

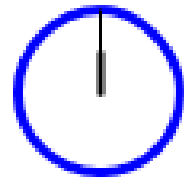
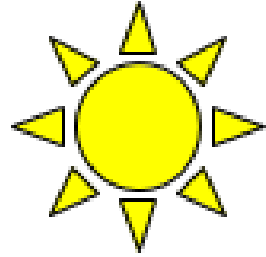
# UTC offsets

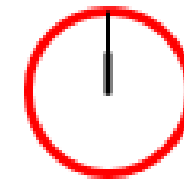
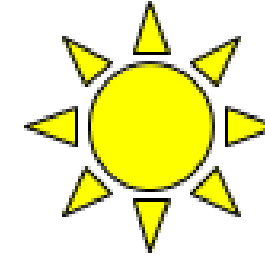
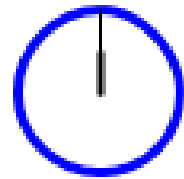
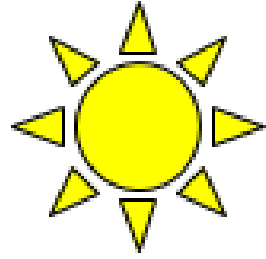
WORKING WITH DATES AND TIMES IN PYTHON

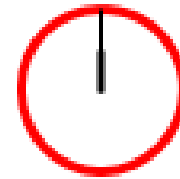
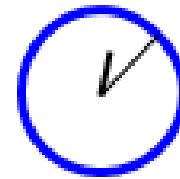
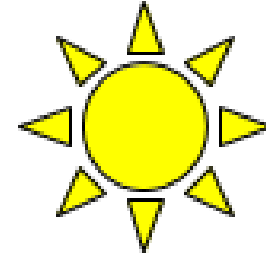
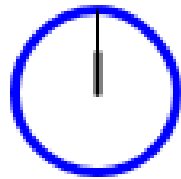
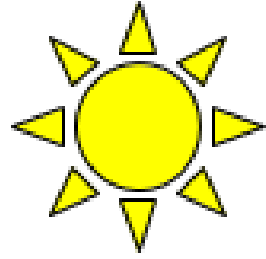


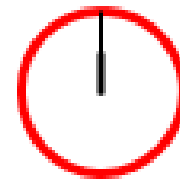
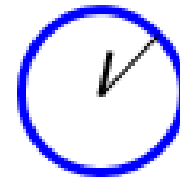
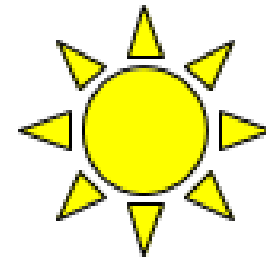
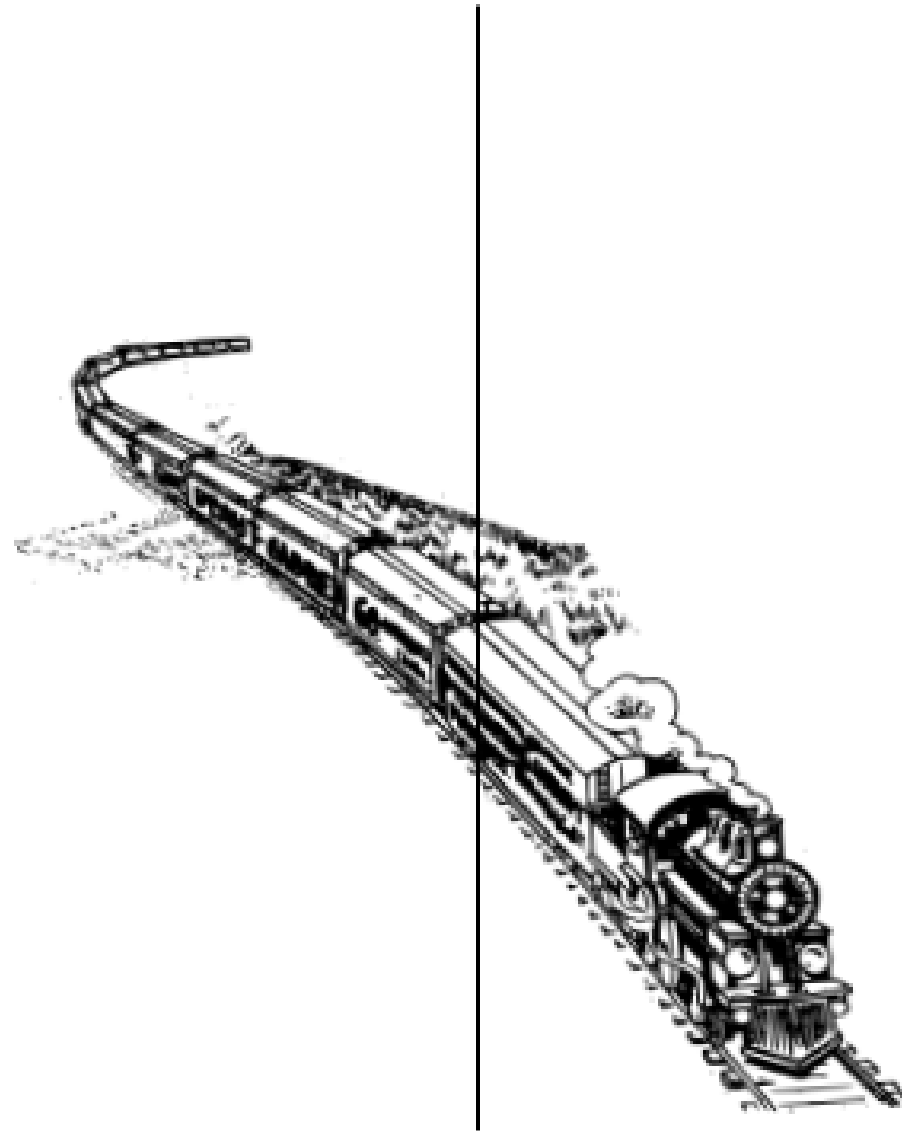
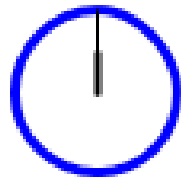
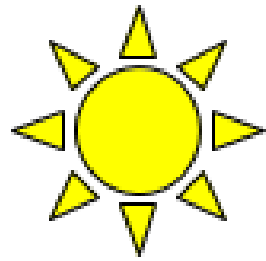
**Max Shron**

