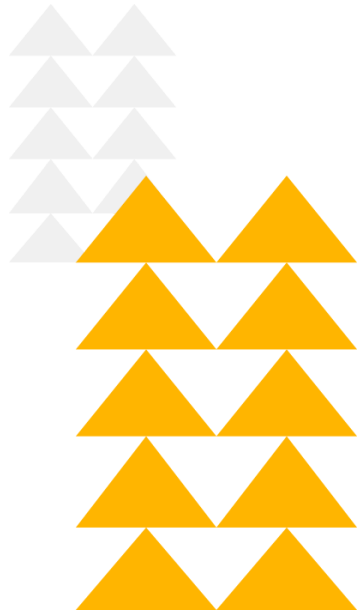
Зачем нужны менеджеры контекста?

Менеджеры контекста позволяют компактно выразить уже знакомый нам паттерн управления ресурсами:

```
r = acquire_resource()
try:
    do_something(r)
finally:
    release_resource(r)
```

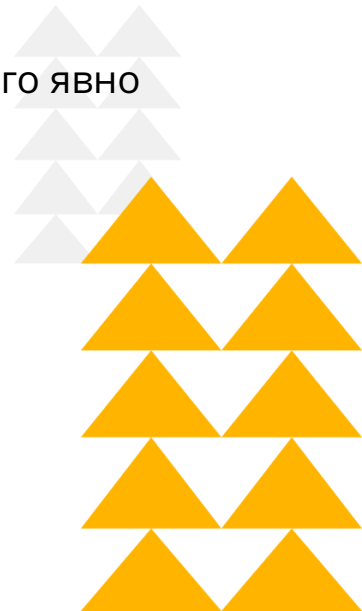


Зачем нужны менеджеры контекста? (2)

С помощью менеджера контекста пример выше можно записать так:

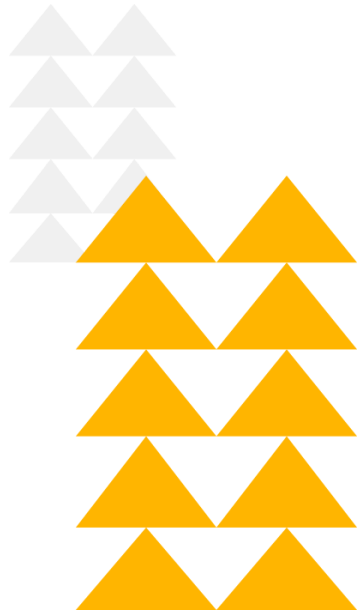
```
01. with acquire_resource() as r:
02.     do_something(r)
```

Действие `release_resource` будет выполнено автоматически, вызывать его явно не нужно.



Протокол менеджеров контекста

Метод `__enter__` инициализирует контекст, например, открывает файл или захватывает мьютекс. Значение, возвращаемое методом `__enter__`, записывается по имени, указанному после оператора `as`

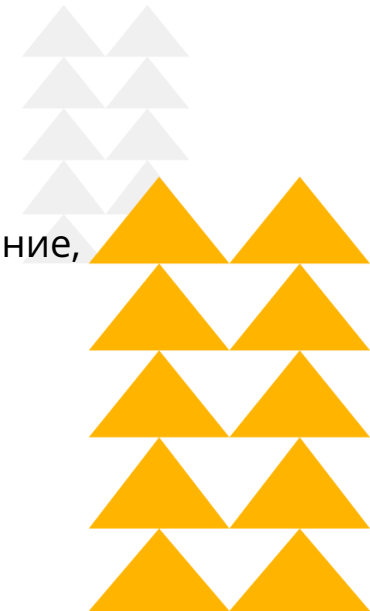


Протокол менеджеров контекста (2)

Метод `__exit__` вызывается после выполнения тела оператора `with`. Метод принимает три аргумента:

