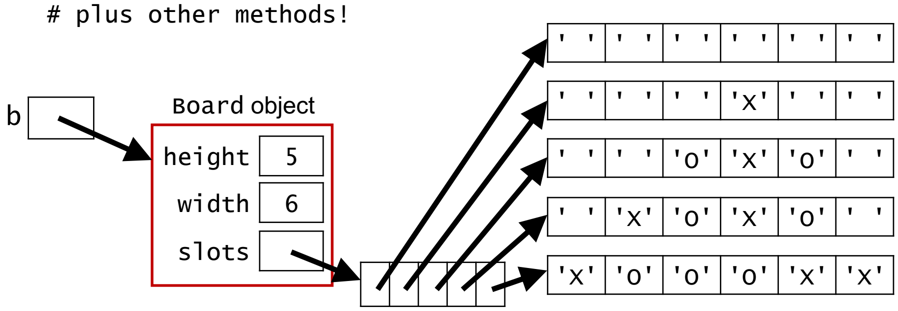
More Objects for Connect Four; Inheritance

Computer Science 111
Boston University

Vahid Azadeh-Ranjbar, Ph.D.

Recall: `Board` Class for Connect Four

```
class Board:
    def __init__(self, height, width):
        ...

    def __repr__(self):
        ...




    # plus other methods!
```



```
' ' ' ' ' ' ' ' ' ' ' '
' ' ' ' ' ' 'X' ' ' ' '
' ' ' ' 'O' 'X' 'O' ' '
' ' 'X' 'O' 'X' 'O' ' '
'X' 'O' 'O' 'O' 'X' 'X'
```

add_checker Method

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

- Why don't we need a `return` statement?
  - `add_checker()`'s only purpose is to change the state of the `Board`
  - when a method changes the internals of an object, those changes will still be there after the method completes
  - thus, no return is needed!

Which of these correctly fills in the blank?

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

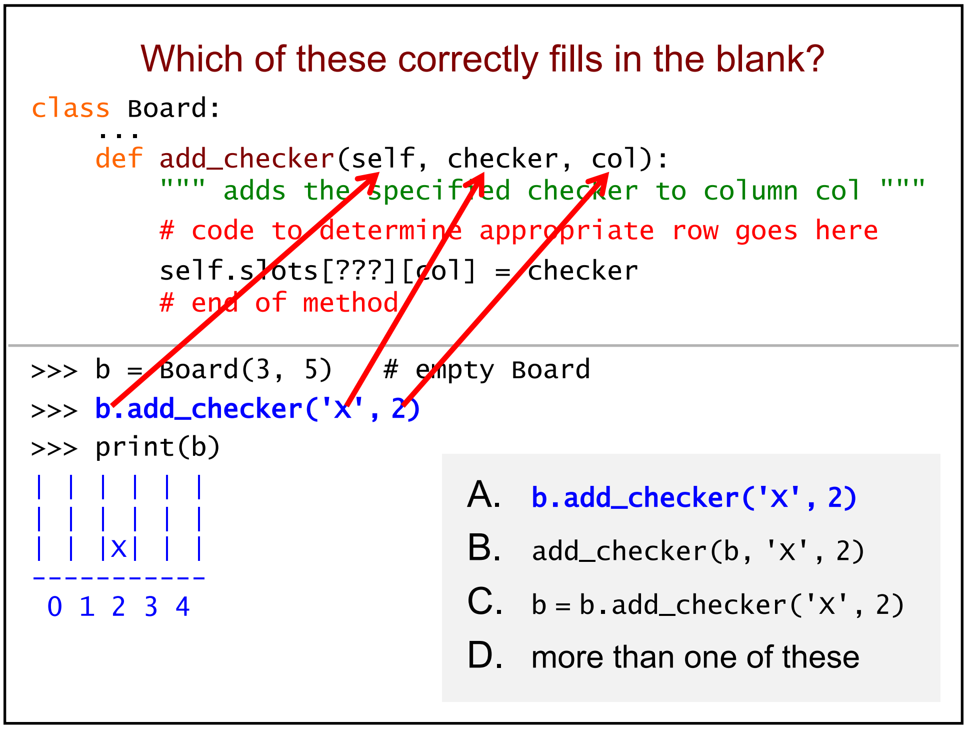
```
>>> b = Board(3, 5)    # empty Board
>>> ________________   # add 'X' to column 2
>>> print(b)
| | | | | |
| | | | | |
| | |X| | |
-----------
 0 1 2 3 4
```

A. `b.add_checker('X', 2)`

B. `add_checker(b, 'X', 2)`

C. `b = b.add_checker('X', 2)`

D. more than one of these



Which of these correctly fills in the blank?

