Python Context Managers

Summary: in this tutorial, you'll learn about the Python context managers and how to use them effectively

Introduction to Python context managers

A **context manager** is an **object** that defines a **runtime context** executing within the `with` statement.

Let's start with a simple example to understand the context manager concept.

Suppose that you have a file called `data.txt` that contains an integer (https://www.pythontutorial.net/advanced-python/python-integers/) `100` .

The following program reads (https://www.pythontutorial.net/python-basics/python-read-text-file/) the `data.txt` file, converts its contents to a number, and shows the result to the standard output:

```
f = open('data.txt')
data = f.readlines()

# convert the number to integer and display it
```

```
print(int(data[0]))

f.close()
```

The code is simple and straightforward.

However, the `data.txt` may contain data that cannot be converted to a number. In this case, the code will result in an exception.

For example, if the `data.txt` contains the string `'100'` instead of the number 100, you'll get the following error:

```
ValueError: invalid literal for int() with base 10: "'100'"
```

Because of this exception, Python may not close the file properly.

To fix this, you may use the `try...except...finally` (https://www.pythontutorial.net/python-basics/python-try-except-finally/) statement:

```
try:
    f = open('data.txt')
    data = f.readlines()
    # convert the number to integer and display it
    print(int(data[0]))
except ValueError as error:
    print(error)
finally:
    f.close()
```

Since the code in the `finally` block always executes, the code will always close the file properly.

This solution works as expected. However, it's quite verbose.

Therefore, Python provides you with a better way that allows you to automatically close the file after you complete processing it.

This is where **context managers** come into play.

The following shows how to use a context manager to process the `data.txt` file:

```
with open('data.txt') as f:
    data = f.readlines()
    print(int(data[0])
```

In this example, we use the `open()` function with the `with` statement. After the `with` block, Python will close automatically.

Python with statement

Here is the typical syntax of the `with` statement:

```
with context as ctx:
    # use the the object

# context is cleaned up
```

How it works.

- When Python encounters the `with` statement, it creates a new context. The context can optionally return an `object` .
- After the `with` block, Python cleans up the context automatically.
- The scope of the `ctx` has the same scope as the `with` statement. It means that you can access the `ctx` both inside and after the `with` statement.

The following shows how to access the `f` variable after the `with` statement:

```
with open('data.txt') as f:
    data = f.readlines()
    print(int(data[0]))
print(f.closed)  # True
```

Python context manager protocol

Python context managers work based on the **context manager protocol**.

The context manager protocol has the following methods:

- `__enter__()` – setup the context and optionally return some object
- `__exit__()` – cleanup the object.

If you want a class (https://www.pythontutorial.net/python-oop/python-class/) to support the context manager protocol, you need to implement these two methods.

Suppose that `ContextManager` is a class that supports the context manager protocol.

The following shows how to use the `ContextManager` class:

```
with ContextManager() as ctx:
    # do something
# done with the context
```

When you use `ContextManager` class with the `with` statement, Python implicitly creates an instance of the `ContextManager` class (`instance`) and automatically call `__enter__()` method on that instance.

The `__enter__()` method may optionally return an object. If so, Python assigns the returned object the `ctx` .

Notice that `ctx` references the object returned by the `__enter__()` method. It doesn't reference the instance of the `ContextManager` class.

If an exception occurs inside the with block or after the `with` block, Python calls the `__exit__()` method on the `instance` object.

Functionally, the `with` statement is equivalent to the following `try...finally` statement:

```
   instance = ContextManager()
ctx = instance.__enter__()

try:
    # do something with the txt
finally:
    # done with the context
    instance.__exit__()
```

The __enter__() method

In the `__enter__()` method, you can carry the necessary steps to setup the context.

Optionally, you can returns an object from the `__enter__()` method.

The __exit__() method

Python always executes the `__exit__()` method even if an exception occurs in the `with` block.

The `__exit__()` method accepts three arguments: exception type, exception value, and traceback object. All of these arguments will be `None` if no exception occurs.

```
def __exit__(self, ex_type, ex_value, ex_traceback):
    ...
```

The `__exit__()` method returns a boolean value, either `True` or `False` .

If the return value is True, Python will make any exception silent. Otherwise, it doesn't silence the exception.

Python context manager applications

As you see from the previous example, the common usage of a context manager is to open and close files automatically.

However, you can use context managers in many other cases:

1) Open – Close

If you want to open and close a resource automatically, you can use a context manager.

For example, you can open a socket and close it using a context manager.

2) Lock – release

Context managers can help you manage locks for objects more effectively. They allow you to acquire a lock and release it automatically.

3) Start – stop

Context managers also help you to work with a scenario that requires the start and stop phases.

For example, you can use a context manager to start a timer and stop it automatically.

3) Change – reset

Context managers can work with change and reset scenario.

For example, your application needs to connect to multiple data sources. And it has a default connection.

To connect to another data source:

- First, use a context manager to change the default connection to a new one.
- Second, work with the new connection
- Third, reset it back to the default connection once you complete working with the new connection.

Implementing Python context manager protocol

The following shows a simple implementation of the `open()` function using the context manager protocol:

```
class File:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    def __enter__(self):
        print(f'Opening the file {self.filename}.')
        self.__file = open(self.filename, self.mode)
        return self.__file

    def __exit__(self, exc_type, exc_value, exc_traceback):
        print(f'Closing the file {self.filename}.')
        if not self.__file.closed:
            self.__file.close()

        return False
```

```
with File('data.txt', 'r') as f:
    print(int(next(f)))
```

How it works.

- First, initialize the `filename` and `mode` in the `__init__()` method.
- Second, open the file in the `__enter__()` method and return the file object.
- Third, close the file if it's open in the `__exit__()` method.

Using Python context manager to implement the start and stop pattern

The following defines a `Timer` class that supports the context manager protocol:

```
from time import perf_counter


class Timer:
    def __init__(self):
        self.elapsed = 0

    def __enter__(self):
        self.start = perf_counter()
        return self

    def __exit__(self, exc_type, exc_value, exc_traceback):
        self.stop = perf_counter()
        self.elapsed = self.stop - self.start
        return False
```

How it works.

- First, import the `perf_counter` from the `time` module.
- Second, start the timer in the `__enter__()` method
- Third, stop the timer in the `__exit__()` method and return the elapsed time.

Now, you can use the `Timer` class to measure the time needed to calculate the Fibonacci of 1000, one million times:

```
def fibonacci(n):
    f1 = 1
    f2 = 1
    for i in range(n-1):
        f1, f2 = f2, f1 + f2

    return f1


with Timer() as timer:
    for _ in range(1, 1000000):
        fibonacci(1000)

print(timer.elapsed)
```

Summary

- Use Python context managers to define runtime contexts when executing in the `with` statement.
- implement the `__enter__()` and `__exit__()` methods to support the context manager protocol.