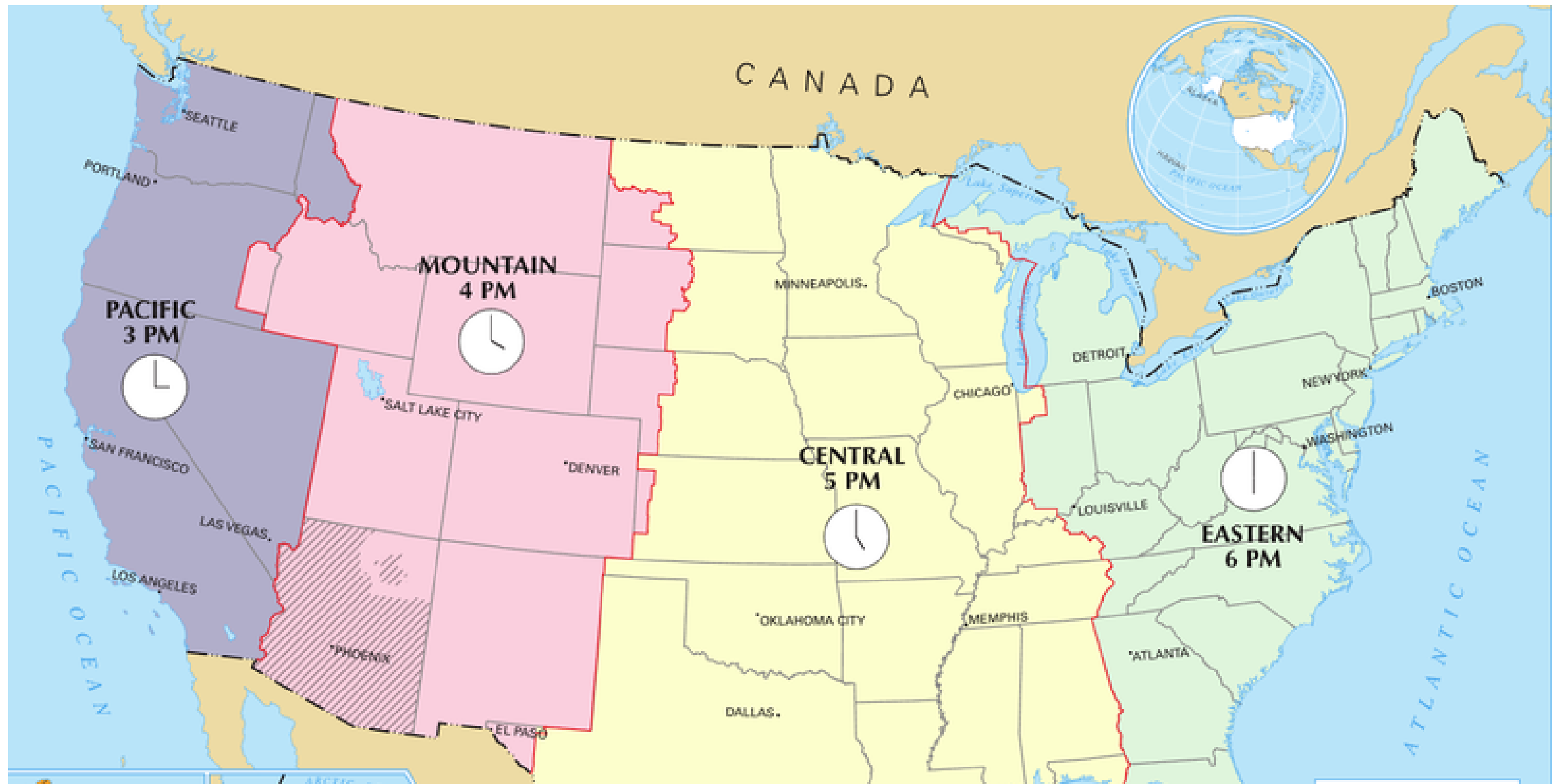
Data Scientist and Author

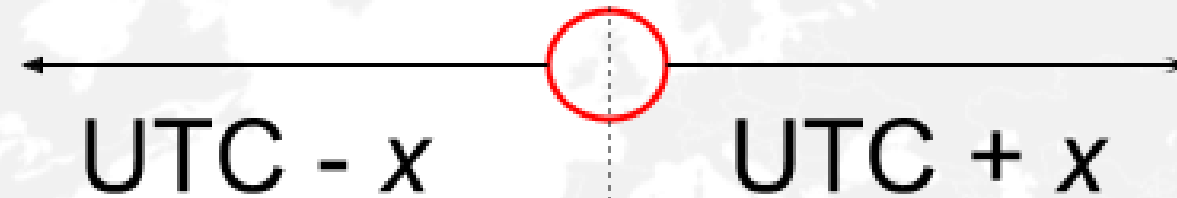












## 7. UTC

But since we're not using the sun anymore, how do we know how to set the clock? Because the United Kingdom was the first to standardize its time, everyone in the world sets their clocks relative to the original historical UK standard. This standard time is called UTC. Because all clocks are set relative to UTC, we can compare time around the world. Generally, clocks west of the UK are set earlier than UTC, and clocks east of the UK are set later than UTC. For example, the eastern United States is typically UTC minus 5 hours, while India is typically UTC plus 5 hours 30 minutes.

# UTC

```
# Import relevant classes  
from datetime import datetime, timedelta, timezone
```



# UTC

```
# Import relevant classes
from datetime import datetime, timedelta, timezone

# US Eastern Standard time zone
ET = timezone(timedelta(hours=-5))

# Timezone-aware datetime
dt = datetime(2017, 12, 30, 15, 9, 3, tzinfo = ET)
```

```
print(dt)
```

```
'2017-12-30 15:09:03-05:00'
```

## 9. UTC

We create a timezone object, which accepts a timedelta that explains how to translate your datetime into UTC. In this case, since the clock that measured our bicycle data set was five hours behind UTC, we create ET to be at UTC-5. We can specify what time zone the clock was in when the last ride started in our data set. The clock that recorded the ride was 5 hours behind UTC. Now if you print it, your datetime includes the UTC offset.

# UTC

```
# India Standard time zone
IST = timezone(timedelta(hours=5, minutes=30))
# Convert to IST
print(dt.astimezone(IST))
```

```
'2017-12-31 01:39:03+05:30'
```

## 10. UTC

Making a datetime "aware" of its timezone means you can ask Python new questions. For example, suppose you want to know what the date and time would have been if the clock had been set to India Standard Time instead. First, create a new timezone object set to UTC plus 5 hours 30 minutes. Now use the `astimezone()` method to ask Python to create a new datetime object corresponding to the same moment, but adjusted to a different time zone. In this case, because clocks in India would have been set 10.5 hours ahead of clocks on the eastern US, the last ride would have taken place on December 31, at 1 hour, 39 minutes, and 3 seconds past midnight local time. Same moment, different clock.