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

```
>>> b = Board(3, 5)    # empty Board
>>> b.add_checker('X', 2)
>>> print(b)
| | | | | |
| | | | | |
| | |X| | |
-----------
 0 1 2 3 4
```

A. `b.add_checker('X', 2)`

B. `add_checker(b, 'X', 2)`

C. `b = b.add_checker('X', 2)`

D. more than one of these

Which of these correctly fills in the blank?

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # end of method
```

```
>>> b = Board(3, 5)    # empty Board
>>> b.add_checker('X', 2)
>>> print(b)
| | | | | |
| | | | | |
| | |X| | |
-----------
 0 1 2 3 4
```

A. `b.add_checker('X', 2)`

B. `add_checker(b, 'X', 2)`
`NameError`

C. `b = b.add_checker('X', 2)`

Which of these correctly fills in the blank?

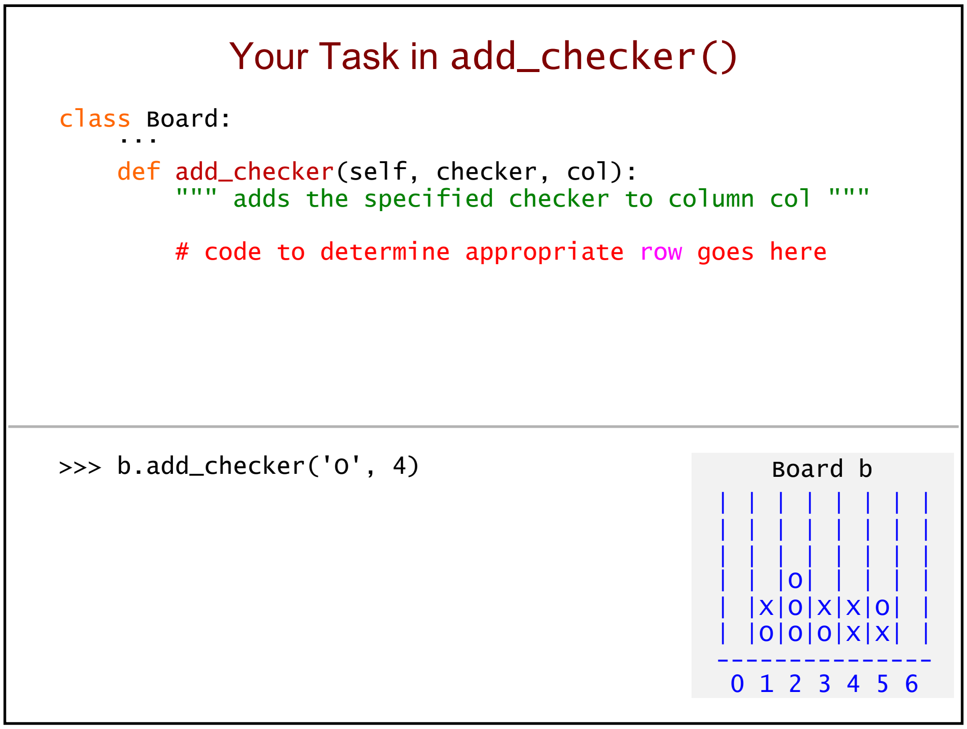
```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """
        # code to determine appropriate row goes here
        self.slots[???][col] = checker
        # no explicit return, so returns None
```

