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

# Advanced Python

Homework Discussion, Session 3

3.1 **date_simple** is conceived as a module that contains functions for working with dates. The motivation for the module is to simplify the interface to Python's **datetime** module (which is well designed, but still requires studying and remembering specialized syntax). Since **date_simple** merely provides an alternate interface to an existing module, we can refer to it as a "wrapper".

a. Review the **date_simple** module's features: PLEASE NOTE THE RETURN VALUE OF EACH FUNCTION. The required functions are **get_date()** (return ***a datetime.date object*** based on a string), **add_day()** (add day(s) to a datetime.date object, producing a new ***datetime.date object***), **add_week()** (like add_day() but add 7 days) and **format_date()** (return a ***formatted date string*** based on a datetime.date object). Just make sure the syntax and overall operations in the example are clear to you - you'll explore the **datetime.date** object in the next step.

b. Crack open the **datetime** module documentation and learn how to use the module to perform the same date processing functions as contemplated for the **date_simple** module. This obviously requires that you read part of the documentation and experiment with the module to achieve the following tasks. Here too are the module features to look for:

   i. create a **datetime.date** object representing today: look for the today() function

   ii. create a **datetime.date** object based on a specified string date: you can use the **strptime()** function or the **datetime.date()** constructor (a *constructor* is the function that produces a new object -- int(), float(), dict() are all constructors)

   iii. add day(s) to a **datetime.date** object: this is done using another object known as **datetime.timedelta**, which represents a particular interval. Look for the **datetime.timedelta()** constructor and learn how to create a timedelta object with an interval of one day (or multiple days). Lastly, learn how to add this **datetime.timedelta** object to a **datetime.date** object to produce a new **datetime.date** object with a modified date

   iv. produce a formatted date string based on a datetime.date object: you can use the **strftime()** function to do this

c. Write the "wrapper" functions that route arguments to the underlying **datetime.date** function calls, and return the results.

i. Note that to determine the type of an object, we must use the **isinstance()** function, which we haven't discussed yet

ii. Note also that the optional arguments to each of the functions are not noted with keywords (i.e., days=2). However, to be optional they must be noted with keywords in the function definitions, i.e. **get_date(str_date=None)** or **add_day(dateobj, days=1)**.

d. <u>Consider the error conditions and how your module will behave in response</u>. Remember that a module will almost never exit() the code directly, but instead will raise exceptions based on various error conditions. This use of exceptions allows our module to communicate to the calling code (the Python code importing our module) what the problem is; most importantly, it returns control to the calling code, allowing the exception to be trapped and handled without necessarily exiting the program.

e. <u>Once your code works, take a second look at it</u>: can it be shorter, cleaner, clearer? Can we eliminate repetition (DRY principle)? Is it hard to read or follow? Are there comments *only when the code is too complex to understand at first glance*? Is it as flat as possible? Does it follow all the guidelines in the Code Quality handout?

f. <u>Remember that as a module author/designer, you're doing more than just following my instructions</u>. Consider that this module is useful enough to be passed to colleagues at work or even interested parties on the web. How could you make it clear and easy to use, and especially have it behave properly when things go wrong?

3.2 (Currently optional -- chose from one of the following two assignments)

**class Logger**

Since each method is so similar, your mind should immediately split out what is the same about them (they all write to the file in the same way) and what is different about them (each one has a different priority level under which it will write).

A key concern here is to avoid repeated code. Each of the 5 "write" methods (**info()**, **debug()**, etc.) is very similar, so much of what happens there should probably go into a separate **write_log()** method, called from each of the "write" methods.

Also the same between them -- deciding on whether or not to write based on the method's priority level (debug, info, etc.) compared against the level configured in the constructor. Really, the only thing that's different between the methods is the level itself of each method.

This indicates that the testing to see whether the write should occur shouldn't even happen in each of the methods. Rather, each write method should simply send its level to the **write_log()** method, and let it compare the method's level to the configured level.

