Python External Resource Management: WITH Statement

Table of Contents

[Python External Resource Management 3](#_Toc120449157)

# 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 |  |
| **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 example:  **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>

# Tokens

Each line of Python code is made up of components called tokens. There are five types of tokens:

**Identifiers** - An identifier is a name used to identify a variable, function, class, module, or other object. An identifier starts with a letter (A to Z or a to z) or an underscore (\_) followed by zero or more letters, underscores, and digits (0 to 9). Case is significant in Python: lowercase and uppercase letters are distinct. Python does not allow punctuation characters such as @, $, and % within identifiers.

**Keywords** - Python has 30 keywords, which are identifiers that Python reserves for special syntactic uses. Keywords contain lowercase letters only. You cannot use keywords as regular identifiers. Some keywords begin simple statements or clauses of compound statements, while other keywords are operators.

* and
* del
* for
* is
* raise
* assert
* elif
* from
* lambda
* return
* break
* else
* global
* not
* try
* class
* except
* if
* or
* while
* continue
* exec
* import
* pass
* with (2.5)
* def
* finally
* in
* print
* yield

**Operators** - Python uses nonalphanumeric characters and character combinations as operators.

* + - addition
* - - subtraction
* \* - multiplication
* / - division
* % -
* \*\*
* //
* <<
* >>
* &
* |
* ^
* ~
* <
* <=
* >
* >=
* <>
* !=
* ==

**Delimiters -** Python uses the following symbols and symbol combinations as delimiters in expressions, lists, dictionaries, various aspects of statements, and strings,

* (
* )
* [
* ]
* {
* }
* ,
* :
* .
* `
* =
* ;
* += - augmented assignment operators
* -= - augmented assignment operators
* \*= - augmented assignment operators
* /= - augmented assignment operators
* //= - augmented assignment operators
* %= - augmented assignment operators
* &= - augmented assignment operators
* |= - augmented assignment operators
* ^= - augmented assignment operators
* >>= - augmented assignment operators
* <<= - augmented assignment operators
* \*\*= - augmented assignment operators
* ' - part of other tokens
* " - part of other tokens
* # - part of other tokens
* \ - part of other tokens

**Literals** - A literal is a number or string that appears directly in a program

* 42 # Integer literal
* 3.14 # Floating-point literal
* 1.0j # Imaginary literal
* 'hello' # String literal
* "world" # Another string literal
* """Good
* night""" # Triple-quoted string literal

**literals and delimiters** - you can create data values of some other fundamental types:

* [ 42, 3.14, 'hello' ] # List
* ( 100, 200, 300 ) # Tuple
* { 'x':42, 'y':3.14 } # Dictionary

**Statements -** You can consider a Python source file as a sequence of simple and compound statements. Unlike other languages, Python has no declarations or other top-level syntax elements, just statements.

* **Simple statements** - A simple statement is one that contains no other statements. A simple statement lies entirely within a logical line. As in other languages, you may place more than one simple statement on a single logical line, with a semicolon (;) as the separator.
  + **expression** can stand on its own as a simple statement
  + **assignment** is a simple statement that assigns values to variables
* **Compound statements** - A compound statement contains one or more other statements and controls their execution.
  + A compound statement has one or more clauses, aligned at the same indentation.
  + Each clause has a **header** starting with a keyword and ending with a colon (:), followed by a body, which is a sequence of one or more statements.
  + Block - When the body contains multiple statements. These statements should be placed on separate logical lines after the header line, indented four spaces rightward. The block lexically ends when the indentation returns to that of the clause header (or further left from there, to the indentation of some enclosing compound statement). Alternatively, the body can be a single simple statement, following the : on the same logical line as the header.