COMP1531

8.1 - Software Engineering Design Principles

SE Design Principles

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 Parameters

decor1.py

```
def fool(zid, name, age, suburb):
       print(zid, name, age, suburb)
 3
   def foo2(zid=None, name=None, age=None, suburb=None):
       print(zid, name, age, suburb)
 6
   if name == ' main ':
8
       foo1('z3418003', 'Hayden', '72', 'Kensington')
10
       foo2('z3418003', 'Hayden')
11
12
       foo2(name='Hayden', suburb='Kensington', age='72', zid='z3418003')
       foo2(age='72', zid='z3418003')
13
14
       foo2('z3418003', suburb='Kensington')
15
```

Function Parameters

decor2.py

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

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

decor3.py

```
1 def foo(*args, **kwargs):
2    print(args) # A list
3    print(kwargs) # A dictionary
4
5 if __name__ == '__main__':
6    foo('this', 'is', truly='dynamic')
```

A proper decorator

decor4.py

```
def make uppercase(input):
           return input.upper()
   def get first name():
 5
           return "Hayden"
 6
   def get last name():
 8
           return "Smith"
 9
10
   if name
               == ' main ':
       print(make_uppercase(get_first_name()))
11
12
       print(make uppercase(get last name()))
```

A proper decorator

decor5.py

This code can be used as a template

```
def make uppercase(function):
           def wrapper(*args, **kwargs):
 3
                   return function(*args, **kwargs).upper()
           return wrapper
 5
   @make uppercase
   def get first name():
           return "Hayden"
 8
10
   @make uppercase
11
   def get last name():
12
       return "Smith"
13
14 if name == ' main ':
     print(get_first_name())
15
16
       print(get last name())
```

Decorator, run twice

decor6.py

```
def run twice(function):
           def wrapper(*args, **kwargs):
                   return function(*args, **kwargs) \
                        + function(*args, **kwargs)
 5
           return wrapper
 6
   @run twice
   def get first name():
           return "Hayden"
10
   @run twice
   def get last name():
13
        return "Smith"
14
15
  if name == '
                    main
      print(get first name())
16
       print(get last name())
17
```

Decorator, run twice

```
1 class Message:
           def init (self, id, text):
                   self.id = id
                   self.text = text
6 messages = [
           Message(1, "Hello"),
           Message(2, "How are you?"),
10
  def get message by id(id):
12
           return [m for m in messages if m.id == id][0]
13
  def message id to obj(function):
14
15
           def wrapper(*args, **kwargs):
                   argsList = list(args)
16
17
                   argsList[0] = get message by id(argsList[0])
                   args = tuple(argsList)
18
                   return function(*args, **kwargs)
19
20
           return wrapper
21
22
   @message id to obj
23 def printMessage(message):
24
           print(message.text)
25
26 if
        name == ' main ':
27
           printMessage(1)
```

decor7.py

Single Responsibility Principle

Every module/function/class in a program should have **responsibility** for just a **single** piece of that program's functionality

Single Responsibility Principle

Functions

We want to ensure that each function is only responsible for one task. If it's not, break it up into multiple functions.

This is often a good idea. The only instances where this might not be a good idea are if it complicates the call**er** substantially (i.e. makes the code calling your split up functions overly complex)

Primary purpose: Readability and modularity

Single Responsibility Principle

Classes

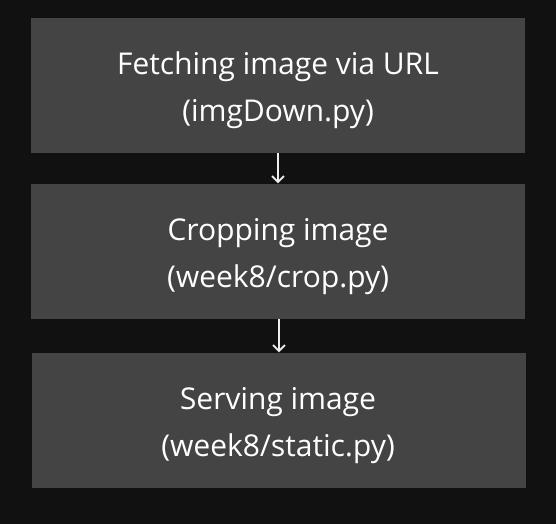
Three files:

- srp2.py: Poor SRP
- srp2_fixed.py: Fixed SRP, abstraction remains
- srp2_fixed2.py: Fixed SRP, no abstraction

We can apply the same principles to classes, ensuring that a single class maintains a single broad responsibility, and each function within the class also has a more specific single responsibility

Project Assistance

Storing and serving images



Imports

3 steps to making your imports happy:

- 1. Use absolute imports
- 2. Include __init__.py's in sub directories you want to import
- 3. export PYTHONPATH to your project folder

1. Use absolute imports

See week8/proj folder

2. __init__.py

Go and create empty __init__.py files in the main directory and any sub directory that you want to import from.

(Not required) Can read more here and here.

3. Python Path

This is something needed to make pytest work

If your project is in ~/cs1531/project

1 export PYTHONPATH="\$PYTHONPATH:~/cs1531/project"

You can add this line to your ~/.bashrc if you don't want to type it in every time you open a terminal