# Adjusting timezone vs changing tzinfo

```
print(dt)
```

```
'2017-12-30 15:09:03-05:00'
```

```
print(dt.replace(tzinfo=timezone.utc))
```

```
'2017-12-30 15:09:03+00:00'
```

```
# Change original to match UTC  
print(dt.astimezone(timezone.utc))
```

```
'2017-12-30 20:09:03+00:00'
```

## 11. Adjusting timezone vs changing tzinfo

Finally, there is an important difference between adjusting timezones and changing the tzinfo directly. You can set the tzinfo directly, using the `replace()` method.

Here we've set the tzinfo to be `timezone.utc`, a convenient object with zero UTC offset. The clock stays the same, but the UTC offset has shifted. Or, just like before, you can call the `astimezone()` method. Now if we adjust into UTC with `astimezone(timezone.utc)`, we change both the UTC offset and the clock itself.

# UTC Offsets

WORKING WITH DATES AND TIMES IN PYTHON

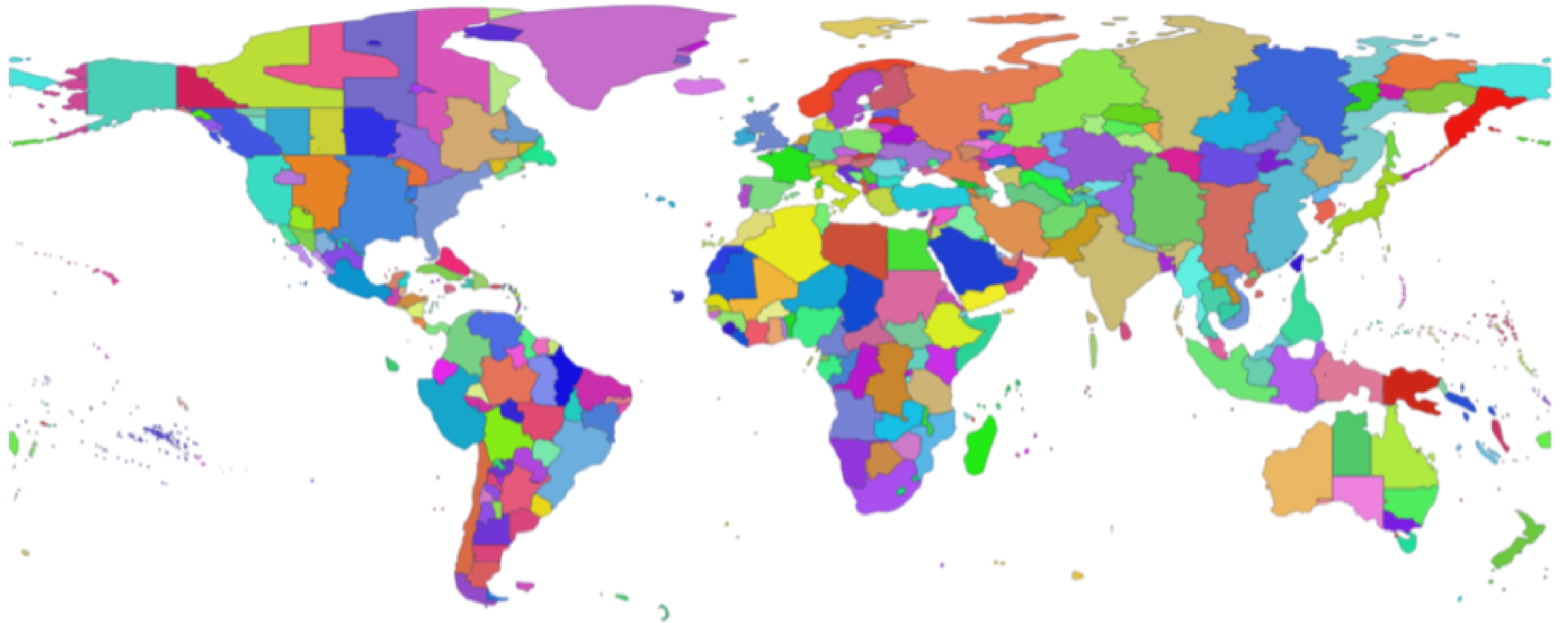
# Time zone database

WORKING WITH DATES AND TIMES IN PYTHON



**Max Shron**

Data Scientist and Author



# Time zone database

```
# Imports  
from datetime import datetime  
from dateutil import tz
```

tz database

## 3. Time zone database

Thankfully, yes. There is a database called tz, updated 3-4 times a year as timezone rules change. Instead, you will use a package called dateutil.

