Lec 18 ~ Classes and Objects...

CS-specific **names**

class, type, user-defined type, template
object, **instance**, **self**, variable, container
method, function
  constructor, initializer, `__init__`
  `__repr__`, printer

CS-specific **topics**

syntax needed to define a **class**
syntax needed to create an **object**
the use of `self` to refer to a specific **object**
+ within the definition of a **class**!

Also!

Midterm exams...
All Python variables are objects...
Examples
+ **Student** class (that we define)
+ **str** class (Python-defined)
+ **Date** class (that we define)

Classes and Objects

An object-oriented programming language allows you to build your **own customized types** of variables.

(1) A *class* is a **type**

(2) An *object* is one such **variable**.

(instance)

There will typically be MANY objects of a single class.

Classes and Objects

Customizing Python

(1) A *class* is a **type**

(2) An *object* is one such **variable**.

(instance)

There will typically be MANY objects of a single class.

Everything in Python is an **object**!

Its capabilities depend on its **class**.

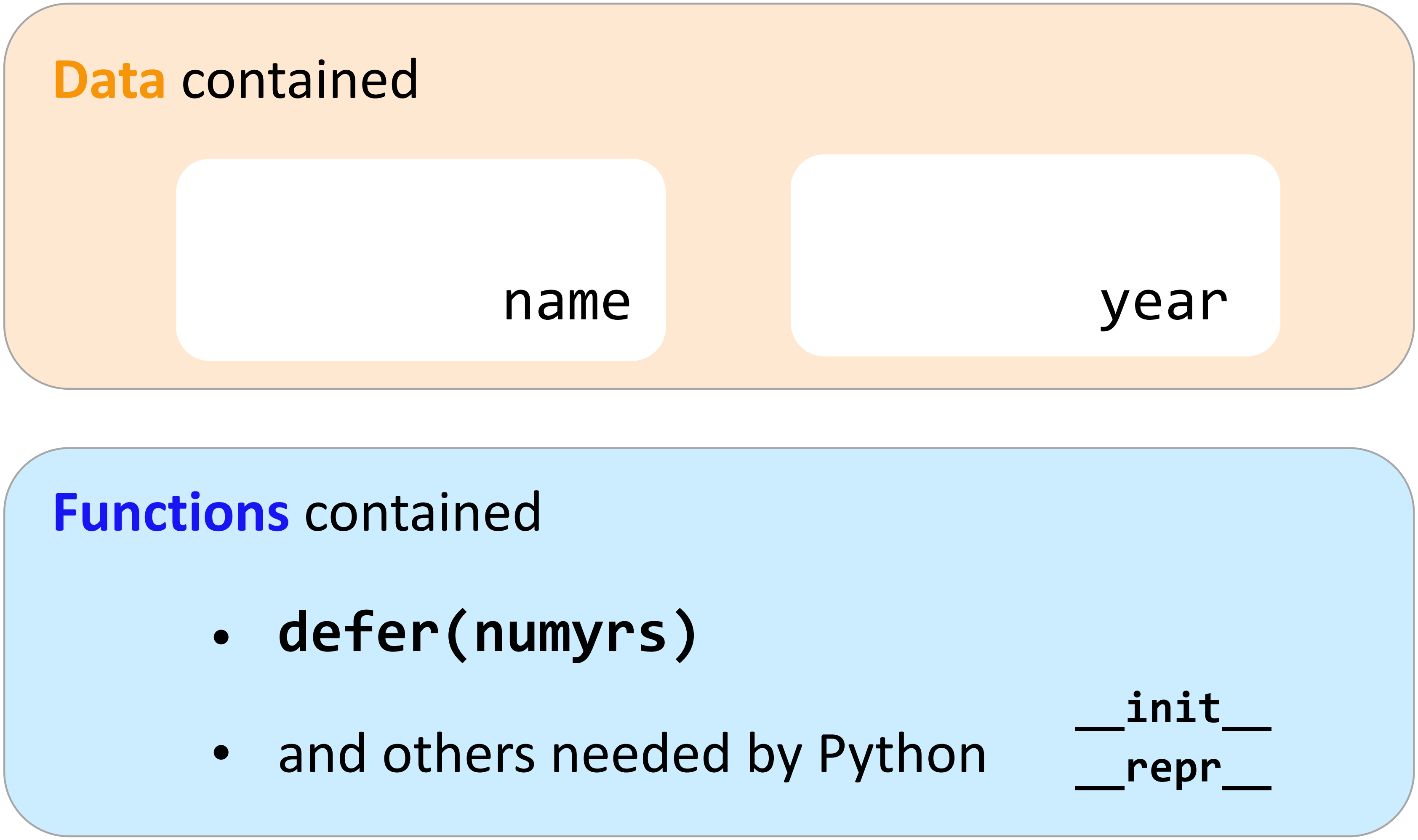
functions

"methods"

type

what's more, you can build your own...

