교수의 파이썬

04_2. with 문과 컨텍스트 매니저

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 파이썬

04_2. with 문과 컨텍스트 매니저

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 고급 파이썬

04_2. with 문과 컨텍스트 매니저

창원대학교 정보통신공학과 교수 박동규

널널한 교수의 고급 파이썬

04_2. with 문과 컨텍스트 매니저

창원대학교 정보통신공학과 교수 박동규

try-except-else-finally 문

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭 되는 부분으로 넘겨준다.

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭되는 부분으로 넘겨준다.
except	try절에서 오류가 발생했을 때 처리할 내용을 담는다.

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭되는 부분으로 넘겨준다.
except	try절에서 오류가 발생했을 때 처리할 내용을 담는다.
else	try절에서 else는 에러가 발생하지 않을때 실행하게 되는 블럭

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭되는 부분으로 넘겨준다.
except	try절에서 오류가 발생했을 때 처리할 내용을 담는다.
else	try절에서 else는 에러가 발생하지 않을때 실행하게 되는 블럭
finally	finally는 예외의 발생 여부와 상관 없이 항상 실행되는 블럭

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭 되는 부분으로 넘겨준다.
except	try절에서 오류가 발생했을 때 처리할 내용을 담는다.
else	try절에서 else는 에러가 발생하지 않을때 실행하게 되는 블럭
finally	finally는 예외의 발생 여부와 상관 없이 항상 실행되는 블럭

try-except-else-finally 문

파이썬의 try - except 문은 에러가 발생할 수 있는 예
외적인 상황을 유연하게 대처할 수 있는 방법이다

try	먼저 try절이 실행되어 예외가 발생하지 않으면 except를 건너뛰는데, 예외가 발생하면 오류를 확인하며 except의 매칭 되는 부분으로 넘겨준다.
except	try절에서 오류가 발생했을 때 처리할 내용을 담는다.
else	try절에서 else는 에러가 발생하지 않을때 실행하게 되는 블럭
finally	finally는 예외의 발생 여부와 상관 없이 항상 실행되는 블럭

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

예외가 발생하던 안하던 항상 수행

divide_ex.py

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

예외가 발생하던 안하던 항상 수행

divide_ex.py

결과

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

예외가 발생하던 안하던 항상 수행

divide_ex.py

결과

```
/Users/dongupak/miniconda3/bin/pyt
divide(100,2) 함수호출 :
결과 : 50.0
수행완료
divide(100,0) 함수호출 :
0으로 나누는 오류발생
수행완료

Process finished with exit code 0
```

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

예외가 발생하던 안하던 항상 수행

divide_ex.py

결과

```
/Users/dongupak/miniconda3/bin/pyt
divide(100,2) 함수호출 :
결과 : 50.0
수행완료
divide(100,0) 함수호출 :
0으로 나누는 오류발생
수행완료

Process finished with exit code 0
```

```python
def divide(x,y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("0으로 나누는 오류발생")
    else:
        print("결과 :", result)
    finally:
        print("수행완료")

print('divide(100,2) 함수호출 :')
divide(100,2)
print('divide(100,0) 함수호출 :')
divide(100,0)
```

ZeroDivisionError 예외가 발생하면 except 절에서 처리

예외가 발생하지 않을 경우 결과 출력

예외가 발생하던 안하던 항상 수행

divide_ex.py

결과

```
/Users/dongupak/miniconda3/bin/pyt
divide(100,2) 함수호출 :
결과 : 50.0
수행완료
divide(100,0) 함수호출 :
0으로 나누는 오류발생
수행완료

Process finished with exit code 0
```

try-else 문

try-else 문

- try 문에는 else 절을 사용할 수 있다. else 절은 예외가 발생하지 않은 경우에 실행되며 반드시 except 절 다음에 위치해야 한다

try-else 문

- try 문에는 else 절을 사용할 수 있다. else 절은 예외가 발생하지 않은 경우에 실행되며 반드시 except 절 다음에 위치해야 한다

- 이 기능을 이용하여 파일을 열어보도록 하자

try-else 문

- try 문에는 else 절을 사용할 수 있다. else 절은 예외가 발생하지 않은 경우에 실행되며 반드시 except 절 다음에 위치해야 한다

- 이 기능을 이용하여 파일을 열어보도록 하자

```
try:
    f = open('foo.txt', 'r')
except FileNotFoundError as e:
    print(str(e))
else:
    data = f.read()
    f.close()
```

try-else 문

- try 문에는 else 절을 사용할 수 있다. else 절은 예외가 발생하지 않은 경우에 실행되며 반드시 except 절 다음에 위치해야 한다

