COMP1531

🐶Software Engineering
7.3 - Design - Decorators

In this lecture

Why?

- Writing clean and well designed code has huge benefits we've discussed previously, so let's learn some more

What?

- Decorators

Decorators

Decorators allow us to add functionality to a function without altering the function itself, by "decorating" (wrapping) around it.

But first... some background

Function Arguments

Arguments in python can either be keyword arguments (named) or non-keyword arguments.

Non-keyword arguments cannot appear after keyword arguments in the argument list

```python
def foo1(zid, name, age, suburb):
    print(zid, name, age, suburb)

def foo2(zid=None, name=None, age=None, suburb=None):
    print(zid, name, age, suburb)

if __name__ == '__main__':

    foo1('z3418003', 'Hayden', '72', 'Kensington')

    foo2('z3418003', 'Hayden')
    foo2(name='Hayden', suburb='Kensington', age='72', zid='z3418003')
    foo2(age='72', zid='z3418003')

    foo2('z3418003', suburb='Kensington')
```

decor1.py

Function Arguments

We can use a generalised method of capturing:

- *args: non-keyword arguments as a list
- *kwargs: keyword arguments as a dictionary

```python
def foo(zid=None, name=None, *args, **kwargs):
    print(zid, name)
    print(args) # A list
    print(kwargs) # A dictionary

if __name__ == '__main__':
    foo('z3418003', None, 'mercury', 'venus', planet1='earth', planet2='mars')
```

decor2.py

```python
def foo(*args, **kwargs):
    print(args) # A list
    print(kwargs) # A dictionary

if __name__ == '__main__':
    foo('this', 'is', truly='dynamic')
```

decor3.py

Decorators: First principles

Consider "make_uppercase" to be a decorator function. It allows you to add functionality to the get first name function without altering the function.

```python
def make_uppercase(input):
    return input.upper()

def get_first_name():
    return "Hayden"

def get_last_name():
    return "Smith"

if __name__ == '__main__':
  print(make_uppercase(get_first_name()))
  print(make_uppercase(get_last_name()))
```

decor4.py

A proper decorator

Now let's generalise it with the proper python decorator syntax.

```python
def make_uppercase(function):
        def wrapper(*args, **kwargs):
                return function(*args, **kwargs).upper()
        return wrapper

@make_uppercase
def get_first_name():
        return "Hayden"

@make_uppercase
def get_last_name():
        return "Smith"

if __name__ == '__main__':
    print(get_first_name())
    print(get_last_name())
```

decor5.py

This code can be used as a template

Decorator, run twice

```python
def run_twice(function):
        def wrapper(*args, **kwargs):
                return function(*args, **kwargs) \
                     + function(*args, **kwargs)
        return wrapper

@run_twice
def get_first_name():
        return "Hayden"

@run_twice
def get_last_name():
        return "Smith"

if __name__ == '__main__':
    print(get_first_name())
    print(get_last_name())
```

decor6.py

Decorator, more

decor7.py

```python
class Message:
        def __init__(self, id, text):
                self.id = id
                self.text = text

messages = [
        Message(1, "Hello"),
        Message(2, "How are you?"),
]

def get_message_by_id(id):
        return [m for m in messages if m.id == id][0]

def message_id_to_obj(function):
        def wrapper(*args, **kwargs):
                argsList = list(args)
                argsList[0] = get_message_by_id(argsList[0])
                args = tuple(argsList)
                return function(*args, **kwargs)
        return wrapper

@message_id_to_obj
def printMessage(message):
        print(message.text)

if __name__ == '__main__':
        printMessage(1)
```

Feedback