PeerWise

https://peerwise.cs.auckland.nz

PeerWise

Ask | Share | Learn

Welcome to PeerWise!

To log in, select your school / institution from the list below:

Aarhus University (Denmark) 1 Go » 2

Just type the first few characters...

PeerWise supports students in the creation, sharing, evaluation and discussion of assessment q

What is PeerWise?

Students use PeerWise to create and to explain their understanding of course related assessment questions, and to answer and discuss questions created by their peers.

Any subje

PeerWise is used in a wi
subjects, including Anthropo
Chemistry, Computer Scie
Population Health, Phar
Medicine, and many more...

Welcome to PeerWise for
Aarhus University

Please log in with your PeerWise account

username: 5
password:

login »

Forgotten your password? Get a new one

Forgotten your username? Recover it

Like to join? Please register...

Registration is very simple 3

participation.

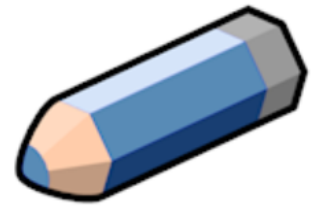
section.

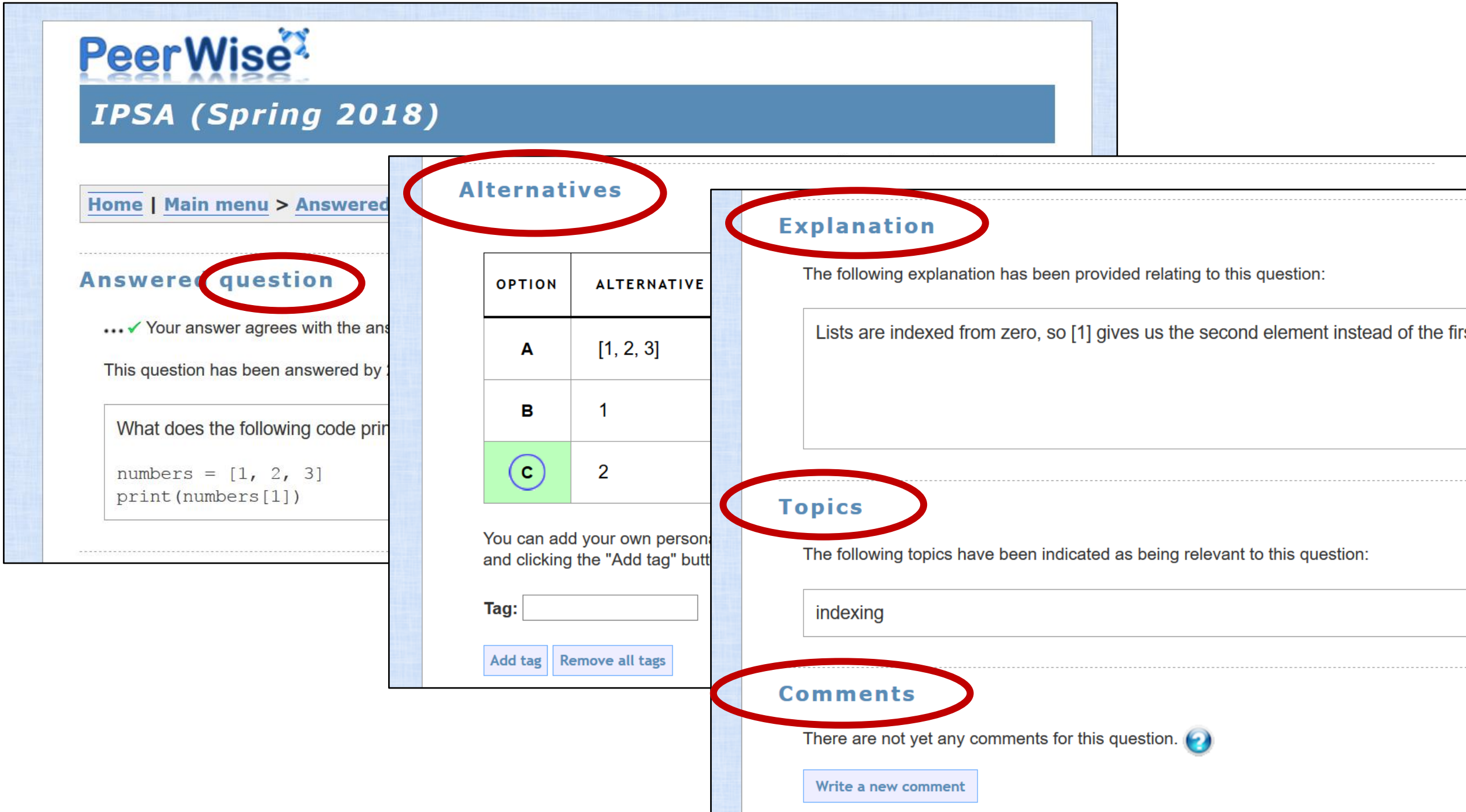
Registration

Welcome to PeerWise! Registration is very simple, and consists of the following 4 steps:

4

Step 1: choose a name
Step 2: choose a password
Step 3: enter the "Course ID" for the course you would like to join ← 17032
Step 4: enter your "Identifier" to join the course ← your AU ID




PeerWise
IPSA (Spring 2018)
Home | Main menu > Answered
Answered question
... ✓ Your answer agrees with the ans
This question has been answered by
What does the following code prin
numbers = [1, 2, 3]
print(numbers[1])
Alternatives
OPTION
ALTERNATIVE
A
[1, 2, 3]
B
1
C
2
You can add your own persona
and clicking the "Add tag" butt
Tag:
Add tag
Remove all tags
Explanation
The following explanation has been provided relating to this question:
Lists are indexed from zero, so [1] gives us the second element instead of the firs
Topics
The following topics have been indicated as being relevant to this question:
indexing
Comments
There are not yet any comments for this question.
Write a new comment

PeerWise

- Students (lecturers, TAs, ...) can poste multiple-choice questions together with answer options and explanations
- Students (and others) can
  - try to answer questions
  - add comments to questions
  - grade the quality and difficulty of questions
- Questions can be sorted by e.g. rating, difficulty, #answers, ...
- Activity on PeerWise releases badges 

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)

Decorators

- @

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

Python decorators are just syntatic sugar

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

≡

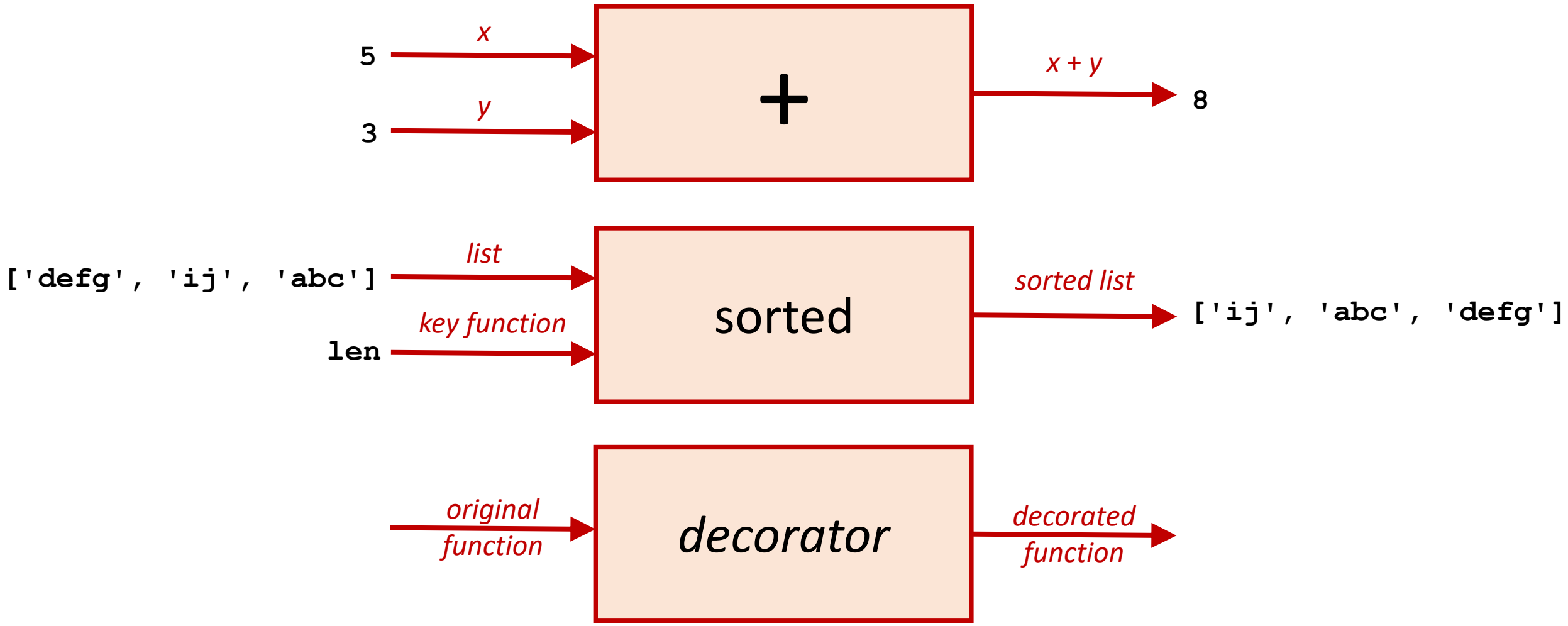
```python
def func(arg1, arg2, ...):
    pass

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

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)

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_quizz.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),\
                   ("got %s arguments, expected %s" % (len(args), 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 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("%s took %.2f sec" % (f.__name__, t))
        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

```
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

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

≡

```python
class A:
  pass

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

@functools.total_ordering (class decorator)

time_it.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
```

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