# Time zone database

```
# Imports
from datetime import datetime
from dateutil import tz

# Eastern time
et = tz.gettz('America/New_York')
```

## tz database

- Format: 'Continent/City'

### 4. Time zone database

Let's start by making a timezone object that corresponds to the eastern United States, where our bicycle data comes from. Within tz, time zones are defined first by the continent they are on, and then by the nearest major city. For example, the time zone used on the eastern seaboard of the United States is 'America/New York'. We fetch this timezone by calling tz.gettz(), and passing 'America/New York' as the string.



# Time zone database

```
# Imports
from datetime import datetime
from dateutil import tz

# Eastern time
et = tz.gettz('America/New_York')
```

## 6. Time zone database

Let's look at our last ride again. Instead of specifying the UTC offset yourself, you pass the timezone you got from tz.

## tz database

- Format: 'Continent/City'
- Examples:
  - 'America/New\_York'
  - 'America/Mexico\_City'
  - 'Europe/London'
  - 'Africa/Accra'

# Time zone database

```
# Last ride  
last = datetime(2017, 12, 30, 15, 9, 3, tzinfo=et)
```

```
print(last)
```

```
'2017-12-30 15:09:03-05:00'
```

## 7. Time zone database

Even more excitingly, this same object will adjust the UTC offset depending on the date and time. If we call `datetime()` with the time of our first ride, and pass in the same timezone info, we see that it gives us a different UTC offset. We will discuss daylight savings time in the next lesson, but suffice to say that in some places the clocks change twice a year. Instead of having to look up when these things change, we just ask the timezone database to know for us. `tz` includes rules for UTC offsets going all the way back to the late 1960s, and sometimes earlier. If you have data stretching over a long period of time, and you really care about getting the exact hours and minutes correct, you can use `tz` to put all of your date and timestamps on to a common scale.

# Time zone database

```
# Last ride
last = datetime(2017, 12, 30, 15, 9, 3, tzinfo=et)
print(last)
```

```
'2017-12-30 15:09:03-05:00'
```

```
# First ride
first = datetime(2017, 10, 1, 15, 23, 25, tzinfo=et)
print(first)
```

```
'2017-10-01 15:23:25-04:00'
```