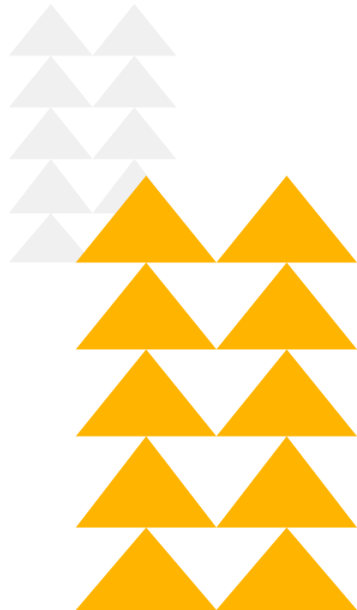
1. тип исключения,
2. само исключение и
3. объект типа `traceback`.

Если в процессе исполнения тела оператора `with` было поднятно исключение, метод `__exit__` может подавить его, вернув `True`.



Протокол менеджеров контекста (3)

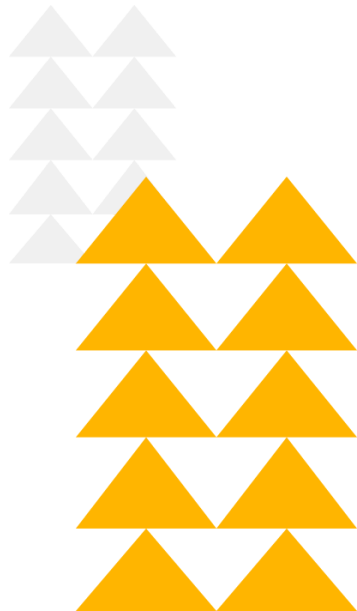
Экземпляр любого класса, реализующего эти два метода, является менеджером контекста.



Менеджер контекста

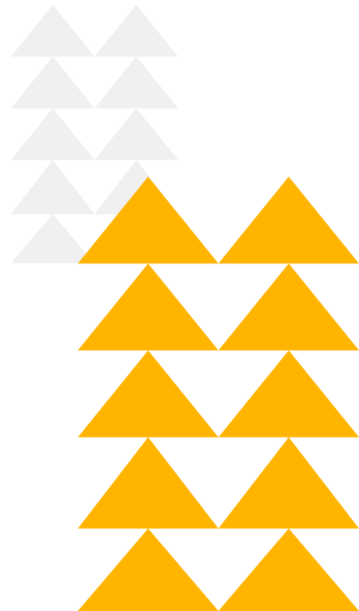
Процесс исполнение оператора `with` можно концептуально записать так:

```
01. >>> manager = acquire_resource()
02. >>> r = manager.__enter__()
03. >>> try:
04.         do_something(r)
05.     finally:
06.         exc_type, exc_value, tb = sys.exc_info()
07.         suppress = manager.__exit__(exc_type, exc_value, tb)
08.         if exc_value is not None and not suppress:
09.             raise exc_value
```



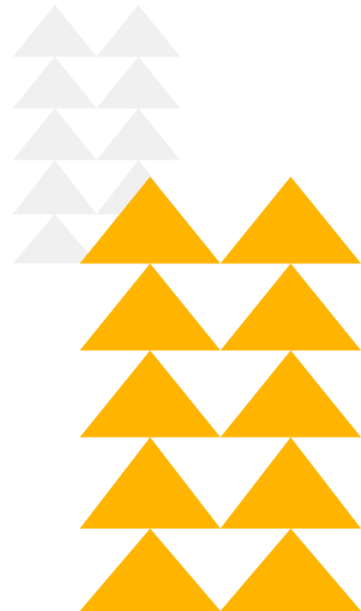
Пример: opened

```
01. class opened:
02.     def __init__(self, path, *args, **kwargs):
03.         self.opener = partial(open, path, *args, **kwargs)
04.
05.     def __enter__(self):
06.         self.fp = self.opener()
07.         return self.fp
08.
09.     def __exit__(self, *exc_info):
10.         self.fp.close()
11.
12. with opened("./example.txt", mode="rt") as fp:
13.     pass
```

