Decorators

- @

www.python.org/dev/peps/pep-0318/

Course overview

Basic programming
Advanced / specific python
Libraries & applications

1. Introduction to Python	10. Functions as objects	19. Linear programming
2. Python basics / if	11. Object oriented programming	20. Generators, iterators, with
3. Basic operations	12. Class hierarchies	21. Modules and packages
4. Lists / while / for	13. Exceptions and files	22. Working with text
5. Tuples / comprehensions	14. Doc, testing, debugging	23. Relational data
6. Dictionaries and sets	15. Decorators	24. Clustering
7. Functions	16. Dynamic programming	25. Graphical user interfaces (GUI)
8. Recursion	17. Visualization and optimization	26. Java vs Python
9. Recursion and Iteration	18. Multi-dimensional data	27. Final lecture

10 handins
1 final project (last 1 month)

Python decorators are just syntatic sugar

Python

```python
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

≡

Python

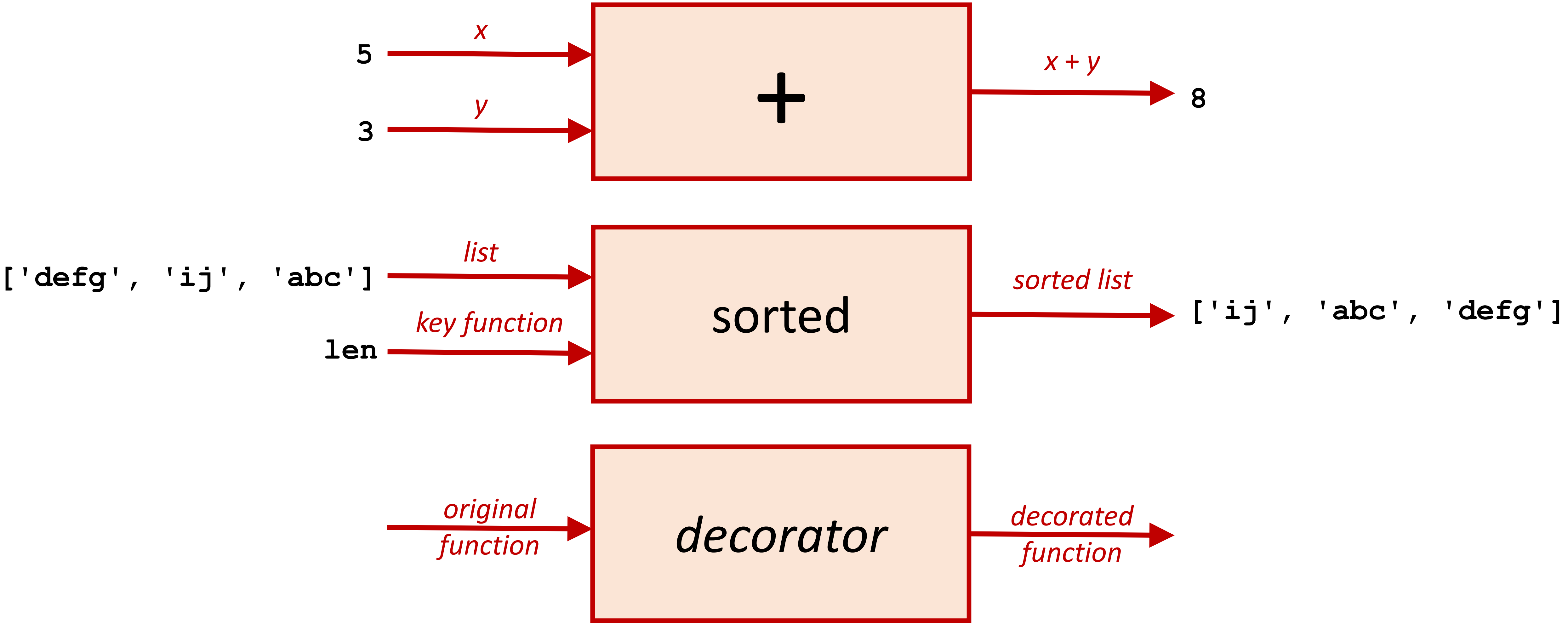
```python
def func(arg1, arg2, ...):
    pass