- 이 기능을 이용하여 파일을 열어보도록 하자

```
1       try:
2           f = open('foo.txt', 'r')
3       except FileNotFoundError as e:
4           print(str(e))
5       else:
6           data = f.read()
7           f.close()
```

open() 함수를 이용하여 foo.txt 라는 이름의 파일을 찾아서 가져오는 기능. 이때 만약 foo.txt라는 파일이 없다면 except 절이 수행될 것이고, foo.txt 파일이 있다면 else 절이 수행될 것이다. 파일을 읽어와서 data에 저장하는 기능이 있음

try - finally

- try 문에는 finally 절을 사용할 수 있다. finally 절은 try문 수행도중 발생하는 예외에 관계없이 항상 수행된다.

- 일반적으로 리소스를 오픈한 후 이를 close() 하는 경우에 많이 사용된다.

try - finally

- try 문에는 finally 절을 사용할 수 있다. finally 절은 try문 수행도중 발생하는 예외에 관계없이 항상 수행된다.

- 일반적으로 리소스를 오픈한 후 이를 close() 하는 경우에 많이 사용된다.

```
try:
    f = open("file.txt", "w")
    try:
        f.write('Hello World!')
    finally:
        f.close()
except IOError:
    print('oops!')
```

try - finally

- try 문에는 finally 절을 사용할 수 있다. finally 절은 try문 수행도중 발생하는 예외에 관계없이 항상 수행된다.

- 일반적으로 리소스를 오픈한 후 이를 close() 하는 경우에 많이 사용된다.

```
try:
    f = open("file.txt", "w")
    try:
        f.write('Hello World!')
    finally:
        f.close()
except IOError:
    print('oops!')
```

많은 try-except, try-finally 문으로 인해 코드 가독성이 떨어진다. f.close()를 실행하지 않아도 대부분 잘 동작한다

with 문

- try/finally 구문을 더 간편하게 사용할 수 있다
- context manager에 의해 실행되는 __enter__(), __exit__() 을 정의하여 with 구문 body 의 앞부분과 뒷부분 실행 코드를 대신할 수 있다.

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

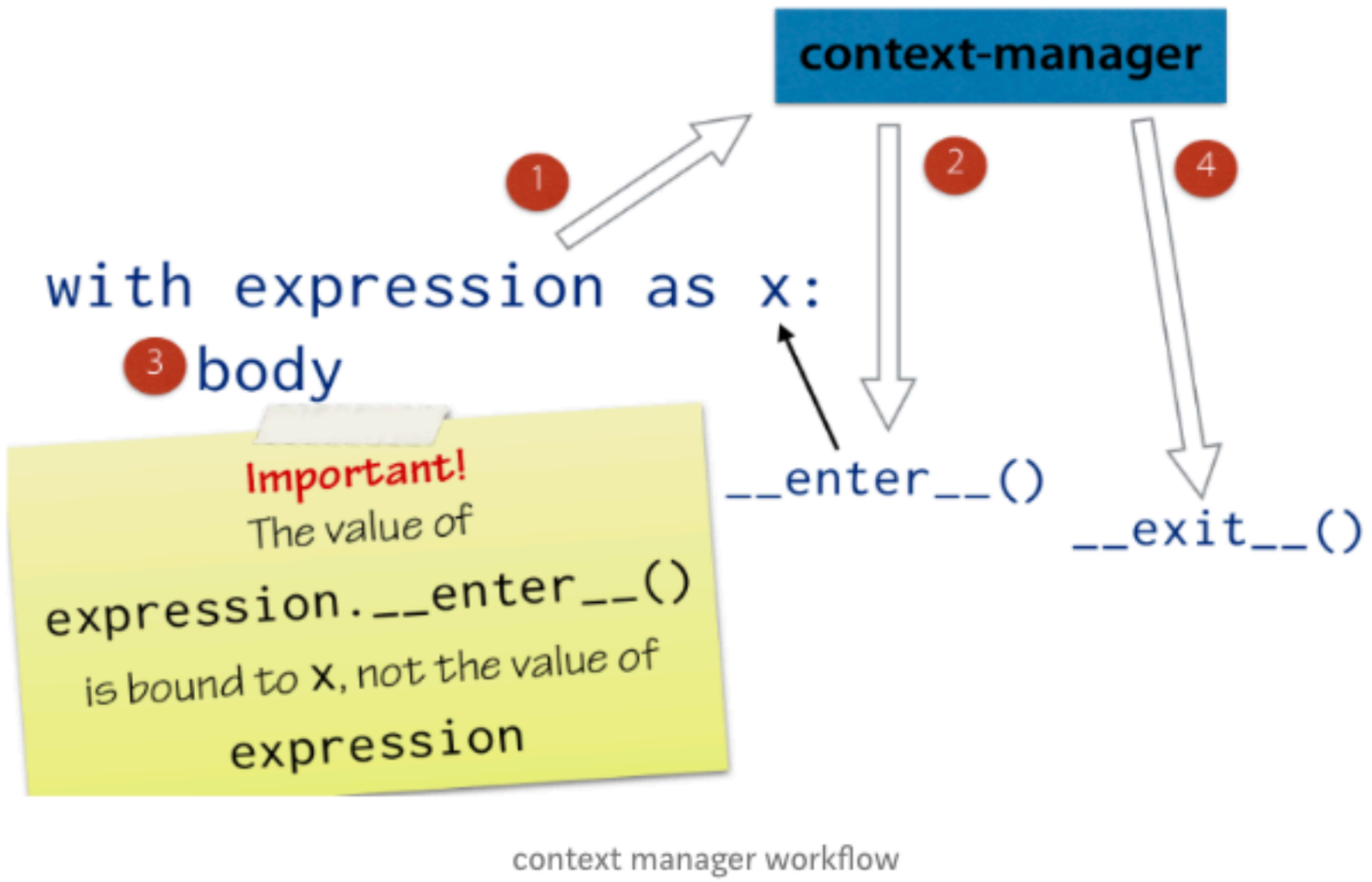
```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

