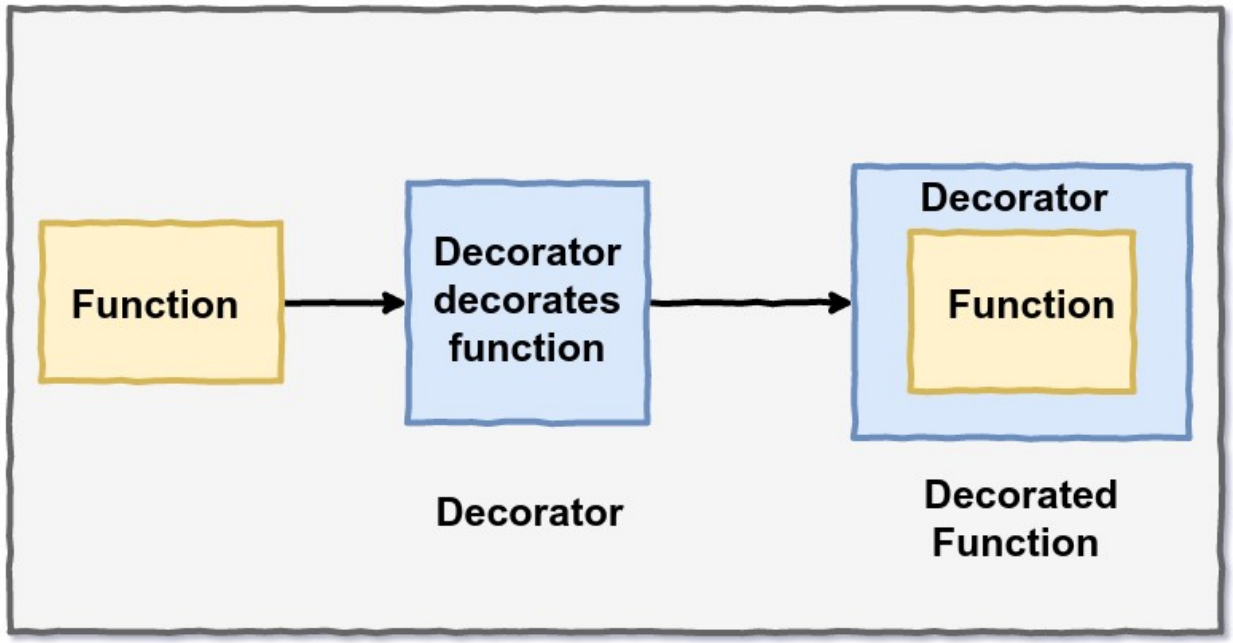
Декораторы

Декоратор – это обертка функции, которая позволяет изменить её поведение, не изменяя её код

Декоратор представляет собой функцию, которая получает на вход декорируемую функцию и возвращает декорированную функцию



Декораторы. Примеры

```
def my_decorator(func):
    def my_wrapper(a, b):
        print('Function', func.__name__, 'start')
        res = func(a, b)
        print('Result:', res)

    return my_wrapper

def add2(x, y):
    return x+y

add_decored = my_decorator(add2) # function
add_decored(2,5)
# Function add2 start
# Result: 7
```

Оператор декорирования функций

```
# assign to add2 decorator my_decorator
add2 = my_decorator(add2)
add2(2,5)
# Function add2 start
# Result: 7
```

Выражение `add2 = my_decorator(add2)` эквивалентно объявлению имени декоратора со знаком @ перед объявлением декорируемой функции

```
# another way to assign decorator my_decorator to add2
@my_decorator
def add2(a,b):
    return a+b

add2(2,5)
```

Декораторы. Примеры

```
def bold(func):
    def wrapper(who):
        print ("<b>")
        func(who)
        print ("</b>")
    return wrapper

def italic(func):
    def wrapper(who):
        print ("<i>")
        func(who)
        print ("</i>")
    return wrapper
```

Декораторы. Примеры

К функции может быть применено несколько декораторов

```
@italic
@bold
def hello(who):
    print ("Hello", who)

hello("World")
# <i>
# <b>
# Hello World
# </b>
# </i>
```

Декораторы. Примеры

```
import time
def measure_time(func):
    def wrapper(*arg):
        t = time.time()
        res = func(*arg)
        print(f"Took {time.time()-t} seconds to run")
        return res
    return wrapper

@measure_time
def func(n):
    sorted(range(n,0,-1))

func(10000)
# Took 0.002993345260620117 seconds to run
```