```
>>> b = Board(3, 5)    # empty Board
>>> b.add_checker('X', 2)
>>> print(b)
| | | | | |
| | | | | |
| | |X| | |
-----------
 0 1 2 3 4
```

A. `b.add_checker('X', 2)`

B. `add_checker(b, 'X', 2)`
`NameError`

C.
```
b = b.add_checker('X', 2)
print(b)
None    # no more Board!
```

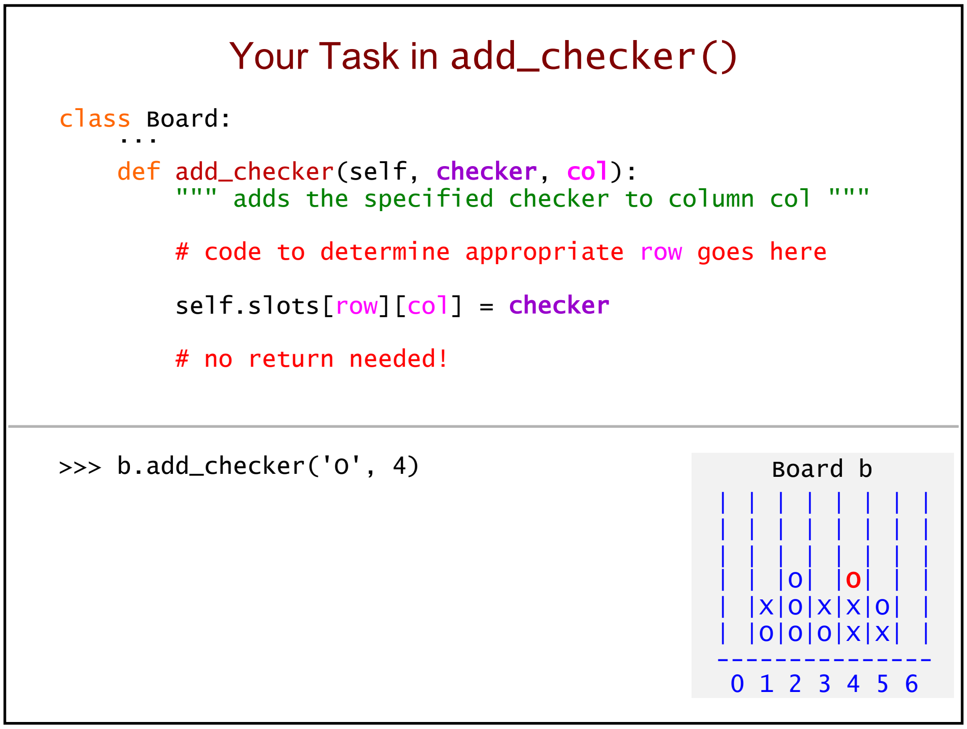


Your Task in add_checker()

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """

        # code to determine appropriate row goes here
```

```
>>> b.add_checker('O', 4)
```

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| | | | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```



Your Task in add_checker()

```
class Board:
    ...
    def add_checker(self, checker, col):
        """ adds the specified checker to column col """

        # code to determine appropriate row goes here

        self.slots[row][col] = checker

        # no return needed!
```

```
>>> b.add_checker('O', 4)
```

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```

Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):   # buggy version!
        """ adds the specified checker to column col """

        row = 0
        while self.slots[row][col] == ' ':
            row += 1

        self.slots[row][col] = checker
```

A. `b.add_checker('X', 0)`

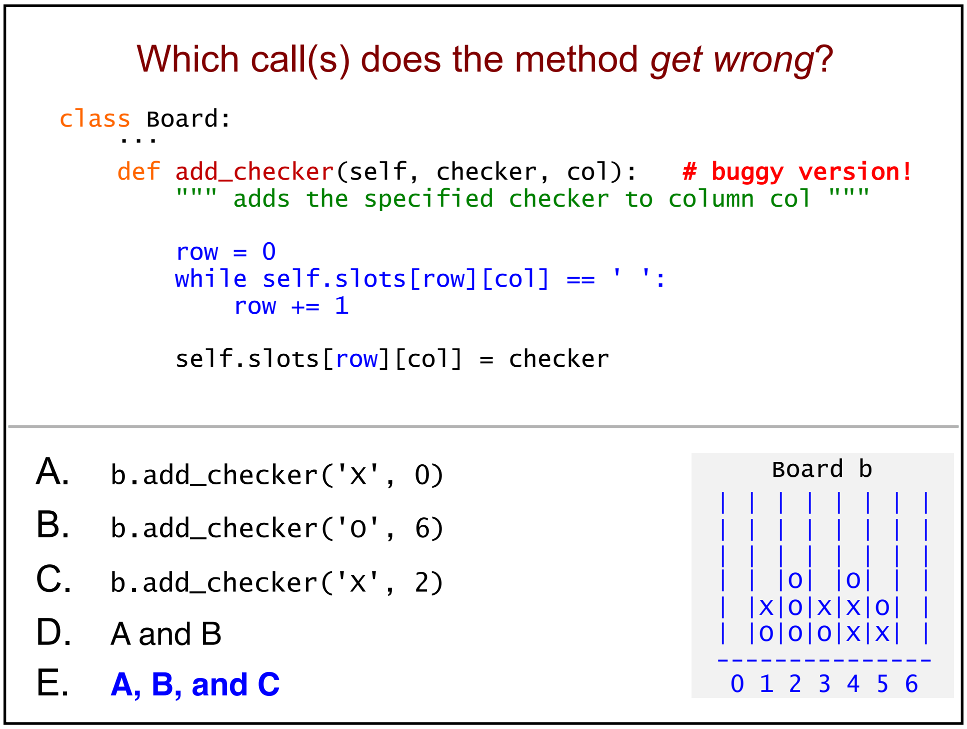
B. `b.add_checker('O', 6)`

C. `b.add_checker('X', 2)`

D. A and B

E. A, B, and C