```
set things up
try:
    do something
finally:
    tear things down
```

```
class controlled_execution:
    def __enter__(self):
        set things up
        return thing
    def __exit__(self, type, value, traceback):
        tear things down

with controlled_execution() as thing:
    some code using thing
```

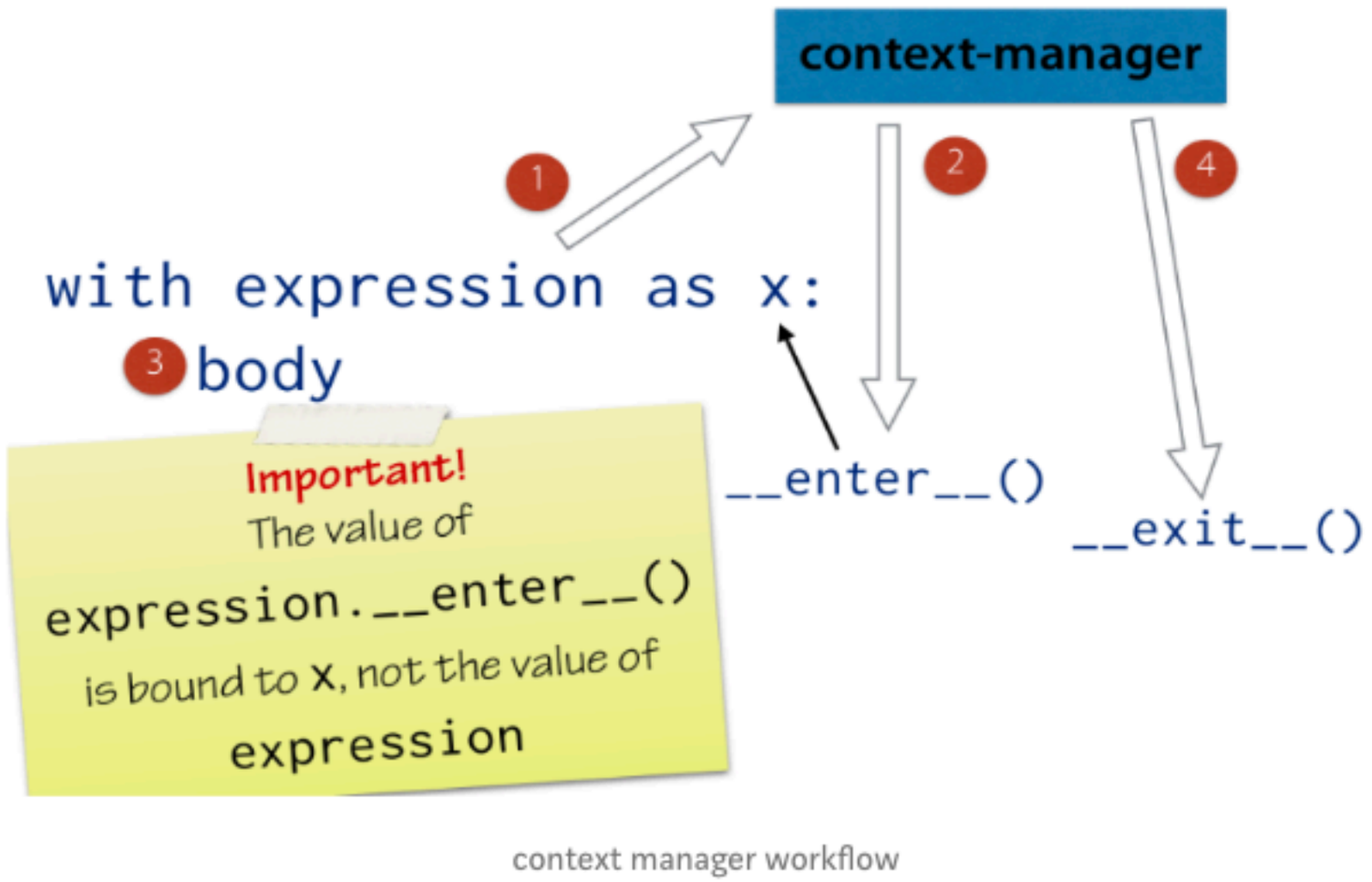
컨텍스트 매니저



context manager workflow

__enter__()

- called before entering the with-statement body
- return value bound to a variable
- can return value of any type

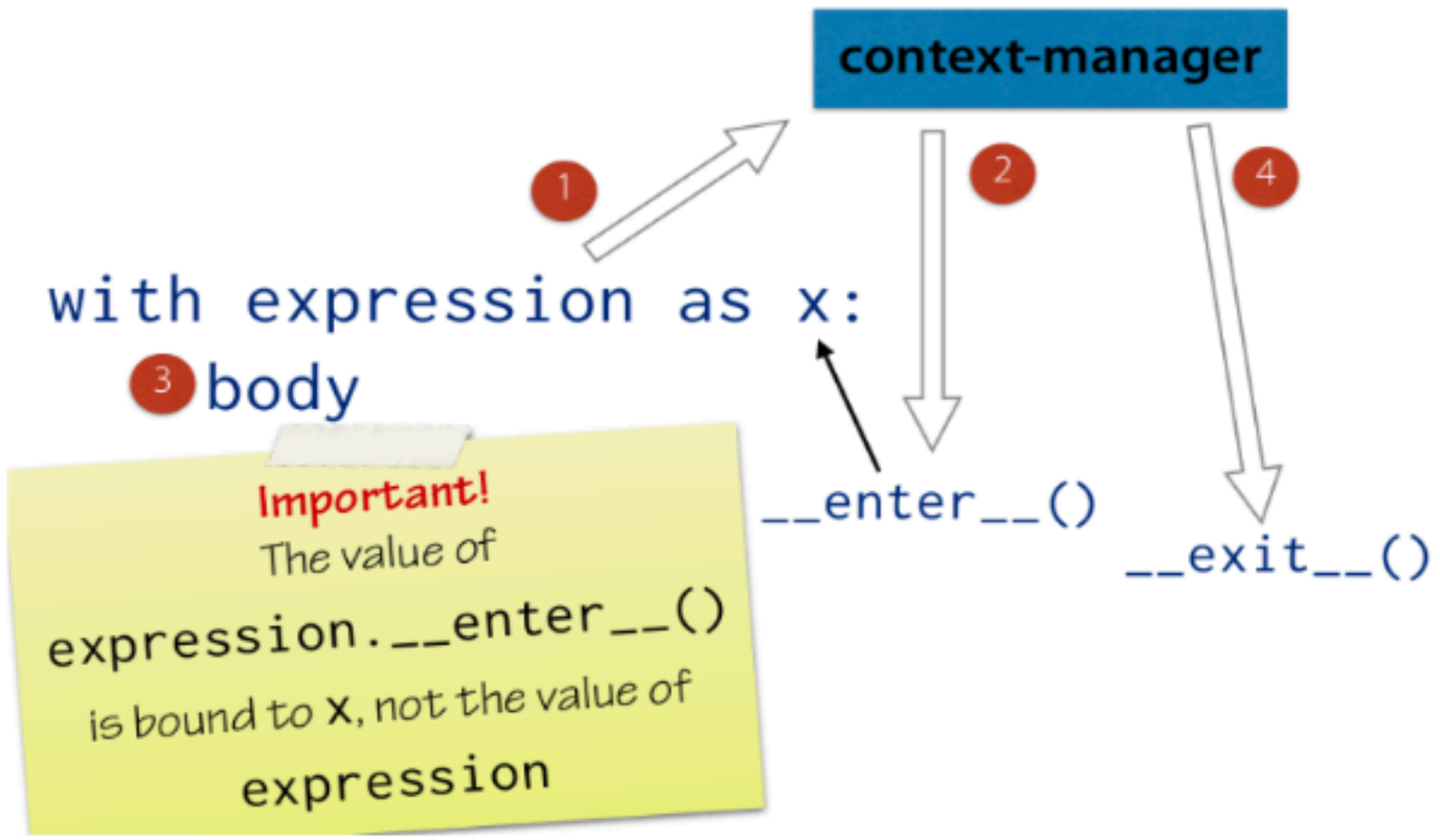


context manager workflow

__enter__()

- called before entering the with-statement body
- return value bound to a variable
- can return value of any type

https://towardsdatascience.com/10-topics-python-intermediate-programmer-should-know-3c865e8533d6



context manager workflow

with 문에서 사용하도록 설계된 객체를 말함

__enter__()

- called before entering the with-statement body
- return value bound to a variable
- can return value of any type

https://towardsdatascience.com/10-topics-python-intermediate-programmer-should-know-3c865e8533d6

with 구문

- with 구문이 실행되면 context manager가 자동적으로
  - __enter__() 메소드를 실행한다
    - 이 메소드가 반환하는 값이 as의 thing으로 지정된다
    - 그후 some code using thing 에 해당하는 body 코드가 실행된다
- 예외적인 상황이 생겨도 __exit__() 메소드는 호출이 보장된다

사례

- 파이썬의 file 객체는 __enter__(), __exit__() 메소드가 구현되어 있다.

- 이 객체는 file object 자신을 반환한다. __exit__() 메소드는 당연히 파일을 close한다

Lab

