Everything is a data that you can set and read"

Note: core.NS ZEN rule #1 - *"Everything is a data and everything is accessible and modifiable through a V() procedure."* including functions. Functions are the *"First-Class citizens"* and treated exactly like any data.

Every function and data on **core.NS** namespace addressable by its path, which is exactly like Unix filesystem path. Since data and a code are only logically separated, you do have a total freedom on how allocate them, but me, I am following "good-ol'", time-tested Unix filesystem patterns.

CHAPTER 2

What is the place of the **core.NS** in the landscape of the Python packages

core.NS is an *"Application Framework"*, means that is the tool for creating applications. And its have everything you need to do that:

1. functions and data access primitives
2. low-level functionality is separated from your application code with help of *drivers*
3. command-line arguments parsing. You do not have to create parser. The one already provided for you. Just define help texts and functions.
4. Startup and Shutdown functions. You just have to define the functions. All logic of the startup and shutdown execution is provided.
5. "Smart" console
6. "Smart" log subsystem
7. Support for cooperative multitasking via gevent

2.1 First steps in core.NS

> **Warning:** **core.NS** has been developed for Python3. Ans specifically for Python 3.6+ There is no plans to backport core.NS to Python2. Sorry, folk!

As we already know, the very first steps in **core.NS** is initialization of the Namespace and receiving ability to call functions and access data objects on the Namespace.

```
Python 3.7.4 (default, Jul  9 2019, 18:13:23)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from corens  import *
>>> ns, f, F = NS()
```

(continues on next page)

(continued from previous page)

```
6 >>> V = f("V")
7 >>> V("/config/answer")
8 42
```

- On line 4, we are importing a basic functionality of the **core.NS**. Usually, that is all that you will need.
- On line 5, we are creating new Namespace. This call returns three elements: * Reference to a namespace itself. Namespace in fact, is a one, big, complicated Python dictionary. Nothing more than that. You can work with Namespace directly, as you would work with any dictionary, but it is not recommended. * Reference to a function *"f()"*. This function is searching and returning you a reference to any other function stored in Namespace. * Reference to a function *"F()"*. This function acts similarly to an *f()*, but instead of returning the reference, *F()* install this function in Python builtins
- On line 6, we are referencing function *V()*, which we will need to access data elements.

2.2 Namespace

Note: core.NS ZEN rule #2 - Well regulated namespace of the data and code elements are more manageable and comprehensible, then an artificial maze of the objects and cryptic defaults for the local and global variables. In a Namespace, you can't be wrong, because, what's you place in the element defined by the path, that's what you will get. All the time.

Namespace - it is a in-memory (actually not only in-memory) tree-like collection of the data elements. Considering, that the functions are the "First-Class Citizens", those data elements also include functions. Each element defined by it's path. */bin/V* , */config/answer*, */home/MyData*. There are two types of the data elements.

- Directories. Structure, which can not hold any data by itself, besides certain *metadata*, but which can hold a references to other data elements. For example */bin/V*, directory */bin* keeping a reference to a data element *V*.
- Data element. It is an atomic, indivisible, placeholder for storing an actual data. Data elements can store a references to another elements, but they are not a directories and they are terminating the path. For example:
  - Directory */bin* can form a finite path, if you are referring this directory, or be a part of the path to a data element, like */bin/V*
  - Data element forming a finite path with the name of the data element at the end. Only directories can be an intermediate parts of the path.

In order to refer an element in the namespace, you must know its path. For functions, **core.NS** provides special *syntax sugar* which will simplify the search. But for other data elements, function *V()* expects the passing of the full path to the element. There is no *current* or *relative* path. references. Only full path, which provides unquestionable, direct reference of the data element. Function returns *None*, if data element not exists.

2.3 Functions

Note: core.NS ZEN rule #3 - Object-Oriented approach is nice, but it doesn't mean that you have to use it everywhere. Good functions are hard to beat. Write functions !

What's differentiate core.NS function from regular Python function. There are reference to a global context, passed to the function as first parameter. If function is a part of the drivers definition, there are two default first parameters. You

would say that it will be mundane to pass the same parameter to the each and every functions defined for **core.NS**. Not if you are using *partially applied functions*.