```
   Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```



Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):    # buggy version!
        """ adds the specified checker to column col """

        row = 0
        while self.slots[row][col] == ' ':
            row += 1

        self.slots[row][col] = checker
```

A. `b.add_checker('X', 0)`

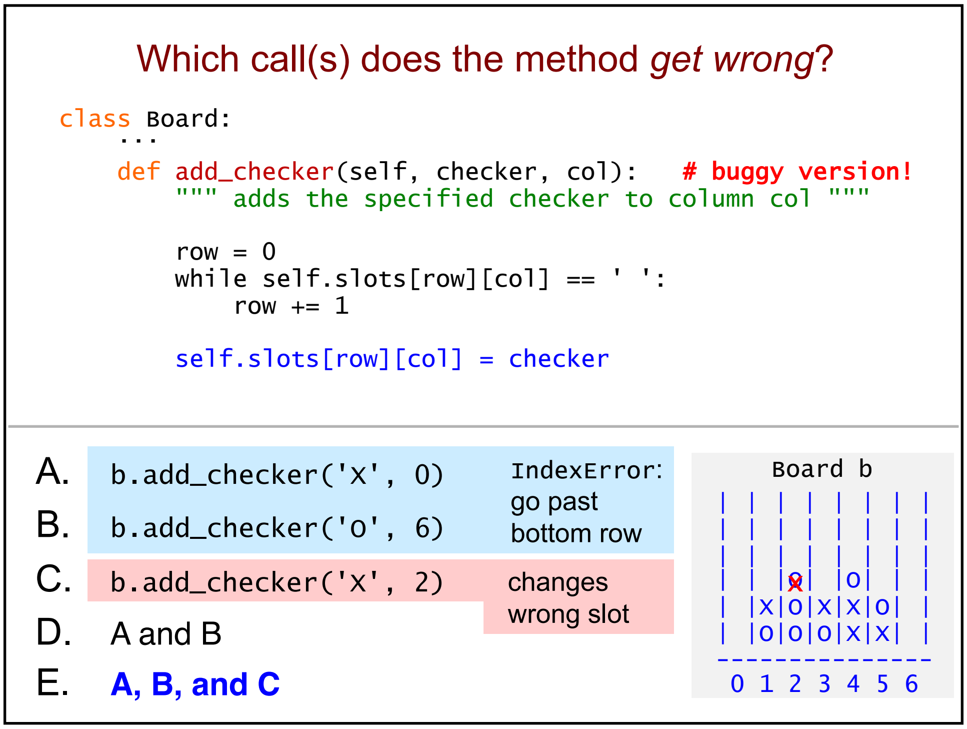
B. `b.add_checker('O', 6)`

C. `b.add_checker('X', 2)`

D. A and B

E. **A, B, and C**

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |O| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```



Which call(s) does the method *get wrong*?

