Contents

Tips, Tricks, and Pitfalls

Level 5

I: Dates and Times

1. While this is not directly related to **datetime**, the **operator** module can come in handy when dealing with timedelta objects (and in many other situations). The **operator** module contains some useful functions that can be used instead of the standard Python operators. For example, **operator.add** is equivalent to **+** and **operator.sub** is equivalent to **-**. This can be useful if you wish to use an operator as a callable (to store in a variable or pass around as parameters to other functions). One such example is below:

```
import operator

# This dict maps strings to corresponding operator functions.
opDict = {
          '+': operator.add,
          '-': operator.sub,
          '*': operator.mul,
          '/': operator.div,
         }

def calculate(a, b, op):
    # op contains a callable.
    return op(a, b)

val1 = float(raw_input('Enter the first value: '))
opStr = raw_input('Enter the operator to perform (+, -, * /): ')
val2 = float(raw_input('Enter the second value: '))

# opDict.get(opStr) will return the operator function from the opDict,
# that corresponds to the opStr string.
# This function (callable) then gets passed-into the
# calculate function as the op parameter.
res = calculate(val1, val2, opDict.get(opStr))

print res
```

   Hint: This may come in handy for one of the exercises.

II: Decorators

1. The lecture mentioned that decorators are *syntactic sugar* and demonstrates how to create a simple decorator. However, we glossed over what the alternative (non syntactic-sugar) approach would be to achieve the same thing. The below is an example of function wrapping, with and without decorators:

   Without decorators:

```python
import time

def Timer(fn, *args, **kwargs):
    # Create an inner function to be returned.
    def timedFn():
        start = time.time()
        fn(*args, **kwargs)
        end = time.time()
        print 'Time taken for {0}: {1}'.format(fn, end-start)
    return timedFn

def MyFunction():
    time.sleep(5)

# Does not print the time taken
MyFunction()

# Wraps MyFunction in the Timer function and saved it down as a callable.
timedMyFunction = Timer(MyFunction)

# Prints the time taken for MyFunction
timedMyFunction()
```

   With decorators:

```python
import time

@Timer
def MyFunction():
    time.sleep(5)

# Prints the time taken for MyFunction
# There is no need to explicitly wrap each call to MyFunction, since it was decorated.
# This also makes it much simpler to turn the Timer for MyFunction on/off:
# In the non-decorator case, you would need to add/remove the Timer wrapping
# in every place in code that uses MyFunction. Whereas, in the decorator case, you can simply
# decorate/un-decorate the function declaration itself.
MyFunction()
```