func = dec2(dec1(func))
```

'pie-decorator' syntax

dec1, dec2, ... are functions (decorators) taking a *function as an argument* and *returning a new function*

Note: decorators are listed bottom up in order of execution

Recap functions



Contrived example : Plus one (I-II)

plus_one1.py

```
def plus_one(x):
    return x + 1

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

print(plus_one(square(5)))
print(plus_one(cube(5)))
```

Python shell

```
| 26
| 126
```

Assume we *always* need to call `plus_one` on the result of `square` and `cube` (don't ask why!)

plus_one2.py

```
def plus_one(x):
    return x + 1

def square(x):
    return plus_one(x ** 2)

def cube(x):
    return plus_one(x ** 3)

print(square(5))
print(cube(5))
```

Python shell

```
| 26
| 126
```

We could call `plus_one` inside functions (but could be more `return` statements in functions)

Contrived example : Plus one (III-IV)

plus_one3.py

```
def plus_one(x):
    return x + 1

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

square_original = square
cube_original = cube

square = lambda x: plus_one(square_original(x))
cube = lambda x: plus_one(cube_original(x))

print(square(5))
print(cube(5))
```

Python shell

```
| 26
| 126
```

Overwrite `square` and `cube` with decorated versions

plus_one4.py

```
def plus_one(x):
    return x + 1

def plus_one_decorator(f):
    return lambda x: plus_one(f(x))

def square(x):
    return x ** 2

def cube(x):
    return x ** 3

square = plus_one_decorator(square)
cube = plus_one_decorator(cube)

print(square(5))
print(cube(5))
```

Python shell

```
| 26
| 126
```

Create a decorator function `plus_one_decorator`

Contrived example : Plus one (V-VI)

plus_one5.py

```
def plus_one(x):
    return x + 1

def plus_one_decorator(f):
    return lambda x: plus_one(f(x))

@plus_one_decorator
def square(x):
    return x ** 2

@plus_one_decorator
def cube(x):
    return x ** 3

print(square(5))
print(cube(5))
```

Python shell

```
| 26
| 126
```

Use Python decorator syntax

plus_one6.py

```
def plus_one_decorator(f):
    def plus_one(x):
        return f(x) + 1

    return plus_one

@plus_one_decorator
def square(x):
    return x ** 2

@plus_one_decorator
def cube(x):
    return x ** 3

print(square(5))
print(cube(5))
```

Python shell

```
| 26
| 126
```

Create local function instead of using `lambda`

Contrived example : Plus one (VII)

plus_one7.py

```
def plus_one_decorator(f):
    def plus_one(x):
        return f(x) + 1
    return plus_one

@plus_one_decorator
@plus_one_decorator
def square(x):
    return x ** 2

@plus_one_decorator
@plus_one_decorator
@plus_one_decorator
def cube(x):
    return x ** 3

print(square(5))
print(cube(5))
```

Python shell

```
| 27
| 128
```

- A function can have an arbitrary number of decorators (also the same repeated)
- Decorators are listed bottom up in order of execution

Handling arguments

run_twice1.py

```
def run_twice(f):
    def wrapper():
        f()
        f()
    return wrapper
@run_twice
def hello_world():
    print('Hello world')
hello_world()
```

Python shell

```
| Hello world
| Hello world
```

"`wrapper`" is a common name for the function returned by a decorator

run_twice2.py

```
def run_twice(f):
    def wrapper(*args):
        f(*args)
        f(*args)
    return wrapper
@run_twice
def hello_world():
    print('Hello world')
@run_twice
def hello(txt):
    print('Hello', txt)
hello_world()
hello('Mars')
```

Python shell