```
class Board:
    ...
    def add_checker(self, checker, col):    # buggy version!
        """ adds the specified checker to column col """

        row = 0
        while self.slots[row][col] == ' ':
            row += 1

        self.slots[row][col] = checker
```

A. `b.add_checker('X', 0)`

B. `b.add_checker('O', 6)`

`IndexError`: go past bottom row

C. `b.add_checker('X', 2)`

changes wrong slot

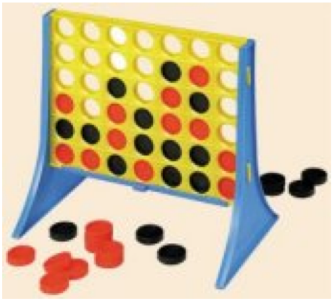
D. A and B

E. **A, B, and C**

```
    Board b
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | |X| |O| | |
| |X|O|X|X|O| |
| |O|O|O|X|X| |
---------------
 0 1 2 3 4 5 6
```

Other objects?

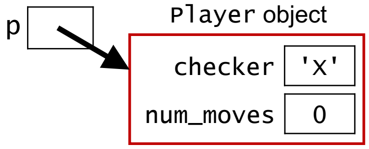
- We made a class (called “board”) to represent the actual board for connect-four game.
- What other objects do we need to play this game?
  - **The players**



Also in PS 9: A `Player` Class

```
class Player:
    def __init__(self, checker):
        ...

    def __repr__(self):
        ...

    def opponent_checker(self):
        ...

    def next_move(self, b):
        """ Get a next move for this player that is valid
            for the board b.
        """
        self.num_moves += 1

        while True:
            col = int(input('Enter a column: '))
            # if valid column index, return that integer
            # else, print 'Try again!' and keep looping
```

p = Player('X')



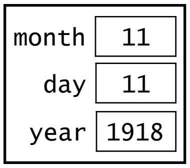
The APIs of Our Board and Player Classes

```
class Board:
   __init__(self, col)
   __repr__(self)
   add_checker(self, checker, col)
   clear(self)
   add_checkers(self, colnums)
   can_add_to(self, col)
   is_full(self)
   remove_checker(self, col)
   is_win_for(self, checker)

class Player:
   __init__(self,col)
   __repr__(self)
   opponent_checker(self)
   next_move(self, b)
```

Make sure to take full advantage of these methods in your work on PS 9!

Recall: Our Date Class

```
class Date:
    def __init__(self, new_month, new_day, new_year):
        """ Constructor """
        self.month = new_month
        self.day = new_day
        self.year = new_year

    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).
        """
        s =  "%02d/%02d/%04d" % (self.month, self.day, self.year)
        return s

    def is_leap_year(self):
        """ Returns True if the calling object is
            in a leap year. Otherwise, returns False.
        """
        if self.year % 400 == 0:
            return True
        elif self.year % 100 == 0:
            return False
        elif self.year % 4 == 0:
            return True
        return False
```

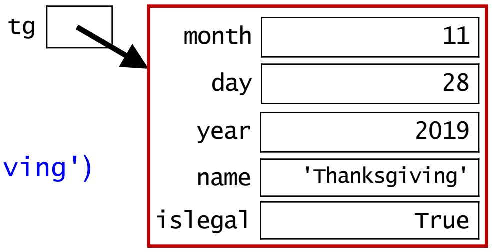

month 11

day 11

year 1918

Holidays == Special Dates!

- Each holiday has:
  - a month
  - a day
  - a year
  - a name (e.g., 'Thanksgiving')
  - an indicator of whether it's a legal holiday



- We want Holiday objects to have Date-like functionality:

```
>>> tg = Holiday(11, 28, 2019, 'Thanksgiving')
>>> today = Date(11, 18, 2019)
>>> tg.days_between(today)
result: 10
```

- But we want them to behave differently in at least one way:

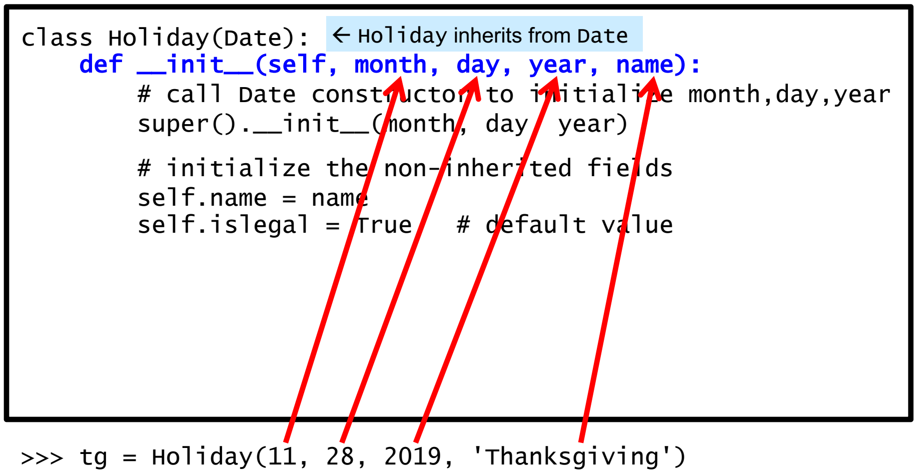
```
>>> print(tg)                       >>> print(today)
Thanksgiving (11/28/2019)           11/18/2019
```

Let Holiday Inherit From Date!