# Time zone database

WORKING WITH DATES AND TIMES IN PYTHON

# Starting Daylight Saving Time

WORKING WITH DATES AND TIMES IN PYTHON

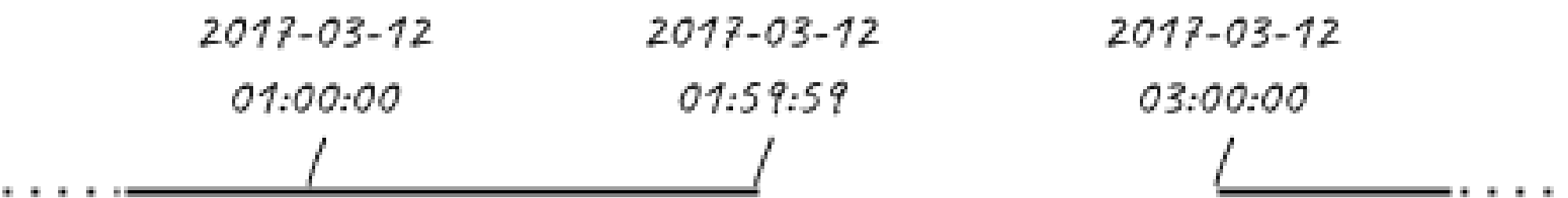


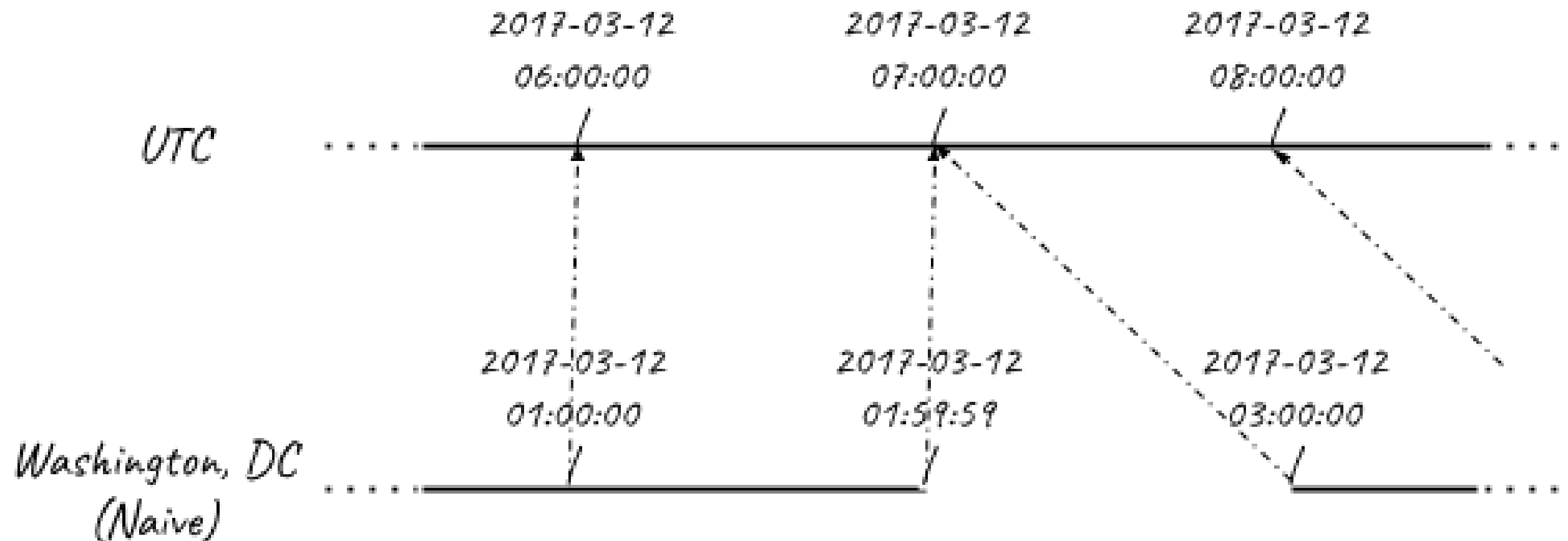
**Max Shron**

Data Scientist and Author

Washington, DC  
(Naive)

..... 2017-03-12 01:00:00 | 2017-03-12 01:59:59 | 2017-03-12 03:00:00 .....





## 2. Start of Daylight Saving Time

Let's look at an example. On March 12, 2017, in Washington, D.C., the clock jumped straight from 1:59 am to 3 am. The clock "springs forward". It never officially struck 2 am anywhere on the East Coast of the United States that day.

# Start of Daylight Saving Time

```
spring_ahead_159am = datetime(2017, 3, 12, 1, 59, 59)
spring_ahead_159am.isoformat()
```

```
'2017-03-12T01:59:59'
```

```
spring_ahead_3am = datetime(2017, 3, 12, 3, 0, 0)
spring_ahead_3am.isoformat()
```

```
'2017-03-12T03:00:00'
```

```
(spring_ahead_3am - spring_ahead_159am).total_seconds()
```

```
3601
```



# Start of Daylight Saving Time

```
from datetime import timezone, timedelta
```

```
EST = timezone(timedelta(hours=-5))
```

```
EDT = timezone(timedelta(hours=-4))
```

# Start of Daylight Saving Time

```
spring_ahead_159am = spring_ahead_159am.replace(tzinfo = EST)
spring_ahead_159am.isoformat()
```

```
'2017-03-12T01:59:59-05:00'
```

```
spring_ahead_3am = spring_ahead_159am.replace(tzinfo = EDT)
spring_ahead_3am.isoformat()
```

```
'2017-03-12T03:00:00-04:00'
```

```
(spring_ahead_3am - spring_ahead_159am).seconds
```

```
1
```