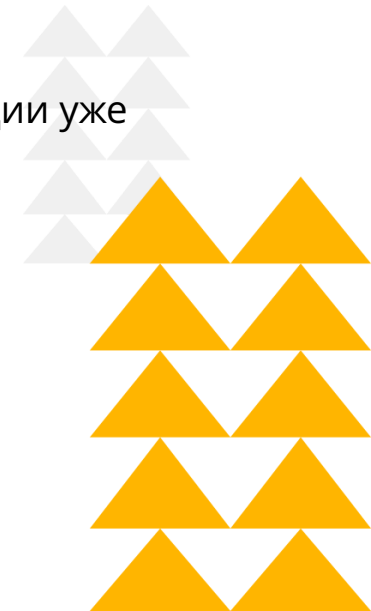
Пример: cd

```
01. >>> class cd:
02.         def __init__(self, path):
03.             self.path = path
04.         def __enter__(self):
05.             self.saved_cwd = os.getcwd()
06.             os.chdir(self.path)
07.         def __exit__(self, *exc_info):
08.             os.chdir(self.saved_cwd)
09. >>> print(os.getcwd())
10. '/Users/vital/Development/Py/py_course_2019'
11. >>> with cd("/tmp"):
12.         print(os.getcwd())
13. '/tmp'
```



Take away: менеджеры контекста

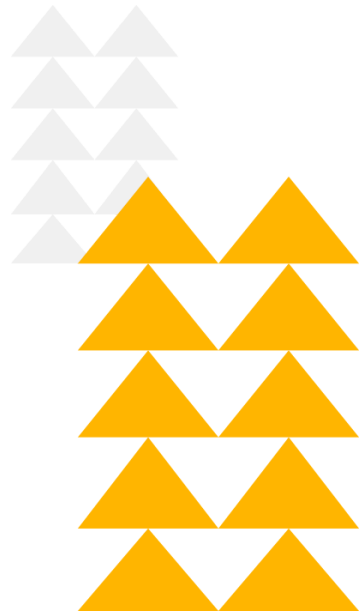
- Менеджеры контекста — удобный способ управлять жизненным циклом ресурсов в Python
- Для работы с менеджером контекста используется оператор `with`
- Менеджером контекста является экземпляр любого класса, реализующего методы `__enter__` и `__exit__`
- Некоторые встроенные типы, например, файлы и примитивы синхронизации уже поддерживают протокол менеджеров контекста — этим можно и нужно пользоваться при написании кода.



contextlib

Модуль `contextlib` содержит функции и классы, украшающие жизнь любителя менеджеров контекста:

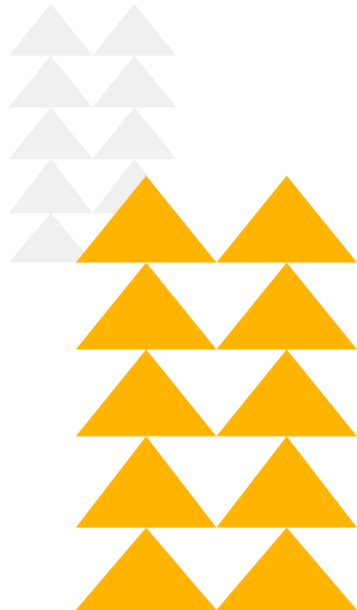
- `closing`,
- `suppress`,
- `ContextDecorator`
- и многое другое.



Модуль contextlib: closing

Менеджер контекста `closing` обобщает логику уже известного нам `opened` на экземпляр любого класса, реализующего метод `close`.

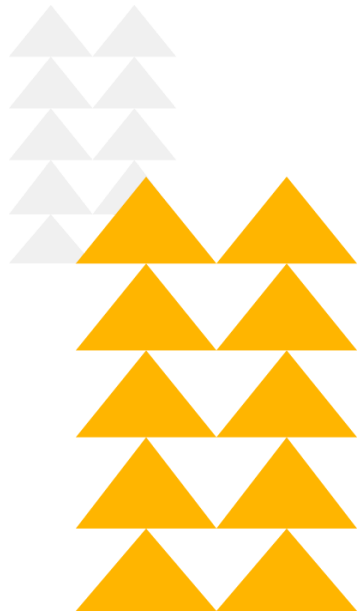
Реализовать `closing` самому несложно, но приятно, когда в стандартной библиотеке языка есть и такие мелочи.