What is the *partially applied function* ? Without going too deep in the theory of Lambda calculus and Functional Programming, I'll try to give a perfectly simple definition to this, in the reality, quite complicated term. *Partially applied function* is such function which is bound with some of its parameters without actual execution of this function. The reference on the function and bound parameters are serving as a reference on this function that you can call, without specify parameters, that you already bound. Let me illustrate this concept with this simple example:

```
1   from corens  import *
2
3   ns, f, F = NS()
4   V = f("V")
5   I = f("I")
6   V("/home/counter", 0)
7
8   def add1(ns, _path):
9           V = f("V")
10          V(_path, V(_path)+1)
11          return V(_path)
12
13  I("/bin/add1", add1)
14
15  f("add1")("/home/counter")
16  print(V("/home/counter"))
```

The first news is on the line 5. We are referencing function *I()*. This function creates *partially applied function* and stored it in **core.NS** namespace for a latter use. Line 6, we are initializing our counter on the namespace. Then we are creating a what looks like a normal Python function. But, wait. Look at the first parameter - this is **core.NS** function, taking a reference to a namespace. Next "line of the interest" is 13. What you can find inside the function add1, should't cause any troubles. Remember: **core.NS** functions are not pure and *V()* is a function for accessing data stored on namespace. And of course, if you look at source code of *V()* defined in ns.py, you will see that's the V is sure, **core.NS** function too. So, let's take a look at line 13. We are defining function with path */bin/add1* and the function is add1 as we defined it. This function will be converted to a *partially applied function* by the function *I()*. So *I()* is just a *syntax sugar* for *V()*. You can create a partially applied functions with nothing but *V()*, but I will spare details of "now" for now. Then on line 15, we are referencing function that we just define and call the reference with parameter. Remember, *f()* return you the reference on the *partially applied function*. Line 16 shall bring you the fact that the value stored in *"/home/counter"*, indeed increased.

So, the **core.NS** namespace do store *partially applied functions* for which you do not have to remember to pass the first parameter. Parameter bound is "no-error" parameter.

2.4 Drivers

Note: core.NS ZEN rule #4 - Drivers are the perfect technics to hide low-level logic from the high-level logic and organize communication between them via functons

What is the "low-level logic" and why drivers in **core.NS** ? Please allow me to give a simple definition of what is **core.NS** driver:

The core.NS driver is a element of the code, which besides the reference to the global namespace, having a reference to a context

Essentially, each driver it is a directory, holding some context-specific data as well as *partial applied functions*, each

of them is with two bound arguments: one to a global namespace, another one is to a directory, keeping context. The closest "relative" of the **core.NS** drivers is a Unix drivers with contexts in */dev* . There are low-level logic to work with, for exaple */dev/sda*, */dev/sdb* and so on. Each of this logics, is provided with context and without context are rather abstract.

Let me bring some example. We will extend a functionality of the **core.NS** with the counters. Context for counter driver implementation will be stored in */dev*. Each counter shall have a unique name. Each named counter must have an easy to understand, namespace-based interface. For example, we can call function *f("/dev/counter/create")()* to create counter and *f("/dev/c/$countername/++")()* to increase the value of the counter.

```
      def _counter_open(ns, ctx, name):
              c = nsGet(ns, "/dev/c/{}".format(name))
  if c is not None:
      return c
  else:
      c = nsMkdir(ns, "/dev/c/{}".format(name))
                              nsSet(ns, "/dev/c/{}/counter".format(name), 0)
                              nsSet(ns, "/dev/c/{}/++".format(name), partial(_counter_
↪increase, ns, c))
              return c

      def _counter_increase(ns, ctx):
              _path = ctx["__name__"]
              c = "{}/counter".format(_path)
              nsSet(ns, c, nsGet(ns, c) + 1)
              return nsGet(ns, c)
```

Unlike in previous examples, in low-level implementations, I am recommending to use functions from *corens.ns* than *V()*. There are reasons for that. Function *_counter_open()* takes extra parameter - name. When we will know the name, first, we are checking if such name in */dev/c* already exists. If not, we shall create directory, initialize default value for the counter and create context-sensitive *partially-applied* function ++

After that, wrap it in *_tpl = {}* section of the module that you are adding to your **core.NS**.

```
_tpl = {
                'counters': {
                                'create': _counter_open,
                }
}
```

And then use *Mk()* function to initialize driver in */dev* tree as *Mk('counters')*

2.5 What's new ?

2.6 Manpages

/bin/stamp Return a float number with current Unix timestamp

Index

Symbols