

```

from typing import Union
from math import floor

class Date():
    """Represents a year, month, and day"""
    def __init__(self, year, month, day) -> None:

        # Flag to identify if the year is BC
        self.bc = False

        # Dictionary to indicate allowable months and their number of days
        self.MONTH_MAPPING = {
            "January":31,
            "February":28,
            "March":31,
            "April":30,
            "May":31,
            "June":30,
            "July":31,
            "August":31,
            "September":30,
            "October":31,
            "November":30,
            "December":31
        }

        # Call the make date function to construct the date attributes
        self.make_date(year, month, day)

    def _validate_year(self, year) -> None:
        """
        Validate the year input.

        This function will assign the self.year attribute and identify if the
        year is a leap year.

        Additionally, it will set the self.bc flag if the year is less than 0
        """

        # Per the article, below, the first time a leap year was instituted
        # was 46 B.C.
        # https://billpetro.com/history-of-leap-year/
        if int(year) < -46:
            #This is a BC year
            self.bc = True
            #Get the absolute value of the year. For example, the year will be
            #represented as 500 B.C. instead of -500
            self.year=abs(year)
        elif int(year) < 0:
            self.bc = True
            #Get the absolute value to remove error in leap year calculation
            self.year=abs(year)
            self._is_leap(self.year)
        else:
            self.bc = False
            #assign the year
            self.year=year

```

```

        # Check to see if the year is a leap year. This will reassign the
        # number of days in February
        self._is_leap(year)

def _validate_month(self, month) -> None:
    """
    Validate the month based on the rules defined in make_date().

    This function will perform the self.month attribute assignment if the
    month is valid.
    """
    try:
        # Use the month type to determine how to process it...
        if type(month)==int:
            # Error handling for invalid months
            if month > 12 or month < 1:
                raise Exception("ERROR: Invalid Month.")
            # The dictionary will retain it's insertion order, so index
            # into it and get the key where month = index+1
            else:
                for index, key in enumerate(self.MONTH_MAPPING.keys()):
                    if month == index+1:
                        self.month=key
                        break

            # Raise an exception if the month key does not return a value,
            # otherwise assign it to the month attribute
        elif type(month)==str:
            if not self.MONTH_MAPPING[f"{month}"]:
                raise Exception("ERROR: Invalid Month.")
            else:
                self.month=month

    except Exception as e:
        print(f"{e}")

def _validate_day(self, day) -> None:
    """
    Validate the day input.

    This function will perform the self.day attribute assignment if the
    day is valid.
    """
    try:
        if day > self.MONTH_MAPPING[self.month] or day < 1:
            raise Exception("ERROR: Invalid Day.")
        else:
            self.day = day

    except Exception as e:
        print(f"{e}")

def _is_leap(self, year) -> None:
    """
    Verify whether the year is a leap year.

```

```

"""

# No year that is not divisble by 4 can be a leap year
if year%4:
    self.MONTH_MAPPING["February"] = 28
# If the year is divisible by 4, it could still not be a leap year
# because of the turn of the century
elif not year%4:
    # Every 4 centuries will be a leap year
    if not year%400:
        self.MONTH_MAPPING["February"] = 29
    # If the year is divisible by 100, and not 400, it is not a leap year
    elif not year%100:
        self.MONTH_MAPPING["February"] = 28
    # All other cases where not year%4 are leap years
    else:
        self.MONTH_MAPPING["February"] = 29

def date_to_int(self) -> int:
    """
    Convert the current date to an integer that represents the number of
    days since day 0 for date comparison.
    """
    if self.bc:
        year_to_days = (self.year*365)-floor(self.year/4)
        month_to_days = self._get_days_from_month()
        return -year_to_days-month_to_days-self.day
    else:
        year_to_days = (self.year*365)+floor(self.year/4)
        month_to_days = self._get_days_from_month()
        return year_to_days+month_to_days+self.day

def _get_days_from_month(self):
    days=0
    for key, value in self.MONTH_MAPPING.items():
        days+=value
    if key==self.month:
        return days

def make_date(self, year:int, month:Union[int, str], day:int) -> str:
    """
    Takes in a year, month, and day and produces an equivalent date string.

    Month supports both integer and string inputs.

    Strings must be the full month name with the first letter capitalized.
    For example: February

    NOTE:
    The validation order is critical, and must occur in year, month, day
    order. Both the year and month must be valid before checking for leap
    year. If it is a leap year, February will have an additional day, which
    will be validated against the self.MONTH_MAPPING dictionary.

    """

    # NOTE: This is how the question wants me to do this...

```

```

self.year=year
self.month=month
self.day=day

"""
NOTE: This is me being fancy, I apologize in advance. The validation
functions are used to implement error checking for the respective
inputs. Also, this allows for accurate date checking with leap years
accounted for.
"""

# Validate the year input and assign it to the self.year attribute
self._validate_year(year)

# Validate the month input and assign it to the self.month attribute
self._validate_month(month)

# Now that the self.MONTH_MAPPING dictionary contains the correct
# number of days for February, validate the day input and assign the
# self.day attribute
self._validate_day(day)

def print_date(self) -> str:
    """
    Take in a date object and print the date. If the self.bc flag is set,
    append a BC tag to the date.
    """
    if self.bc:
        print(f"{self.year}-{self.month}-{self.day} BC")
    else:
        print(f"{self.year}-{self.month}-{self.day}")

def is_after(date1:Date, date2:Date) -> bool:
    """
    Determine if the first date comes after the second.
    """
    if date1.date_to_int() > date2.date_to_int():
        return True
    else:
        return False

```