```
| Hello world
| Hello world
| Hello Mars
| Hello Mars
```

`args` holds the arguments in a tuple given to the function to be decorated

Question – What does the decorated program print ?

decorator_quiz.py

```
def double(f):
    def wrapper(*args):
        return 2 * f(*args)
    return wrapper

def add_three(f):
    def wrapper(*args):
        return 3 + f(*args)
    return wrapper

@double
@add_three
def seven():
    return 7

print(seven())
```

- 7
- 10
- 14
- 17
- 20
- Don’t know

Example: Enforcing argument types

- Defining decorators can be (slightly) complicated
- Using decorators is easy

integer_sum1.py

```
def integer_sum(*args):
    assert all([isinstance(x, int) for x in args]),\
           'all arguments most be int'
    return sum(args)
```

Python shell

```
> integer_sum(1, 2, 3, 4)
| 10
> integer_sum(1, 2, 3.2, 4)
| AssertionError: all arguments most be int
```

integer_sum2.py

```
def enforce_integer(f):    # decorator function
    def wrapper(*args):
        assert all([isinstance(x, int) for x in args]),\
               'all arguments most be int'
        return f(*args)
    return wrapper

@enforce_integer
def integer_sum(*args):
    return sum(args)
```

Python shell

```
> integer_sum(1, 2, 3, 4)
| 10
> integer_sum(1, 2, 3.2, 4)
| AssertionError: all arguments most be int
```

Decorators can take arguments

```python
@dec(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

≡

```python
def func(arg1, arg2, ...):
    pass
func = dec(argA, argB, ...)(func)
```

`dec` is a function (decorator) that takes a *list of arguments* and *returns a function* (to decorate `func`) that takes a *function as an argument* and *returns a new function*

Example: Generic type enforcing

print_repeated.py

```
def enforce_types(*decorator_args):
    def decorator(f):
        def wrapper(*args):
            assert len(args) == len(decorator_args), \
                   f'got {len(args)} arguments, expected {len(decorator_args)}'
            assert all([isinstance(x, t) for x, t in zip(args, decorator_args)]), \
                   'unexpected types'
            return f(*args)
        return wrapper
    return decorator

@enforce_types(str, int)  # decorator with arguments
def print_repeated(txt, n):
    print(txt * n)

print_repeated('Hello ', 3)
print_repeated('Hello ', 'world')
```

Python

```
@dec(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

≡

Python

```
def func(arg1, arg2, ...):
    pass
func = dec(argA, argB, ...)(func)
```

Python shell

```
| Hello Hello Hello
| AssertionError: unexpected types
```

Example: A timer decorator

time_it.py

```
import time

def time_it(f):
    def wrapper(*args, **kwargs):
        t_start = time.time()
        result = f(*args, **kwargs)
        t_end = time.time()
        t = t_end - t_start
        print(f'{f.__name__} took {t:.2f} seconds')
        return result

    return wrapper

@time_it
def slow_function(n):
    sum_ = 0
    for x in range(n):
        sum_ += x
    print('The sum is:', sum_)

for i in range(6):
    slow_function(1_000_000 * 2 ** i)
```

Python shell

```
| The sum is: 499999500000
| slow_function took 0.27 sec
| The sum is: 1999999000000
| slow_function took 0.23 sec
| The sum is: 7999998000000
| slow_function took 0.41 sec
| The sum is: 31999996000000
| slow_function took 0.81 sec
| The sum is: 127999992000000
| slow_function took 1.52 sec
| The sum is: 511999984000000
| slow_function took 3.12 sec
```

Built-in `@property`

- decorator specific for class methods
- allows accessing `x.attribute()` as `x.attribute`,
  convenient if `attribute` does not take any arguments (also readonly)

rectangle1.py

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

#   @property
    def area(self):
        return self.width * self.height
```