```
>>> import sys
>>> f = open("x.txt")   # x.txt 를 연다
>>> f
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.__enter__()
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.read(1)
'1'
>>> f.__exit__()
>>> f.read(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

Lab

```
>>> import sys
>>> f = open("x.txt")   # x.txt 를 연다
>>> f
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.__enter__()
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.read(1)
'1'
>>> f.__exit__()      # f.__exit()__()에 의해 f.close() 가 자동으로 실행됨
>>> f.read(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

Lab

```
>>> import sys
>>> f = open("x.txt")          # x.txt를 연다
>>> f
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.__enter__()
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.read(1)
'1'
>>> f.__exit__()          # f.__exit()__()에 의해 f.close() 가 자동으로 실행됨
>>> f.read(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

Lab

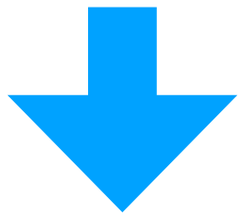
```
>>> import sys
>>> f = open("x.txt")          # x.txt를 연다
>>> f
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.__enter__()
<_io.TextIOWrapper name='x.txt' mode='r' encoding='UTF-8'>
>>> f.read(1)
'1'
>>> f.__exit__()               # f.__exit()__()에 의해 f.close() 가 자동으로 실행됨
>>> f.read(1)                  # 따라서 f.read() 메소드는 오류를 출력
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
>>>
```

Lab

```
>>> with open("x.txt") as f:
...     data = f.read()
...     print(data)
...
1
2
3
4
5

>>>
```

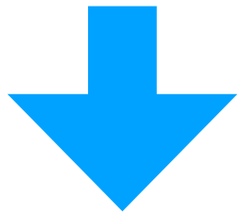
with 구문을 사용함으로써
open()이 간단해지고, finally도 필요없어졌다

```
try:
     f = open("file.txt", "w")
     try:
          f.write('Hello World!')
     finally:
          f.close()
except IOError:
     print('oops!')
```



```
try:
     with open("file.txt", "w") as outfile:
          outfile.write('Hello World!')
except IOError:
     print('oops!')
```

```python
try:
    f = open("file.txt", "w")
    try:
        f.write('Hello World!')
    finally:
        f.close()
except IOError:
    print('oops!')
```



```python
# 좀 더 간결하고 효율적인 코드
try:
    with open("file.txt", "w") as outfile:
        outfile.write('Hello World!')
except IOError:
    print('oops!')
```

with 구문의 장점

- with 구문을 사용하면 컨텍스트 매니저가 자동적으로 해야할 일을 수행하고 구문을 빠져나오면 반드시 해야할 일을 자동적으로 수행한다.
- 따라서 객체의 재사용성이 높아진다
- 코드가 간결해진다

감사합니다