Python External Resource Management: WITH Statement

Table of Contents

[Python External Resource Management: WITH Statement 3](#_Toc120455488)

[WITH Statement 3](#_Toc120455489)

[External Resources 3](#_Toc120455490)

[Context Managers 3](#_Toc120455491)

# Python External Resource Management: WITH Statement

## WITH Statement

|  |  |
| --- | --- |
| WITH statement in Python is useful tool for properly managing external resources | allows you to take advantage of existing **context managers** to automatically handle the setup and teardown phases whenever you’re dealing with external resources or with operations that require those phases. |
| context management protocol allows you to create your own context managers so you can customize the way you deal with system resources | write more expressive code and avoid resource leaks in your programs. The with statement helps you implement some common resource management patterns |

## External Resources

|  |  |
| --- | --- |
| * [files](https://realpython.com/working-with-files-in-python/) – with open * [locks](https://en.wikipedia.org/wiki/Lock_(computer_science)) - * network connections | [memory leak](https://en.wikipedia.org/wiki/Memory_leak) – when available memory gets reduced every time you create and open a new instance of a given resource without closing an existing one. |
| * Setup phase – opening a file, creating a lock, opening a network connection * Usage Phase – reading/writing to a file, database or connection * Teardown phase – closing a file, releasing a lock, closing a network connection | Stale connections and locks can slow down a system. |

## Context Managers

|  |  |
| --- | --- |
| An object which controls the environment seen in a with statement by defining \_\_enter\_\_() and \_\_exit\_\_() methods | WITH statement does the following:   1. \_\_enter\_\_() is called by the with statement to enter the runtime context. 2. \_\_exit\_\_() is called when the execution leaves the with code block. |
| **Example 1:** VAR **=** EXPR  VAR**.**\_\_enter\_\_**()**  **try:**  BLOCK  **finally:**  VAR**.**\_\_exit\_\_**()** | WITH example:  **with** VAR **=** EXPR**:**  BLOCK |
| **Example 2:**  **@contextmanager**  **def** opening**(**filename**):**  f **=** open**(**filename**)**  **try:**  **yield** f  **finally:**  f**.**close**()** | WITH example2:  **with** f **=** opening**(**filename**):**  **...**read data **from** f... |

<https://peps.python.org/pep-0343/>

**context variable**

A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See [contextvars](https://docs.python.org/3/library/contextvars.html" \l "module-contextvars" \o "contextvars: Context Variables).

<https://docs.python.org/3/library/contextvars.html#module-contextvars>

# The with Statement Approach

|  |  |
| --- | --- |
| The Python with statement creates a runtime context that allows you to run a group of statements under the control of a context manager | PEP 343 added the with statement to make it possible to factor out standard use cases of the try … finally statement. |
| with expression as target\_var:  do\_something(target\_var) | WITH run steps:   1. Call expression to obtain a context manager. 2. Store the context manager’s .\_\_enter\_\_() and .\_\_exit\_\_() methods for later use. 3. Call .\_\_enter\_\_() on the context manager and bind its return value to target\_var if provided. 4. Execute the with code block. 5. Call .\_\_exit\_\_() on the context manager when the with code block finishes. |

## Multiple Context Managers

with A() as a, B() as b:

pass

This works like nested with statements but without nesting. This might be useful when you need to open two files at a time, the first for reading and the second for writing:

with open("input.txt") as in\_file, open("output.txt", "w") as out\_file:

# Read content from input.txt

# Transform the content

# Write the transformed content to output.txt

pass