Designing a **`student`** class !



Data contained

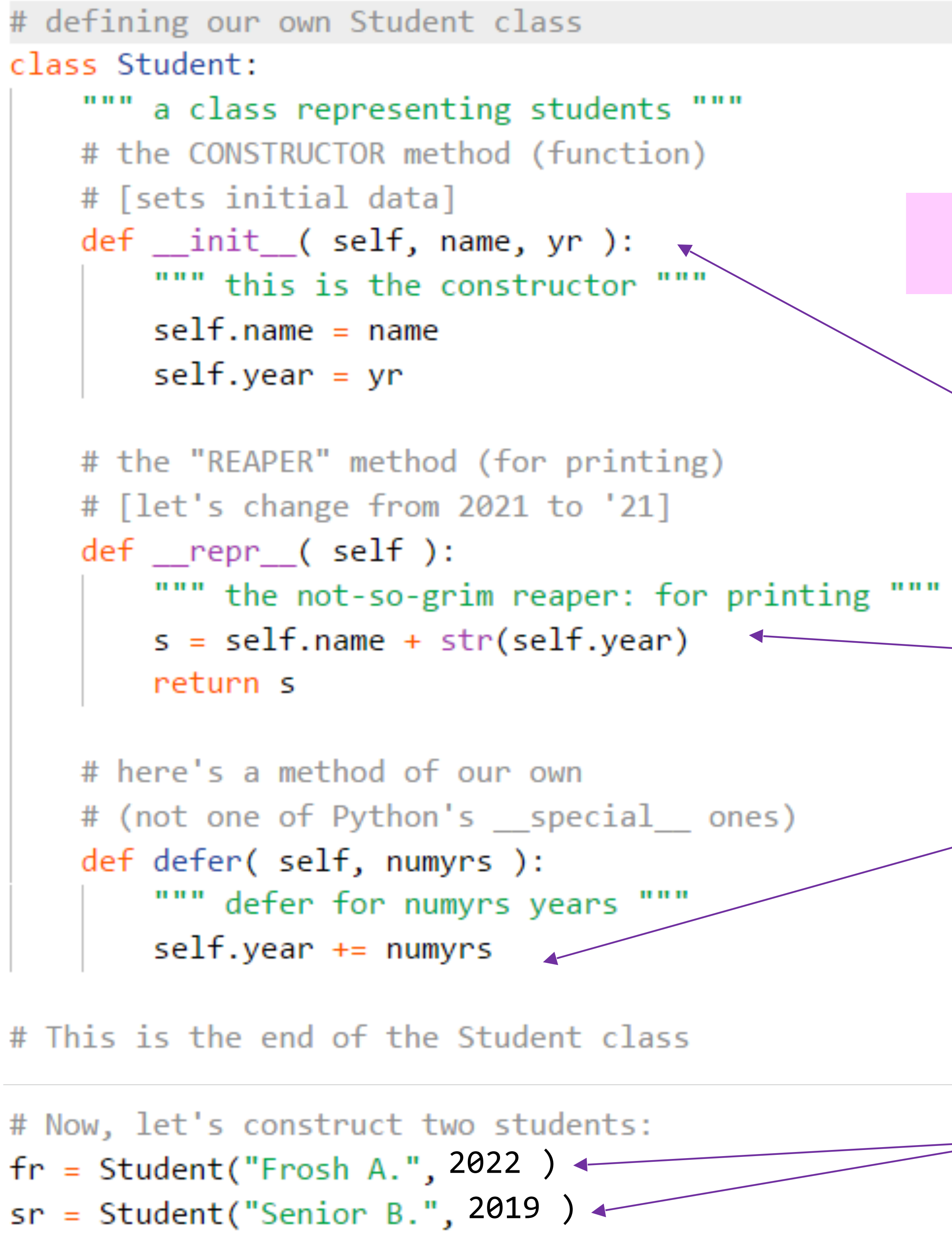
`name`

`year`

Functions contained

- **`defer(numyrs)`**
- and others needed by Python **`__init__`** **`__repr__`**

One-page example



```
# defining our own Student class
class Student:
    """ a class representing students """
    # the CONSTRUCTOR method (function)
    # [sets initial data]
    def __init__( self, name, yr ):
        """ this is the constructor """
        self.name = name
        self.year = yr

    # the "REAPER" method (for printing)
    # [let's change from 2021 to '21]
    def __repr__( self ):
        """ the not-so-grim reaper: for printing """
        s = self.name + str(self.year)
        return s

    # here's a method of our own
    # (not one of Python's __special__ ones)
    def defer( self, numyrs ):
        """ defer for numyrs years """
        self.year += numyrs

# This is the end of the Student class

# Now, let's construct two students:
fr = Student("Frosh A.", 2022 )
sr = Student("Senior B.", 2019 )
```

