# Task

Our goal is to create the infrastructure for a calendar application that can store events and check if two events would overlap (clash). In order to do this, we will create classes that represent a moment in time, a span of time, an event (like "lunch with mom"), and an event planner. You will also create an Exception class, and you'll handle that exception and other kinds of more basic exceptions as part of the various methods and functions that you'll be writing. We won't actually be creating the full application, as that would be too large a project – but you could imagine storing the event planner to a text file and creating a small menu to load the event planner, show it to the user, and give them options for adding/removing events.

We will specify exactly what the class, method, and function signatures must be. We recommend that you start by creating a couple of the classes and just their init/str/repr methods, and play around with the classes as nothing more than fancy data containers with named subvalues. Perhaps the Moment and TimeSpan classes would be a good start, so that you see one class whose objects contain values of the other class as its own subvalues (because a TimeSpan is really a pair of Moments). Once you're comfortable with that, you can keep implementing more and more of the required methods, until you've created a nice set of data types that you could use in making your own PythonCalendar program. (There would be a lot of effort that would seem like busywork at this point and would make it larger than a two-week assignment, so we're not asking for that full calendar program).

# Required Classes

1. `Moment` class:

   Instance Variables:

   - `year` :: **`int`** Assume it's non-negative.
   - `month` :: **`int`** expected: 1 <= month <= 12
   - `day` :: **`int`** expected: 1 <= day <= last day of the current month
   - `hour` :: **`int`** expected: 0 <= hour <= 23
   - `minute` :: **`int`** expected: 0 <= minute<= 59

   Methods:

   - `__init__(self, year=0, month=0, day=0, hour=0, minute=0)` :: constructor.
     - If any of the 5 arguments given aren't integers, or they are out of the excepted ranges listed above, this method must raise a **`TimeError`** (see below for description). You can create any string as the message for the exception.
   - `__str__(self)` :: returns string in this format: `'2014/12/05-23:59'`
   - `__repr__(self)` :: returns a string in this format: `'Moment(2014, 12, 5, 23, 59)'`
   - `before(self, other)` :: returns bool: if this moment is strictly earlier than the other Moment.
   - `after (self, other)` :: returns bool: whether this moment is strictly later than the other Moment.
   - `delta(year=0,month=0, day=0, hour=0, minute=0)` :: given more of any unit of time, add it to this current moment.
     - Can have positive or non-positive values added.
     - Add in the argument for minutes first, then hours, days, months, years, etc. Deal with overflow each step of the way: there's no 6:82a.m., and there's no 35th of June, so you need to "clock over" the values accordingly. If you added in times from the largest to smallest units of time, you'd get different (==wrong) answers.
     - Pay close attention to leap years when deciding whether February 29th exists or not. Perhaps you could write an extra function, or revisit your lab tasks?

2. `TimeSpan` class:

Instance Variables:

- `start` :: **Moment**
- `stop` :: **Moment**

Methods

- `__init__(self, start, stop)` :: Constructor.
  - **stop** can't be before **start** (though they may be equivalent). If this rule is disobeyed, raise a **TimeError** with any message of your choosing.
- `__str__(self)` :: returns string representation with this format:
  `'TimeSpan(2014/12/12-07:30, 2014/12/12-10:15)'`
- `__repr__(self)` :: returns the following string representation.
  `'TimeSpan(Moment(2014, 12, 12, 7, 30), Moment(2014, 12, 12, 10, 15))'`
- `during_moment(self,moment)` :: returns a bool, indicating whether the given **Moment** is between (or equal to) the **start/stop** of this time span.
- `overlaps(self, other)` :: returns **True** if this **TimeSpan** overlaps at all with the other **TimeSpan**, even if just for a single **Moment**. Returns **False** if not.
- `set_duration(self, year=0, month=0, day=0, hour=0, minute=0)` :: allows us to reset the **stop** of this event's timespan. These five inputs describe the *length* of the timespan, they do not hardcode either endpoint's exact moment.
  - If the **stop** ends up before the **start** time, don't change the **stop** value, and instead raise a **TimeError** exception.