Python shell

```
> r = Rectangle(3, 4)
> print(r.area())
| 12
```

rectangle2.py

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height
```

Python shell

```
> r = Rectangle(3, 4)
> print(r.area)
| 12
```

Class decorators

Python

```python
@dec2
@dec1
class A:
  pass
```

≡

Python

```python
class A:
  pass

A = dec2(dec1(A))
```

Module `dataclasses` (Since Python 3.7)

- New (and more configurable) alternative to `namedtuple`

Python shell

```
> from dataclasses import dataclass
> @dataclass  # uses a decorator to add methods to the class
  class Person:
      name: str  # uses type annotation to define fields
      appeared: int
      height: str = 'unknown height'  # field with default value
> person = Person('Donald Duck', 1934, '3 feet')
> person
| Person(name='Donald Duck', appeared=1934, height='3 feet')
> person.name
| 'Donald Duck'
> person.height = '3.5 feet'
> Person('Mickey Mouse', 1928)
| Person(name='Mickey Mouse', appeared=1928, height='unknown height')
```

docs.python.org/3/library/dataclasses.html#module-dataclasses

Raymond Hettinger - Dataclasses: The code generator to end all code generators - PyCon 2018

@functools.total_ordering (class decorator)

student.py

```
import functools

@functools.total_ordering
class Student():
    def __init__(self, name, student_id):
        self.name = name
        self.id = student_id
    def __eq__(self, other):
        return (self.name == other.name and self.id == other.id)
    def __lt__(self, other):
        my_name = ', '.join(reversed(self.name.split()))
        other_name = ', '.join(reversed(other.name.split()))
        return (my_name < other_name
                or (my_name == other_name and self.id < other.id))
donald = Student('Donald Duck', 7)
gladstone = Student('Gladstone Gander', 42)
grandma = Student('Grandma Duck', 1)
```

Automatically creates <, <=, >, >= if at least one of the functions is implemented and == is implemented

Python shell

```
> donald < grandma
| True
> grandma >= gladstone
| False
> grandma <= gladstone
| True
> donald > gladstone
| False
```

class_decorator.py

```python
def add_lessequal(cls):
    '''Class decorator to add __le__ given __eq__ and __lt__.'''
    cls.__le__ = lambda self, other : self == other or self < other
    return cls  # the original class cls with attribute __le__ added

def add_lessequal(cls):  # alternative
    class sub_cls(cls):
        def __le__(self, other):
            return self == other or self < other

    return sub_cls  # new subclass of class cls

@add_lessequal  # Vector = add_lessequal(Vector)
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def _length_squared(self):
        return self.x ** 2 + self.y ** 2

    def __eq__(self, other):
        # Required, otherwise Vector(1, 2) == Vector(1, 2) is False
        return self._length_squared() == other._length_squared()

    def __lt__(self, other):
        return self._length_squared() < other._length_squared()

#    def __le__(self, other):
#        return self._length_squared() <= other._length_squared()

#    def __le__(self, other):
#        return self == other or self < other
```

Python shell

```
> u = Vector(3, 4)
> v = Vector(2, 5)
> u.__eq__(v)
| False
> u.__ne__(v)
| True  # not u.__eq__(v)
> u.__lt__(v)
| True
> u.__gt__(v)
| NotImplemented  # special value
> u.__le__(v)
| True  # added by @add_lessequal
> u.__ge__(v)
| NotImplemented  # special value
> u == v
| False
> u != v
| True
> u < v
| True
> u > v  # v < u
| False
> u <= v
| True
> u >= v  # v <= u
| False
```

https://docs.python.org/3/reference/datamodel.html#basic-customization

Summary

- *@decorator_name*
- Pyton decorators are just syntatic sugar
- Adds functionality to a function without having to augment each call to the function or each return statement in the function
- There are decorators for functions, class methods, and classes
- There are many decorators in the Python Standard Library
- Decorators are easy to use
- ...and (slightly) harder to write