## 6. Start of Daylight Saving Time

We assign our first timestamp, at 1:59 am to be in EST. When we call `isoformat()`, we see it has the correct offset. We assign our second timestamp, at 3:00 am, to be in EDT, and again check the output with `isoformat()`. When we subtract the two datetime objects, we see correctly that one second has elapsed. Putting things in terms of UTC once again allowed us to make proper comparisons.

# Start of Daylight Saving Time

Using `dateutil`

```
# Import tz
from dateutil import tz

# Create eastern timezone
eastern = tz.gettz('America/New_York')

# 2017-03-12 01:59:59 in Eastern Time (EST)
spring_ahead_159am = datetime(2017, 3, 12, 1, 59, 59,
                              tzinfo = eastern)

# 2017-03-12 03:00:00 in Eastern Time (EDT)
spring_ahead_3am = datetime(2017, 3, 12, 3, 0, 0,
                            tzinfo = eastern)
```

## 7. Start of Daylight Saving Time

But how do we know when the cutoff is without looking it up ourselves? `dateutil` to the rescue again. Just like before when it saved us from having to define timezones by hand, `dateutil` saves us from having to know daylight savings rules. We create a timezone object by calling `tz.gettz()` and pass our timezone description string. Recall that since Washington, D.C. is in the `America/New_York` time zone, that's what we use. Once again we create a datetime corresponding to 1:59 am on the day that the east coast of the US springs forward. This time though, we set the `tzinfo` to eastern time. Similarly, we create a datetime set to 3 am on March 12th, and when we set `tzinfo` to be eastern time, `dateutil` figures out for us that it should be in EDT.

