New York University
School of Continuing and Professional Studies
Management and Information Technology

Advanced Python

Homework, Session 3

Homework due next week: **3.1 ONLY**

VIVA LAS MODULES: this assignment requires that you learn more about the **datetime** module and can get you started reading module documentation. I have listed the functions and objects you'll need in the homework discussion document.

A **datetime.date** object is a special object that can hold any date and calculate a new date based on a date interval (3 days, 2 weeks, etc.), as well as render the date as a string. The interface to **datetime.date** is not complicated, but it does require the user to learn some specialized syntax.

3.1 Create a module **date_simple** that provides a convenient interface to some of the date functions of the **datetime** module.

a. basic usage

The functions we'll create convert strings to dates or dates to strings. Each function either returns a **datetime.date** object (noted below as **dt**) or returns a string that represents a date. The module can thus allow you to easily calculate and represent dates without having to learn or remember the **datetime.date** syntax.

get_date(): takes an ***optional string*** and returns a date object

add_day(): takes a ***date object*** and ***an optional integer*** and returns a date object

add_week(): same as **add_day()** but * 7

format_date(): takes a ***date object*** and an ***optional string*** and returns a string

```
import date_simple as ds

# initializing a new datetime.date object
dt = ds.get_date()              # datetime.date object set to today
dt = ds.get_date('2016-05-05')  # object set to May 5, 2016
dt = ds.get_date('5/5/2016')    # same
dt = ds.get_date('5-May-16')    # same


# advancing the date
dt = ds.add_day(dt)             # new datetime object one day later
dt = ds.add_day(dt, 3)          # new datetime object 3 days later
dt = ds.add_day(dt, -2)         # new datetime object 2 days earlier


dt = ds.add_week(dt)            # same as add_day but adds 7 days
dt = ds.add_week(dt, 2)         # samE as add_day but adds 14 days
```

```
# print a formatted date string
print ds.format_date(dt)                # 2016-05-05
print ds.format_date(dt, 'MM/DD/YYY')   # 05/05/2016
print ds.format_date(dt, 'DD-Mon-YY')   # 05-May-16
```

b. extra credit usage: add month or year

Of course, a month is not always 30 days and a year is not always 365 days! These functions may have to go about a date change in a way other than by using the **datetime.timedelta** object. (In the special case of advancing from the last day of a longer month to a shorter (for example May 31 + 1 month), the date advances to the last day of the following month (i.e., June 30, since June has only 30 days). In the very special case of a leap-year day (i.e., February 29 + 1 year), advance to the end of February of the following year)

```
# add a month or year
dt = ds.add_month(dt)   # new datetime object one month later
dt = ds.add_year(dt)    # new datetime object one year later
```

c. handling and raising exceptions

Coding for a module requires a slightly different approach than for a script. If there is an error, we want to allow the calling code to handle it. For this reason, library code does not usually **exit()** on its own, but rather raises exceptions or allows exceptions to occur. We may wish to have some errors "bubble up" from the operations in our code (allowing them to occur), we may want to trap such exceptions and raise our own, or under certain circumstances we may wish to **raise** the exception ourselves if our own code detects an error.

Here's a rundown of exceptions required to be handled/raised -- you may think of others as well:

i. bad date format to **get_date()**: raise a **ValueError** exception with a descriptive message

ii. first argument to **add_day()** is not a **datetime.date** object, or optional 2nd argument is not an integer: raise a **TypeError** exception. (These errors must be trapped internally (i.e., in order to calculate dates you must use the arguments to **add_day()** in a **datetime.timedelta()** calculation, which would raise a **TypeError** or **ValueError**) and re-raised by the module with a module-specific **TypeError**.)

iii. Bad object types to **format_date()** (datetime.date object and string): raise a **TypeError** exception with a descriptive error message.

iv. Bad template format to **format_date()**: raise a **ValueError** exception with a descriptive error message.

[continued]

Here's an example of module code trapping an IOError exception and raising the same exception with a more specific error message:

```
try:
    fh = open('xxxx.txt')
except IOError, e:
    raise IOError("config file can't be opened:  {}".format(e))
```

Some sample output of the above:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
IOError: config file can't be opened:  [Errno 2] No such file or
directory: 'xxxx.txt'
```

In this particular case, the above code is responding to an **IOError** that Python raises when the OS can't find the file we're attempting to open. And, we're going right ahead and re-raising the same exception. But we're adding our own more descriptive error message.

Note the use of variable **e**: in this case this variable contains the error message that was passed from the OS to Python and then to the exception; it is the reason the file can't be opened (might be not found, or might be lacking permissions). So we include this string in our own error message.