3. `Event` class:

**Instance Variables:**

- `title` :: **string**
- `location` :: **string**
- `timespan` :: **TimeSpan**

**Methods:**

- `__init__(self, title, location, timespan)` :: Constructor.
  - Assume that both **title** and **location** contain no quotes in them for simplicity's sake.
- `__str__(self)` :: returns string of the following shape (given that **title="lunch with mom"**, **location="home"**).
  `'Event: lunch with mom. Location: home. 2014/12/12-07:30, 2014/12/12-10:15'`
- `__repr__(self)` :: returns string of the following shape (same title/location as previous example):
  `'Event("lunch with mom", "home", Moment(2014, 12, 12, 7, 30), Moment(2014, 12, 12, 10, 15))'`

4. `TimeError(Exception)` class:

- We need **Exception** to be the parent class of **TimeError**. Be sure to declare your class as:
  `class TimeError(Exception):`

Instance Variable:

- `msg` :: **string**

Methods:

- `__init__(self, msg)` :: accepts and stores a message about how time has been misrepresented.
- `__str__(self)` :: returns the **msg** string.
- `__repr__(self)` :: returns the **msg** string.

5. `ScheduleConflictError(Exception)` class:
   - We need **`Exception`** to be the parent class. Be sure to declare your class as:
     ```
     class ScheduleConflictError (Exception):
     ```

   Instance Variable:
   - `msg` :: **`string`**

   Methods:
   - `__init__(self, msg)` :: accepts and stores a message about the conflict.
   - `__str__(self)` :: returns the **`msg`** string.
   - `__repr__(self)` :: returns the **`msg`** string.

6. `EventPlanner` class:

   **Instance Variables**
   - `owner` :: **`string`**
   - `events` :: list of **`Event`** values

   **Methods**
   - `__init__ (self, owner, events=None)` :: Constructor. If **`events==None`**, this constructor creates a new empty list to initialize the **`events`** variable. Otherwise, the given argument is used.
   - `__str__(self)` :: returns string of the shape of these examples.
     - `'EventPlanner("Lisa Simpson", [Event("lunch with mom", "home", Moment(2014, 12, 12, 7, 30), Moment(2014, 12, 12, 10, 15)), Event("sleep", "bedroom", Moment(2014, 12, 12, 10, 30), Moment(2014, 12, 15, 14, 30))]'` **(all in one line)**
     - `'EventPlanner("Bart Simpson",[])'`
   - `__repr__(self)` :: returns the same string as `__str__`.
   - `add_event(self, event)` :: adds an event to this **`EventPlanner`** if there's space for it (no overlapping events that are already scheduled). Requirement: **always place the new event at the END of the list!** (we need to be able to predict the ordering for testing purposes).
     - If any event is already planned with any overlap with the new event, then we are not allowed to add this event. Instead of adding the event to events, you must raise a **`ScheduleConflictError`** (with any string message of your choosing).
     - Returns **`None`** when successful.
   - `add_events(self, events)` :: attempts to add multiple events from a list of **`Event`** values.
     - ***must catch*** any `ScheduleConflictErrors`. These `Event` values that caused exceptions when we attempted to add them aren't added into the events list.
     - Returns an integer, how many events ***failed*** to be added. (A completely successful run returns 0).
   - `available_at(self, moment)` :: figures out whether we're currently in another event at that exact moment or not; returns a bool (**`True`** for available, **`False`** for not).
   - `available_during(self, timespan)` :: figures out whether we're available during the entire duration of the given timespan; returns a bool (**`True`** for available, **`False`** for not).
   - `can_attend(self, event)` :: determines if the **`Event`** argument fits our current schedule; returns that boolean result.

## Notes / Assumptions / Requirements

- **Don't import anything**, and you can use all built-ins/methods that are otherwise available. ☺
- You may add as many additional methods and functions as you'd like in support of your solution.
- `__str__` and `__repr__` are used by many other places in Python to get a string representation of the object. Specifically, `str()` calls `__str__`, and `repr()` calls `__repr__`, on any objects they receive as arguments. `__str__` should generate a human-centric representation, appropriate for a human reading what is present. `__repr__` should generate a Python-centric representation. The goal of `__repr__` is actually to have a string that could be evaluated and re-generate an identical object. In general, we would quite prefer it to look like a valid constructor call, if possible. We can have the same representation in `__str__` and `__repr__` if we'd like; in fact, ***when we didn't explicitly describe two different string representations*** for `__str__` and `__repr__` to return, we can define one in terms of the other, like this:

```
def __repr__(self):
    return str(self)
```