# Daylight Saving

WORKING WITH DATES AND TIMES IN PYTHON

# Ending Daylight Saving Time

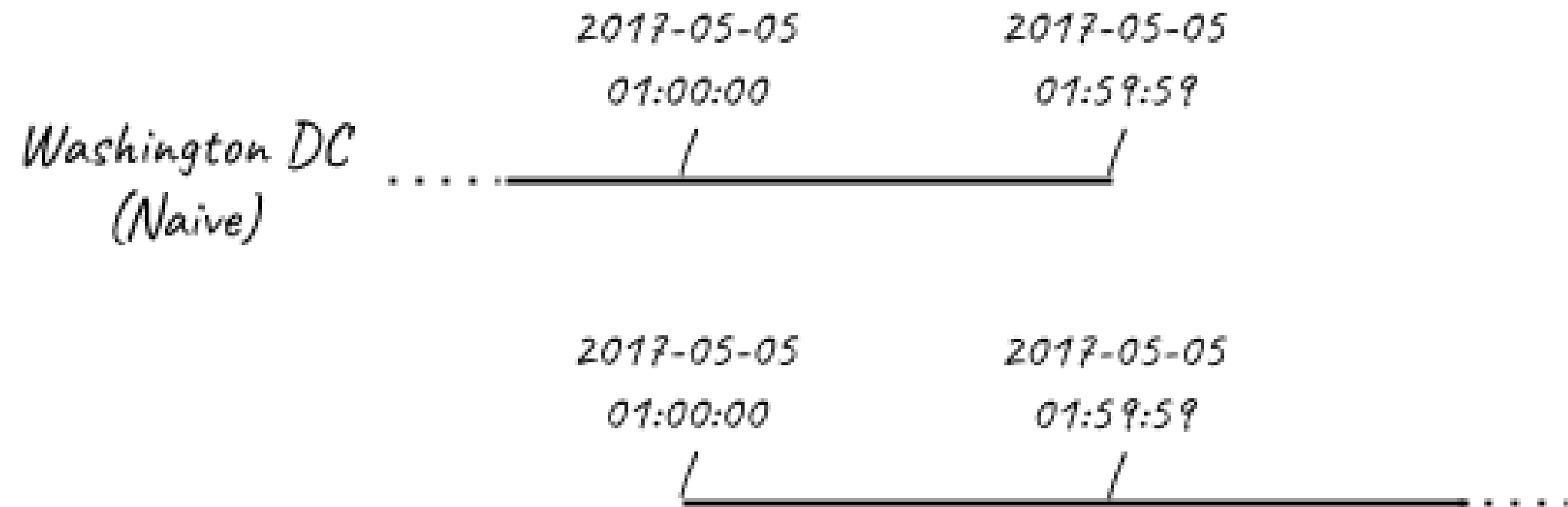
WORKING WITH DATES AND TIMES IN PYTHON

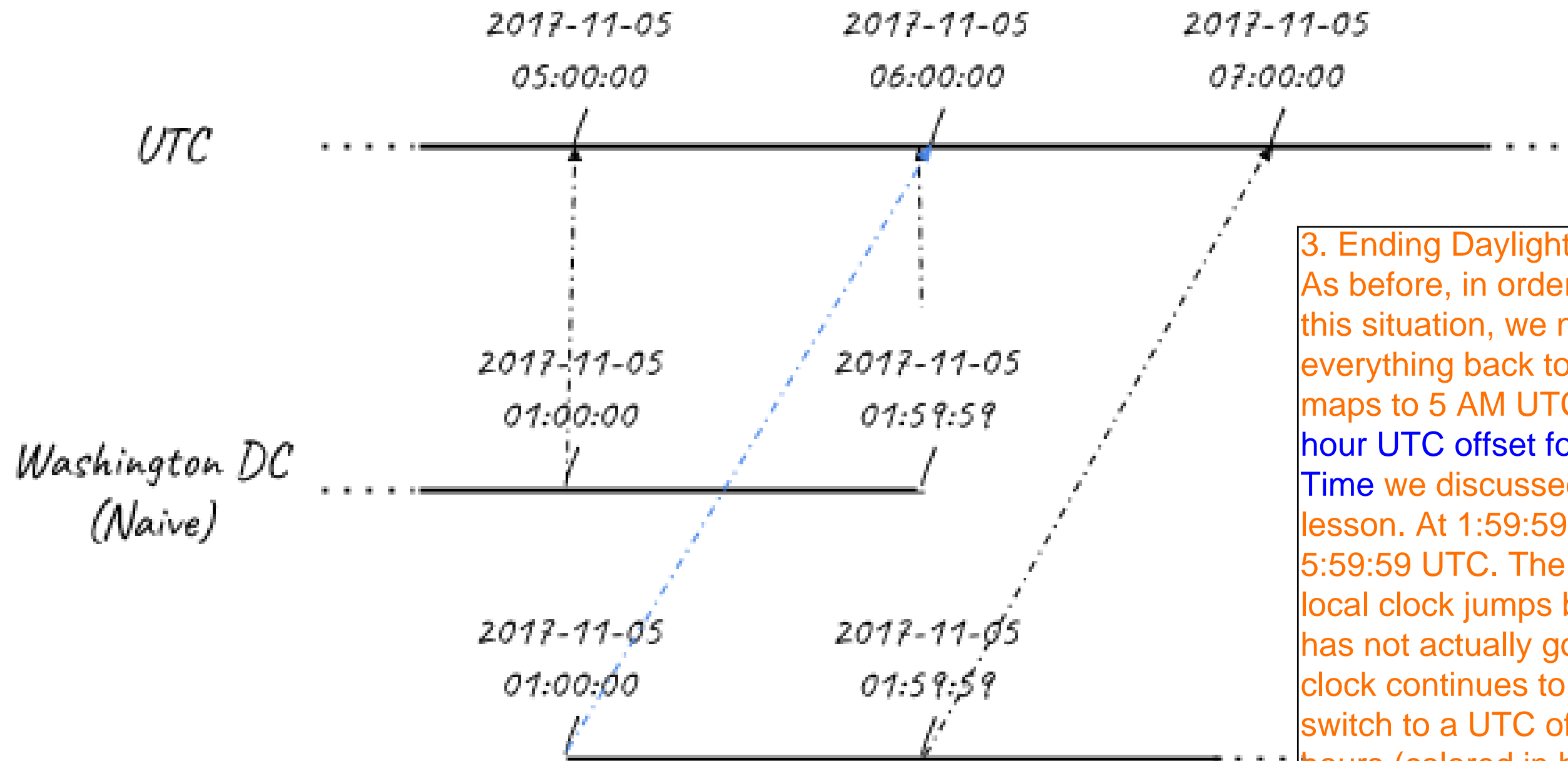


**Max Shron**

Data Scientist and Author

2. Ending Daylight Saving Time  
Let's look back at our example in Washington, D.C., on the day that daylight saving time ended. On November 5th, 2017, at 2 AM the clocks jumped back an hour. That means there were two 1 AMs! We've represented this by "folding" over our timeline to show the repeat.





**3. Ending Daylight Saving Time**  
As before, in order to make sense of this situation, we need to map everything back to UTC. The first 1 AM maps to 5 AM UTC. This is the minus 4 hour UTC offset for Eastern Daylight Time we discussed in the previous lesson. At 1:59:59 local time, we're at 5:59:59 UTC. The next moment, our local clock jumps back, but since time has not actually gone backward, the clock continues to tick in UTC. We switch to a UTC offset of minus 5 hours (colored in blue), and the second 1 AM corresponds to 6 AM UTC.

# Ending Daylight Saving Time

```
eastern = tz.gettz('US/Eastern')  
# 2017-11-05 01:00:00  
first_1am = datetime(2017, 11, 5, 1, 0, 0,  
                      tzinfo = eastern)  
tz.datetime_ambiguous(first_1am)
```

True

```
# 2017-11-05 01:00:00 again  
second_1am = datetime(2017, 11, 5, 1, 0, 0,  
                       tzinfo = eastern)  
second_1am = tz.fold(second_1am)
```



# Ending Daylight Saving Time

```
(first_1am - second_1am).total_seconds()
```

```
0.0
```

```
first_1am = first_1am.astimezone(tz.UTC)
second_1am = second_1am.astimezone(tz.UTC)
(second_1am - first_1am).total_seconds()
```

```
3600.0
```

## 6. Ending Daylight Saving Time

We've covered how to handle springing forward and falling back, both with hand-coded UTC offsets and with `dateutil`. Python often tries to be helpful by glossing over daylight saving time difference, and oftentimes that's what you want. However, when you do care about it, use `dateutil` to set the timezone information correctly and then switch into UTC for the most accurate comparisons between events.

# Ending Daylight Saving Time

WORKING WITH DATES AND TIMES IN PYTHON