|  |  |
| --- | --- |
| time.time() | 返回当前系统时间戳 |
| time.ctime([sec]) | 返回当前系统时间的字符串  -->传入时间戳时，返回该时间戳对应的时间 |
| time.gmtime([sec]) | 返回***0时区***时间元组  -->传入时间戳时，返回改时间戳对应的***0时区***下的时间元组 |
| time.localtime(sec) | 返回***当前时区***时间元组  -->传入时间戳时，返回改时间戳对应的***当前时***区下的时间元组 |
| time.mktime(time\_struct\_obj) | 根据传入的time\_struct\_obj，返回时间戳  P.s 根据同样的时间戳，使用time.gmtime以及time.localtime所得到的time\_struct不一样。再使用time.mktime再返回所得到的时间戳也不一样。这是因为time\_sturct没有时区，等于time.mktime接受的time\_struct根本不是同一个时间，所以返回的时间戳也不一样。 |
| time.sleep(secs) | 让程序挂起指定的秒数 |
| time.strftime(format, struct\_obj) | 传入struct\_obj并且按一定格式输出。 |
| time.strptime(str,format) | 传入时间字符串格式转为struct\_obj格式 |

Time 模块运用

Datetime模块

|  |  |
| --- | --- |
| time.time() | 返回当前系统时间戳 |
| time.ctime([sec]) | 返回当前系统时间的字符串  -->传入时间戳时，返回该时间戳对应的时间 |
| time.gmtime([sec]) | 返回***0时区***时间元组  -->传入时间戳时，返回改时间戳对应的***0时区***下的时间元组 |
| time.localtime(sec) | 返回***当前时区***时间元组  -->传入时间戳时，返回改时间戳对应的***当前时***区下的时间元组 |
|  | 根据传入的time\_struct\_obj，返回时间戳  P.s 根据同样的时间戳，使用time.gmtime以及time.localtime所得到的time\_struct不一样。再使用time.mktime再返回所得到的时间戳也不一样。这是因为time\_sturct没有时区，等于time.mktime接受的time\_struct根本不是同一个时间，所以返回的时间戳也不一样。 |
| time.sleep(secs) | 让程序挂起指定的秒数 |
| time.strftime(format, struct\_obj) | 传入struct\_obj并且按一定格式输出。 |
| time.strptime(str,format) | 传入时间字符串格式转为struct\_obj格式 |

|  |  |  |  |
| --- | --- | --- | --- |
| datetime.date.today() | | 返回今日日期的date对象 | |
| datetime.datetime.now() | | 返回当前时区的此时此刻的datetime对象  如果使用datetime.datetime.now()则返回naïve类型  如果使用datetime.datetime.now([时区])则返回aware类型 | |
| datetime对象. strftime(“…”) | | 返回格式化输出时间字符串 | |
| datetime.datetime.strptime(时间字符串，格式) | | 返回该时间字符串对应的datetime对象 | |
| datetime\_obj.timetuple() | | 从datetime\_obj中返回一个时间元组类型 | |
| datetime\_obj - datetime.timedelta([days,hours,minutes…]) | | | 通过datetime.timedelta构建timedelta类型，然后使用对datetime\_obj可以进行加减时间 |
| datetime\_obj 之间比较 [ > < =] | | | |
| datetime.replace(tzinfo=pytz.timezone(“xxx”)) | 将datetime的时区强制变为另一个时区而不改变时间，或者为原本 native的datetime指定时区 | | |
| datetime.astimezone(tz=pytz.timezone(“xxx”) | 将aware或native的 datetime的时区转化为指定时区下的时间  如果原来是aware的则Python根据原时区和指定新时区不同进行转化  如果原来是native的则Python认为native的时区就是系统的时区然后根据指定新时区不同进行转化(3.6才支持，否则datetime对象波许是aware的) | | |
| datetime.tzinfo | 打印出datetime的tzinfo信息，同时也是判断是否为naïve或aware的方式  如果为naïve则显示为空  如果是aware则显示时区信息。 | | |
| Datetime.fromtimstamp(时间戳) | 根据时间戳，返回datetime对象。该datetime对象是native的，并且以机器本地时区作为依据进行转换。 | | |

TimeDelta对象

|  |  |
| --- | --- |
| .total\_seconds() | 时间间隔总秒数 |

P.s必须同为naïve或者同为aware的datetime才可以相互计算。

Pytz模块

|  |  |
| --- | --- |
| pytz.timezone(“时区名) | 返回时区 |
| pytz.timezeon(“时区名”).localize(datetime) | 接受一个native datetime对象，不改变时间, 返回该时区的aware timezone datetime对象, |

Note:

