A Closer Look at MeetMe

Classes and Objects and Methods, Oh My

So let’s start with something simpler ...

A die (plural: dice) with any number of faces.

Constructor die(nfaces) determines number of faces.

Methods: roll()

Public attribute: value

MeetMe has several moving parts

A die is a kind of object ...

class Die(object):
    ""
    A die (plural: dice) that can be rolled to randomly choose one of its faces. The die can have any positive number of faces.
    ""
Each die has its own namespace.

class Die(object):
    
    A die (plural: dice) that can be rolled to randomly choose one of its faces. ...

    Public attribute:
        value: an integer between 1 and the number of faces, indicating which face is currently showing.
        
    __init__(...)
        The new die has nfaces faces.
        args:
            self: this die
            nfaces: how many faces the die has
        effects:
            this die will have nfaces faces, one of which is initially showing.
    
    _faces = 1    # Private attribute
    value = 1     # Public attribute

roll() will roll the die (choose a random face)

def roll(self):
    
    Randomly choose a face (numbered 1..faces).
    args:
        self: this die
    effects:
        one of the faces is selected and will be accessible in subsequent calls to face()
    
    self.value = random.randint(1,self._faces)

__init__(...) says how to initialize a new die

class die(object):
    ... 
    
    def __init__(self, nfaces):
        
        The new die has nfaces faces.
        args:
            self: this die
            nfaces: how many faces the die has
        effects:
            this die will have nfaces faces, one of which is initially showing.
        
        self._faces = nfaces
        random.seed()
        self.roll()

This is already enough to use dice ...

d1 = Die(6)
d2 = Die(6)
d1.roll()
d2.roll()
print("Initial roll is ", d1.value, d2.value)

$ python3 die.py
Initial roll is  2 5
Maybe I’d like to compare dice, using <

def __lt__(self, other):
    """
    self < other if self is showing a lower-numbered face.
    args:
        self: this die
        other: another die
    returns:
        boolean value indicating whether this die is showing
        a lower-numbered face than the other die.
    """
    return self.value < other.value

I can use it ...

def __lt__(self, other):
    """
    ... 
    return self.value < other.value 
    ...

d1 = Die(6)
d2 = Die(6)
d1.roll()
d2.roll()
print("Initial roll is ", d1.value, d2.value)
if d1 < d2:
    print("The first die is smaller")
else:
    print("d2 wins")

I’d like print(d) to do something reasonable ...

def __str__(self):
    """
The face of the die is represented 
by digits.
    """
    return "[" + str(self.value) + "]"

So now I can use a die where a string is needed ...

for i in range(10):
    d1.roll()
    d2.roll()
    if d1 < d2:
        relation = "is less than"
    else:
        relation = "is at least as large as"
    print("Rolled ", d1, relation, d2)

In summary ...

class Die(object):
    value = 1  # Public attribute

def __init__(self, nfaces):
    ...

def roll(self):
    ...

def __lt__(self, other):
    return self.value < other.value

def __str__(self):
    return "[" + str(self.value) + "]"
**So back to Meetme**

<table>
<thead>
<tr>
<th>Potential meeting times</th>
<th>Kevin's available times</th>
<th>Mary's available times</th>
</tr>
</thead>
<tbody>
<tr>
<td>06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td></td>
<td></td>
</tr>
<tr>
<td>08</td>
<td></td>
<td></td>
</tr>
<tr>
<td>09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>11-noon</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>5:00pm</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Creating an Appt: Constructor**

```python
import datetime
class Appt:
    def __init__(self, day, begin, end, desc):
        """Create an appointment on date from begin time to end time.

        Arguments:
        day: A datetime.date object
        begin: A datetime.time object
        end: A datetime.time object,
        after begin
desc: A string describing the appointment
        ""
        self.day = day
        self.begin = begin
        self.end = end
        self.desc = desc
        return
```

---

**The constructor is a bit of a pain ...**

```python
sample = Appt(datetime.date(2012, 10, 31),
              datetime.time(14, 30),
              datetime.time(15, 45),
              "Sample appointment")
```

*We can make it easier with a “factory” to create Appt objects from text like “2012.10.31 14:30 15:45 | Sample meeting”*
Creating an Appt from text: Factory method

@classmethod
def from_string(cls, txt):
    """Factory parses a string to create an Appt
    timespec = fields[0].strip()
desc = fields[1].strip()
fields = timespec.split()
...
appt_date_text = fields[0]
appt_begin_text = fields[1]
appt_end_text = fields[2]
fields = appt_date_text.split(" ")
try:
    year = int(fields[0].strip())
    month = int(fields[1].strip())
    day = int(fields[2].strip())
except:
    raise ValueError("Date in Appt literal should be 9999.99.99 (Year.Month.Day)")

Example:
an_appt = Appt.from_string("2012.10.31 14:30 15:45 | Team mtg")

Exceptions

@classmethod
def from_string(cls, txt):
    """Factory parses a string to create an Appt
    if len(fields) != 2:
        raise ValueError("Appt literal requires exactly one '|' before description")
    fields = appt_date_text.split(" ")
    try:
        year = int(fields[0].strip())
        month = int(fields[1].strip())
        day = int(fields[2].strip())
    except:
        raise ValueError("Date in Appt literal should be 9999.99.99 (Year.Month.Day)")

Keyword (default) arguments

def intersect(self, other, desc="":
    """Return an appointment representing the period in common between this appointment and another. Requires self.overlaps(other).
    Arguments:
        other: Another Appt
desc: (optional) description text for this appointment.
    Returns:
        An appointment representing the time period in common between self and other. Description of returned Appt is copied from this (self), unless a non-null string is provided as desc.
    ""
    if desc="":
        desc = self.desc
The Agenda class “wraps” a list of Appit

class Agenda:

    """"An Agenda is essentially a list of appointments, with some agenda-specific methods.
    """

    def __init__(self):
        """"An empty agenda."""
        self.appts = []

    def append(self, appt):
        """"Add an Appt to the agenda."""
        self.appts.append(appt)

    def __len__(self):
        """"Number of appointments, callable as built-in len() function"""
        return len(self.appts)

Agenda adds new Appt-specific methods

class Agenda:

    ... @classmethod
def from_file(cls, f):
        """"Factory: Read an agenda from a file. ..."
        """

def intersect(self, other, desc=""""):
    """"Return a new agenda containing appointments that are overlaps between appointments in this agenda and appointments in the other agenda."""
What does the main program do?

- Read and interpret the command line
- Open each agenda file
- Create Agenda objects to do the main work
  - Create starting “available” block
  - Intersect with each personal agenda file
- Print results