Ввод данных и обработка исключений

Функция **input** считывает строку, введенную пользователем

```
s = input('Enter string: ')
x = int(input('Enter integer: '))
int('abc') # error
int(12.5) # error
x = float(input('Enter float: '))
```

Для обработки исключений используется конструкция **try–except**

```
try:
    x = int(input('Enter integer: '))
except:
    print('Not integer entered')
```

try–except–else–finally

```
try:
    x = int(input('Enter integer: '))
    y = 1/x
except ValueError:
    print('Not integer entered')
except ZeroDivisionError:
    print('Division by zero')
except Exception:
    print('?')
else:
    # executes if no exception was raised
    print(y)
finally:
   # executes always
    print('Done!')
```

Блок finally

```
# without finally
try:
    run_code1()
except TypeError:
    run_code2()
other_code()
```

```
# with finally
try:
    run_code1()
except TypeError:
    run_code2()
finally:
    other_code()
```

Блок finally

```
try:
    x = 0
    y = 1/x # raises ZeroDivisionError,
#           which will be propagated to caller
finally:
    print('Bye!') # will be executed before
#                 ZeroDivisionError is propagated
```

```
try:
    run_code1()
except TypeError:
    run_code2()
    return None # The finally block is run before this
finally:
    other_code()
```

Оператор with

Конструкция `with–as` используется для оборачивания выполнения блока инструкций менеджером контекста

Менеджер контекста вызывает функции при входе и при выходе из контекста (в том числе, при возникновении исключения)

```
with A() as a:
    do_something()

# it is equivalent to:
# A.__enter__()
# do_something()
# A.__exit__()
```

Менеджер контекста используется, как правило, для инкапсуляции процессов инициализации и завершения

Создание менеджера контекста

```
class HelloContextManager:
    def __enter__(self):
        print("Entering the context...")
        return "Hello, World!"
    def __exit__(self, exc_type, exc_value, exc_tb):
        print("Leaving the context...")
        print(exc_type, exc_value, exc_tb)

with HelloContextManager() as hello:
    print(hello)
# Entering the context...
# Hello, World!
# Leaving the context...
# None None None
```

Параметры exc_type, exc_value, exc_tb метода __exit__() содержат информацию о возникшем исключении (None в случае успешного выхода из контекста)

Менеджер контекста для обработки исключений

with...as иногда более удобная конструкция, чем **try...except...finally**

```
class A:
    def __init__(self):
        self.x = 0
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_value, exc_tb):
        if exc_type is ZeroDivisionError:
            return True # do not raise exception
        else:
            return False

with A() as a:
   print(f"inv={1/a.x}") # nothing will happen
```

Если метод __exit__() возвращает True, исключение не выбрасывается. В противном случае исключение будет выброшено

Декоратор @contextmanager

Менеджер контекста может быть создан с помощью декоратора @contextmanager, без необходимости создания класса

```
from contextlib import contextmanager

@contextmanager
def hello_context_manager():
    print("Entering the context...")
    yield "Hello, World!"
    print("Leaving the context...")

with hello_context_manager() as hello:
    print(hello)
# Entering the context...
# Hello, World!
# Leaving the context...
```

Monkey Patch

Monkey patch – подмена отдельных методов, атрибутов классов или функций с целью тестирования, создания заглушек или изменения функционала внешних библиотек

```
from contextlib import contextmanager
from time import time

@contextmanager
def mock_time():
    global time
    saved_time = time
    time = lambda: 42
    yield
    time = saved_time

with mock_time():
    print(f"Mocked time: {time()}") # 42
time() # 1652701904.2153995
```

Использование менеджера контекста для работы с файлами

Файл будет закрыт вне зависимости от того, что введёт пользователь

```
with open('newfile.txt','w',encoding='utf-8') as f:
    d = int(input())
    print('1 / {} = {}'.format(d,1/d), file=f)
```

Использование **try...except** вместе с with...as:

```
try:
    with open('example.txt','r') as file:
        contents = file.read()
        print(contents)
except:
    print ("Error opening file")
```

Множественные менеджеры контекста

Python поддерживает использование множественных контекстов

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

# rewrite lines in reverse order
with (
    open("input.txt") as in_file,
    open("output.txt", "w") as out_file
):
    for line in in_file:
        out_file.write(line[::-1])
```

Множественные менеджеры контекста, фактически, представляют собой вложенные менеджеры контекстов, помещенные в стек