Модуль contextlib: closing (2)

С помощью `closing` можно, например, безопасно работать с HTTP ресурсами:

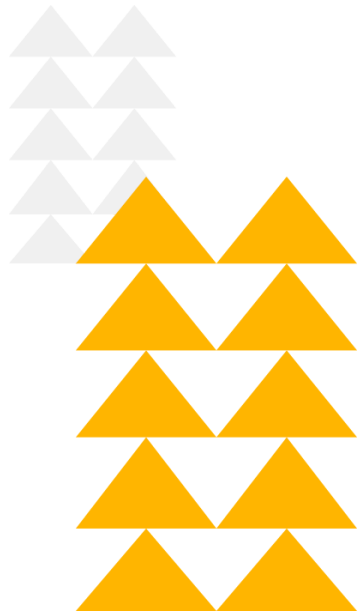
```
01. from contextlib import closing
02. from urllib.request import urlopen
03. url = "http://example.com"
04. with closing(urlopen(url)) as page:
05.     do_something(page)
```



Модуль contextlib: suppress

С помощью менеджера контекста `suppress` можно локального подавить исключения указанных типов:

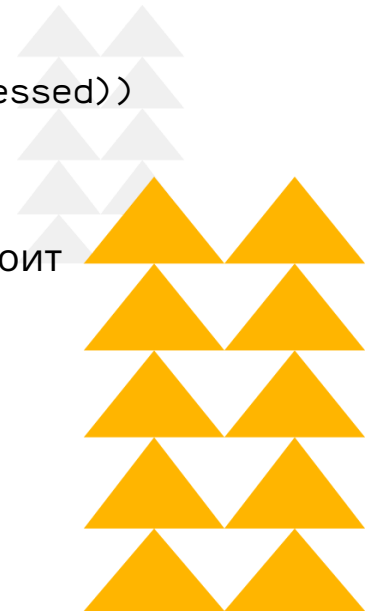
```
01. from contextlib import suppress
02. with suppress(FileNotFoundError):
03.     os.remove("example.txt")
```



Модуль contextlib: suppress (2)

```
01. class supress:
02.     def __init__(self, *suppressed):
03.         self.suppressed = suppressed
04.     def __enter__(self):
05.         pass
06.     def __exit__(self, exc_type, exc_value, tb):
07.         return (exc_type is not None and issubclass(exc_type, suppressed))
```

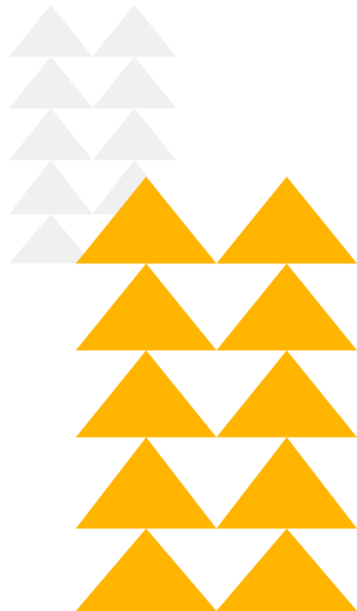
При использовании suppress, как и в целом при работе с исключениями, стоит указывать ниболее специфичный тип исключения.



Модуль contextlib: ContextDecorator

Для того, чтобы менеджер контекста можно было использовать как декоратор, достаточно унаследовать его от `ContextDecorator`.

```
01. from contextlib import suppress, ContextDecorator
02. class suppressed(suppress, ContextDecorator):
03.     pass
04.
05. @suppressed(IOError)
06. def do_something():
07.     pass
```



Спасибо за внимание

Виталий Кудёлка

Разработчик программного обеспечения

vital.kudzelka@leverx.com

github.com/vitalk