Since the levels are written in words and we need to see whether the write method's level is "above" or "below" the configured level, you should think in terms of numeric comparison, to avoid saying things like "if it's warning or error or critical, write it", "if it's info or warning or error or critical, write it", etc.. Instead, if you assign a number scale to the 5 levels you can simply say "if the method's level is greater than or equal to the configured level". So, I used a dictionary to link the level names to numeric values:

```
class Logger(object):
    priorities = {
        'DEBUG': 1,
        'INFO': 2,
        'WARNING': 3,
        'ERROR': 4,
        'CRITICAL': 5
    }
```

What else has to happen when doing a write? We have to check to see whether a date and/or the filename are flagged to appear in each log line. So I wrote a separate method that prepares a **prepend** string that will be prepended to the log line if it is so configured in the constructor.

Here are the methods I implemented and their specific work:

**__init__(self, filename, priority='DEBUG', datetime=True, scriptname=True)**: reading the arguments to the constructor, set each of these values in the instance: **filename, priority, write_date** and **write_scriptname**. (These last two are boolean flags.) Now since each method has access to the instance **self,** it will have access to these values and can make decisions about whether and what to write a log line by looking at the instance's attributes. I also made sure to store the **priority** value as an integer rather than as a string, because I know that later on I'll be testing priority numerically to see whether a log message should write.

**debug(self, msg)**: this write method takes a log message argument and simply calls **self.write_log(msg, 1)** with the log message and the "debug" priority level (which in my value system is **1**)**.**

**info(self, msg)**: just the same as **debug()** but calling **write_log()** with priority level **2**.

**warning(self, msg), error(self, msg)** and **critical(self, msg):** just the same as **debug()** but with levels **3**, **4** and **5**, respectively.

**write_log(self, msg, priority)**: a method to write to the file. It takes the message-to-be-logged argument and the priority level requested (i.e., from **debug()** the value will be **1**, etc.), and then takes three steps:

- checks to see if the priority level is high enough to warrant a write
- calls **compose_prepend()** to retrieve the date and/or filename string for this log write
- opens the file and writes the log line

**compose_prepend(self)**: simply checks the instance to see whether a date and/or filename are desired, in which case these are composed into a single string and returned.

If my description is maybe a bit confusing in its detail, think about writing out the function names and function calls in a new program, and use # comments to indicate what each function will do. Then you can get a better sense of the overall architecture and program flow.

**class Config**:

Again thinking about potential code repetition and the work that Python will be expected to do, and considering that we will be reading multiple key/values from the config file, we probably want our **__init__()** constructor to read the file into a dictionary, and then store that dictionary in the instance.

Then when a key's value is requested in the **get()** method, we can simply read from the dictionary.

When **set()** is called, however, we shouldn't just add the key/value to the instance's dictionary: we should also write the key/value to the file so that the file is always up-to-date and we don't have to think about writing to the file later on.

Here are the methods I implemented:

**__init__(self, filename, overwrite_keys=True)**: this method opens the file (traps and then re-raises a **IOError** exception if the file can't be read), loops through the file, splitting out the key/value pairs, and adds them to a new dictionary stored as an attribute in the instance. It also stores the **overwrite_keys** flag in another instance attribute.

**get(self, keyname)**: simply returns the value found for this keyname in the dictionary we stored in the instance in the constructor. If the key can't be found, it raises a **KeyError** exception

**set(self, keyname, value)**: adds the key and value to the instance's dictionary, and then writes the entire key/value set back to the file (I used a separate method called **write_data()** called from this method). **Thus the file is not appended to; it's completely overwritten**. This "wipe and reload" approach is much cleaner than trying to update the file with just one key/value change.

Of course **set()** also needs to see if the key already exists in the dictionary, because if **self.overwrite_keys** is **True** (as set in the **__init__()** constructor) then we should **raise** a **ValueError**.

**write_data(self)**: simply does the work of opening the file and writing each key and value in the dictionary to the file.