  Just remember the original intent of `str` vs `repr`. It's good practice to define `init`, `str`, and `repr` immediately before writing any extra methods in a Python class.
- When representing a `Moment`, notice that we *always* go from largest to smallest units of time: `yyyy/mm/dd-hh:mm`. (There don't have to be exactly four columns for year, but the others are exact; pad with zeroes always).
- Exceptions – notice where we create exception types, and where we catch those exceptions. Be sure you don't catch the exception too early! When testing smaller parts of your code, it may be possible and even expected/required that specific inputs to a function/method will cause an exception to propagate, rather than returning normally with a return value. In the beginning of this specification, we described possible extensions to this program to make a full calendar application. This user interaction would be a great place to catch more exceptions and ask them to type in values again whenever we catch certain exception types. Since we're not writing the larger program that was described, there's no menu and user interaction present to do this particular style of interaction.

## Testing Your Program

We will be using your class definitions to create objects, and then we will be calling your methods on those objects, and seeing if they have the right behaviors and updated attribute values. This is why we need you to exactly name the classes, methods, parameters, and instance variables exactly as named. It's also why our first suggestion was to create a few classes with not much more than constructors, so that you can interact with them and see them behaving correctly as soon as possible and in the same fashion as we will be testing your code.

We've actually been testing the last few projects with an idea called unit testing, which we briefly discussed earlier in the semester. It was hard to do proper unit testing without function definitions to identify small parts of the program that could be run separately. Now that we've got functions to test, and the classes/objects of the unittest module with which to perform testing, unit testing makes sense. The **`unittest`** module, which helps us write unit testing code, requires writing classes and methods in order to use it. We couldn't really introduce it to you any sooner. But now that you've learned how to create classes and objects, you're ready to learn how to write some unit testing! It's not a required or graded part of your assignment, but we're providing some information on how to create unit tests in this fashion. The one noticeable limitation is that we're creating a separate file for testing, and it needs to import your project code; while performing testing and development, you'll want a name that doesn't start with numbers so that it's a valid identifier and can be imported. So instead of **`2XX_First_List_P7.py`**, maybe name it **`First_Last_P7.py`** until just before you submit your code.

### Creating Test Cases Using PyUnit

There is a sample **`tests_p7.py`** file available to get you started (on piazza). How do you use the provided file? The short answer is to edit **`tests_p7.py`** by:

- Adding more testX methods that check various details about your code as wanted by calling either **`self.assertEquals(expr1, expr2)`**, or calling **`self.assertTrue(bool_expr)`**.
- Update **`num_tests`** so that the rest of the provided code can construct and run the suite of tests for you.

The long version is to read below. The testing file first needs to both import the unit testing code and your own project; here's how the sample **`tests_p7.py`** file begins:

```
import unittest
from your_project_7_code import *    # FIX THE NAME
```

Each test case will define a class, with **`unittest.TestCase`** as the parent class, and it will also define multiple **`testX`** methods (**`X`** will be sequential integers).

```
class AllTests (unittest.TestCase):
     """descriptive comment goes here"""
     def test1(self):
       """descriptive comment"""
        <CODE GOES HERE>
     def test2(self):
       """descriptive comment"""
        <CODE GOES HERE>
num_tests = 2
```

You can add more **`testX`** methods to create more and more checks on how your code behaves.

When you write code to perform the actual test, you can call the method **`self.assertEquals()`** on anything we wish to ensure is equal, such as a particular **`Moment`** object's hour value and a specific number.

The last step is to make our tests runnable. We need to create a runner, create a suite of test cases, and then let the runner run the suite of tests:

```
# create an object that can run tests.
runner = unittest.TextTestRunner()

# define the suite of tests that should be run.
suite1 = TheTestSuite()

# let the runner run the suite of tests.
runner.run(suite1)
```

And that's it! Any situation you want to set up, run, and check that something happens, you can now write those tests directly. It's entirely up to you how involved any individual test case gets.

## Running your Test Cases

Once you've got your wonderful test suite all written up, we want to run it!

```
demo$ python3 tests_p7.py
..
----------------------------------------------------
Ran 2 tests in 0.000s  OK
demo$
```

If everything goes well, you'll get the "OK" message. If not, your **`"""descriptive comment"""`** will be displayed, along with some error information.

## Further Reading

If you'd like to get more advanced with your test cases, please take a look at the **`pyunit`** documentation:

- putting multiple tests in one class, and creating a "test suite" (as we've done)
- creating a simple test case (as we could have done)