Continue

# Python set environment variable

It's crucial to avoid passing untrusted variables to subprocesses using shell=True, as this poses a significant security risk due to the potential for arbitrary command execution. If unavoidable, utilize python3's shlex.quote() to escape strings, especially when dealing with multiple space-separated arguments. The default and safer approach is using shell=False, where arguments are passed as an array. For secure solutions, consider the following methods: Changing your process's environment affects both Python and its subprocesses. This can be achieved by modifying os.environ and then executing the command with subprocess.check_call(). An alternative is to create a copy of the environment, modify it, and pass it to the subprocess, allowing for control over the child's environment without affecting Python's own. For Unix systems, executing env to set environment variables offers another option, though it can be cumbersome with multiple variables. It retains full control over both Python and child environments. When dealing with space-separated arguments, composing a command array that splits the string is necessary to retain original behavior. In cases where environment modification is needed, solutions like subprocess.Popen with an updated env dictionary can be used, but this approach has limitations, especially with variable names that are not valid Python identifiers or contain non-alphanumeric characters. Given that modern operating systems restrict child processes from altering their parent's environment, and considering a Python interpreter's role as a child of the calling shell, alternative approaches should be explored when environment changes are required across process boundaries. Changing the environment of a parent process in Python isn't straightforward. In older systems, it was possible using low-level memory functions and modifying the COMMAND.COM's environment string. However, this approach requires system cooperation and is considered a hacky solution. In modern operating systems like Windows, it may still be possible with permission settings. Nonetheless, I would advise against doing so in Python as it can lead to issues. A more feasible approach is starting a new shell process with a modified environment. The `os` and `subprocess` modules provide the necessary functions for this: ``` import os import subprocess env = os.environ.copy() # Modify the environment copy at will... env['AAA'] = 'BBB' p = subprocess.Popen(["/bin/sh"], env=env) p.wait() ``` The `os.environ` dictionary behaves like a Python dictionary, allowing for common operations such as getting, setting, and checking for key existence. In Python 3, use the `in` keyword to check if a key exists: ``` >>> 'HOME' in os.environ True # In Python 2: >>> os.environ.has_key('HOME') True ``` It is essential to note that changes made by other scripts while your Python script is running will not be reflected when calling `os.environ`. This is because the environment mapping is captured at the time of Python startup, and subsequent changes are not updated. Finally, setting an environment variable using `os.system` or `os.putenv` only affects the current process and its child processes. It does not set the variable globally. You cannot set environment variables directly in Python. However, you can use the `contextlib` module to temporarily set environment variables within a specific scope. There are two implementations provided: `set_env` and `modified_environ`. The `set_env` function sets environment variables for the duration of the context manager block. It creates a copy of the current environment variables before updating them, ensuring they can be restored afterwards. The `modified_environ` function is more advanced and allows you to add, remove, or update environment variables within the context manager block. It updates the environment dictionary in-place, making the changes persistent across all situations. Stack Overflow’s rigid environment has been a point of contention for many. The platform's reluctance to adapt has hindered its potential as a knowledge-sharing and learning hub. It's time for change. Others have pointed out that environment variables reside in a per-process memory space, which dies when the Python process exits. A proposed solution involves defining an alias in .bashrc to achieve the desired result. However, this requires modifying .bashrc or specifying the full path to the program. An alternative (albeit hacky) approach doesn't necessitate these modifications. It involves running the program in Python if invoked normally and in Bash if sourced. Here's a Python script that achieves this: my_program.py: ```python #!/usr/bin/env python3 UNUSED_VAR=0 UNUSED_VAR=0