COMP1531

8.1 - Python - More Design + Iter3

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

```
1  def foo1(zid, name, age, suburb):
2      print(zid, name, age, suburb)
3
4  def foo2(zid=None, name=None, age=None, suburb=None):
5      print(zid, name, age, suburb)
6
7  if __name__ == '__main__':
8
9      foo1('z3418003', 'Hayden', '72', 'Kensington')
10
11     foo2('z3418003', 'Hayden')
12     foo2(name='Hayden', suburb='Kensington', age='72', zid='z3418003')
13     foo2(age='72', zid='z3418003')
14
15     foo2('z3418003', suburb='Kensington')
```

Function Parameters

decor2.py

```
1 def foo(zid=None, name=None, *args, **kwargs):
2     print(zid, name)
3     print(args) # A list
4     print(kwargs) # A dictionary
5
6 if __name__ == '__main__':
7
8     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

```
1  def make_uppercase(input):
2          return input.upper()
3
4  def get_first_name():
5          return "Hayden"
6
7  def get_last_name():
8          return "Smith"
9
10 if __name__ == '__main__':
11     print(make_uppercase(get_first_name()))
12     print(make_uppercase(get_last_name()))
```

A proper decorator

decor5.py

This code can be used as a template

```
1  def make_uppercase(function):
2          def wrapper(*args, **kwargs):
3                  return function(*args, **kwargs).upper()
4          return wrapper
5
6  @make_uppercase
7  def get_first_name():
8          return "Hayden"
9
10 @make_uppercase
11 def get_last_name():
12         return "Smith"
13
14 if __name__ == '__main__':
15     print(get_first_name())
16     print(get_last_name())
```

Decorator, run twice

decor6.py

```
1  def run_twice(function):
2          def wrapper(*args, **kwargs):
3                  return function(*args, **kwargs) \
4                       + function(*args, **kwargs)
5          return wrapper
6
7  @run_twice
8  def get_first_name():
9          return "Hayden"
10
11 @run_twice
12 def get_last_name():
13         return "Smith"
14
15 if __name__ == '__main__':
16     print(get_first_name())
17     print(get_last_name())
```

Decorator, more

decor7.py

```
1  class Message:
2          def __init__(self, id, text):
3                  self.id = id
4                  self.text = text
5
6  messages = [
7          Message(1, "Hello"),
8          Message(2, "How are you?"),
9  ]
10
11 def get_message_by_id(id):
12         return [m for m in messages if m.id == id][0]
13
14 def message_id_to_obj(function):
15         def wrapper(*args, **kwargs):
16                 argsList = list(args)
17                 argsList[0] = get_message_by_id(argsList[0])
18                 args = tuple(argsList)
19                 return function(*args, **kwargs)
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)
```

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

It's OK to do research

Some feedback from students has come in the form of being concerned they're being ask to solve problems they haven't been taught how to solve.

The nature of problem solving is like this often, and much of what you will be doing in your career will be team-based or self-guided research. It's not always fun, but it's certainly something to become familiar with.

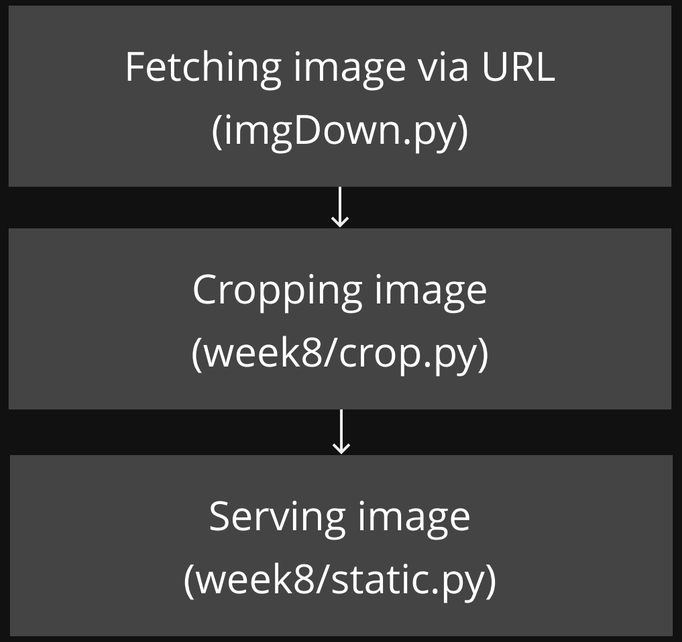
Sending Emails

This is one of the more challenging parts of Iteration 3. You can create a dummy gmail account and make sure it is a less secure app.

Once that is done, you can usually search around the internet for python-based SMTP libraries that will allow you to send emails.

Work collaboratively with your team to solve this problem.

Storing and serving images



Using threads & timers

Most of what we do in python is single-threaded, i.e. if an action takes a while to complete, the program stops and waits for that action to be complete.

To do more interesting things like "set a timer to execute code later" we need to use the threading library so that multiple paths/streams of the program can be completed concurrently.

Using threads & timers

timer.py

```
1 import threading
2 import time
3
4 def hello():
5     print("hello, Timer")
6
7 if __name__ == '__main__':
8     t = threading.Timer(3.0, hello)
9     t.start()
```

source:
https://www.bogotobogo.com/python/Multithread/python_multithreading_subclassing_TimerObject.php