1. UTC为UTC为世界标准时间，GMT为格林威治时间​（Greenwich Mean Time，简称G.M.T.）GMT时间与UTC时间相差不超过0.9秒，达到一致，如果将要超过0.9秒则通过闰秒抵消。
2. 在Python中通常有三种方式表达时间，时间戳，格式化的时间字符串，时间元组(struct\_time)
3. 时间戳(timestamp)代表从1970年1月1日0时，到现在为止的秒数。（1970年1月1日是Unix系统开始应用的时候）

时间戳均已UTC时间作为基准，在任何时区取得时间戳都一样。并且由于这个特性用于服务器之间的时间交流。

一样的时间戳，根据时区的不同，返回的时间也就不一样。

1. 时间元组struct\_time,里面含有9个元素。

|  |  |
| --- | --- |
| tm\_year | 年 |
| tm\_mon | 月(1-12) |
| tm\_mday | 日(1-31) |
| tm\_hour | 小时(0-23) |
| tm\_min | 分(0-59) |
| tm\_sec | 秒(0-59) |
| tm\_wday | 周几(0-6) |
| tm\_yday | 一年中的第几天 |
| tm\_isdsy | 是否为夏令时 |

P.s ***注意时间元组里面没有时区。***

1. datetime 在Python中分为两种类型。第一种是 offset-native就是没有设置时区的datetime, 另外一种是offset-aware设置时区的datetime. 同一种类型的datetime才可以进行比较。

使用datetime\_obj.replace函数可以将一个datetime\_obj 改成offset-native或者offset-aware

6. 常用的操作：

从字符串转为datetime对象：

datestring = “2019-07-08 10:28:00”

dd = datetime.datetime.strptime(datestring, "%Y-%m-%d %H:%M:%S")

为没有设置时区的datetime对象设置时区， 设置以后该datetime为aware的并且时区变为指定的时区下的该时间：

tz = pytz.timezone("Canada/Eastern")

dd = dd.replace(tzinfo=tz) #加拿大东部时区下的“2019-07-08 10:28:00”

dd = dd.replace(pytz.timezone("UTC")) # UTC时区下的“2019-07-08 10:28:00”

7.

LMT: Local Mean Time 当地日出时间

DST: Daylight Saving Time

EST: Eastern Standard Time 东部标准时间，使用在冬天

EDT: Easstern Daylight Time 使用在春，夏，秋

8. ***Datetime中tzinfo关键字参数设置时区和pytz的不兼容性***

原文：Unfortunately using the tzinfo argument of the standard datetime constructors ‘’does not work’’ with pytz for many timezones. 使用datetime构造器的关键字参数tzinfo和使用pytz够构建的在许多时区会不统一。如果该时区包含DST则无法统一。

例如：对于时间字符串2019-09-15 13:54:24

使用datetime的关键字来构建的时候：

test = “2019-09-15 13:54:24”

test\_datetime = datetime.strptime(test, "%Y-%m-%d %H:%M:%S")

test\_datetime = test\_datetime.replace(tzinfo=pytz.timezone("Canada/Eastern"))

print(test\_datetime) # 为2019-09-15 13:54:24-05:18， timezone为<DstTzInfo 'Canada/Eastern' LMT-1 day, 18:42:00 STD>

使用pytz来构建的时候：

from pytz import timezone

test = “2019-09-15 13:54:24”

test\_datetime = datetime.strptime(test, "%Y-%m-%d %H:%M:%S")

eastern = timezone('Canada/Eastern)

print(eastern.localize(test\_datetime)) # 为2019-09-15 13:54:24-04:00, timezone为tzinfo=<DstTzInfo 'Canada/Eastern' EDT-1 day, 20:00:00 DST>

基于此，Django在USE\_TZ=True的情况下所进行时间区转化是根据pytz模块的，所以如果自己手动的使用datetime中关键字tzinfo来进行改变时区进行构建，很有可能于Django（基于pytz）对同一时区的时间转化有所不同。

9. Python时间格式符号

* %y 两位数的年份表示（00-99）
* %Y 四位数的年份表示（000-9999）
* %m 月份（01-12）
* %d 月内中的一天（0-31）
* %H 24小时制小时数（0-23）
* %I 12小时制小时数（01-12）
* %M 分钟数（00-59）
* %S 秒（00-59）
* %a 本地简化星期名称
* %A 本地完整星期名称
* %b 本地简化的月份名称 （例如九月Sep）
* %B 本地完整的月份名称 （例如九月September）
* %c 本地相应的日期表示和时间表示
* %j 年内的一天（001-366）
* %p 本地A.M.或P.M.的等价符
* %U 一年中的星期数（00-53）星期天为星期的开始
* %w 星期（0-6），星期天为星期的开始
* %W 一年中的星期数（00-53）星期一为星期的开始
* %x 本地相应的日期表示
* %X 本地相应的时间表示
* %Z 当前时区的名称
* %% %号本身