```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        ...
```

- Holiday gets all of the attributes and methods of Date.
  - we don't need to redefine them here!
- Holiday is a *subclass* of Date.
- Date is a *superclass* of Holiday.

Constructors and Inheritance



```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True    # default value
```

```
>>> tg = Holiday(11, 28, 2019, 'Thanksgiving')
```

Constructors and Inheritance

```
class Holiday(Date):   ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True    # default value
```

```
>>> tg = Holiday(11, 28, 2019, 'Thanksgiving')
```

- `super()` provides access to the superclass of the current class.
  - allows us to call its version of `__init__`, which initializes the inherited attributes

Overriding an Inherited Method

```
class Holiday(Date):  ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

- To see something different when we print a `Holiday` object, we *override* (i.e., replace) the inherited version of `__repr__`.

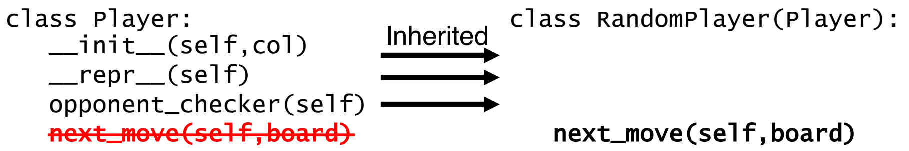
Let `Holiday` Inherit From `Date`!

```
class Holiday(Date):  ← Holiday inherits from Date
    def __init__(self, month, day, year, name):
        # call Date constructor to initialize month,day,year
        super().__init__(month, day, year)

        # initialize the non-inherited fields
        self.name = name
        self.islegal = True   # default value

    def __repr__(self):   # overrides the inherited __repr__
        s = self.name
        mdy = super().__repr__()  # use inherited __repr__
        s += ' (' + mdy + ')'
        return s
```

- That's it! Everything else is inherited!
- All other `Date` methods work the same on `Holiday` objects as they do on `Date` objects!

Inheritance in PS 9

- `Player` – the superclass
  - includes fields and methods needed by all C4 players
  - in particular, a `next_move` method
  - use this class for human players
- `RandomPlayer` – a subclass for an *un*intelligent computer player
  - no new fields
  - overrides `next_move` with a version that chooses at random from the non-full columns



```
class Player:
    __init__(self,col)
    __repr__(self)
    opponent_checker(self)
    ~~next_move(self,board)~~
```

Inherited

```
class RandomPlayer(Player):
```

`next_move(self,board)`

Inheritance in PS 9

- `Player` – the superclass
  - includes fields and methods needed by all C4 players
  - in particular, a `next_move` method
  - use this class for human players
- `RandomPlayer` – a subclass for an *un*intelligent computer player
  - no new fields
  - overrides `next_move` with a version that chooses at random from the non-full columns
- `AIPlayer` – a subclass for an "intelligent" computer player
  - uses AI techniques
  - new fields for details of its strategy
  - overrides `next_move` with a version that tries to determine the best move!

Why AI Is Challenging

Make no mistake about it:
computers process numbers – not symbols.

Computers can only help us to the extent
that we can ***arithmetize*** an activity.

- paraphrasing Alan Perlis

"Arithmetizing" Connect Four

- Our `AIPlayer` assigns a score to each possible move
  - i.e., to each column
- It *looks ahead* some number of moves into the future to determine the score.
  - *lookahead* = # of future moves that the player considers