`Student` is a <u>class</u>

1. constructor, **init**

2. its string **<u>repr</u>**esentation

3. change things via methods

define

use

`fr` and **`sr`** are <u>objects</u>

Everything is an object!

strings, for example:

```
In : s = str( 42 )
```

This calls the **`str`** ***constructor***.

```
In : type(s)
<type 'str'>
```

Shows the type of **`s`** is **`str`**

```
In : dir(s)
```

Shows all of the methods (functions) of **`s`**

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__', '__hash__', '__init__',
'__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition',
'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Let's try some!

Objects

Like a list, an object is a container, but much more customizable:

(1) Its data elements have *names chosen by the programmer*.

(2) An object contains its own functions, called ***methods***

(3) In methods, objects refer to themselves as **`self`**

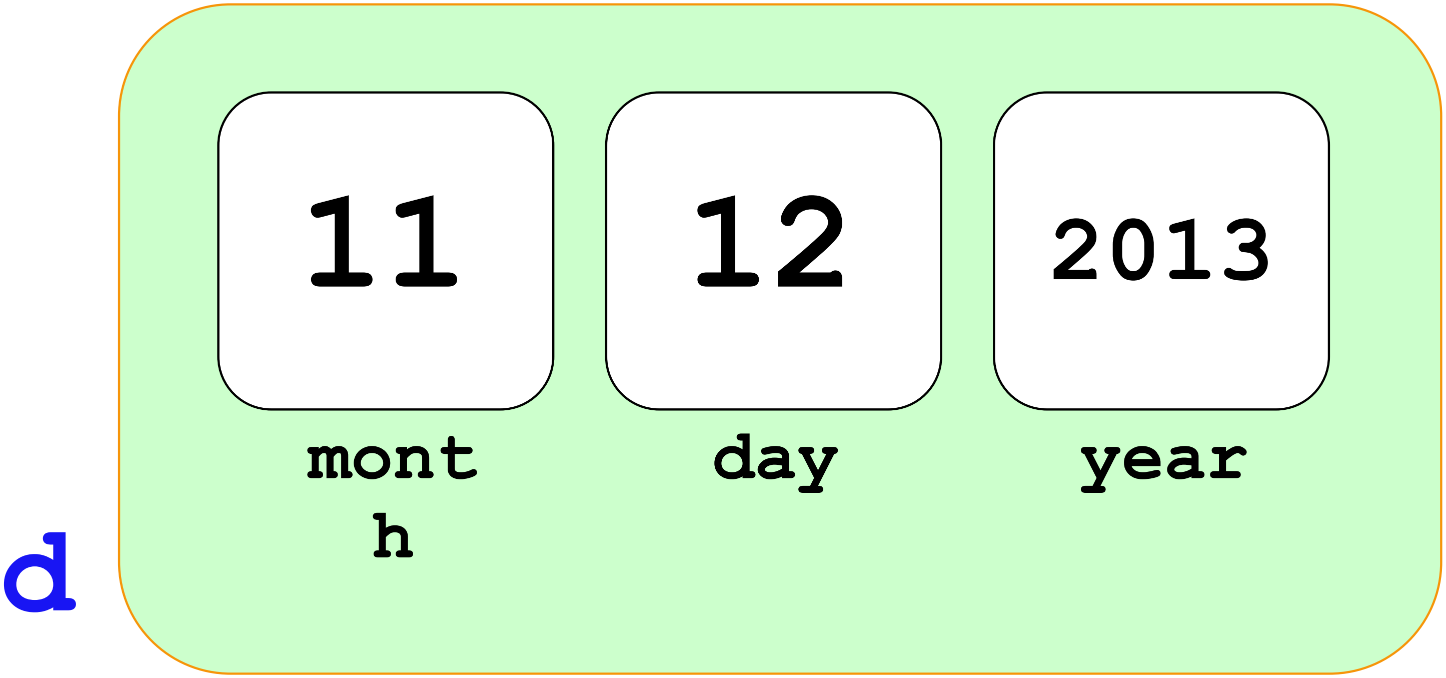
(4) Python signals special methods with two underscores:

`__init__` is called the ***constructor***; it creates new objects

`__repr__` tells Python how to print its objects

I guess we should doubly ***underscore*** *these two methods!*

A Date class and object, d



11

12

2013

mont
h

day

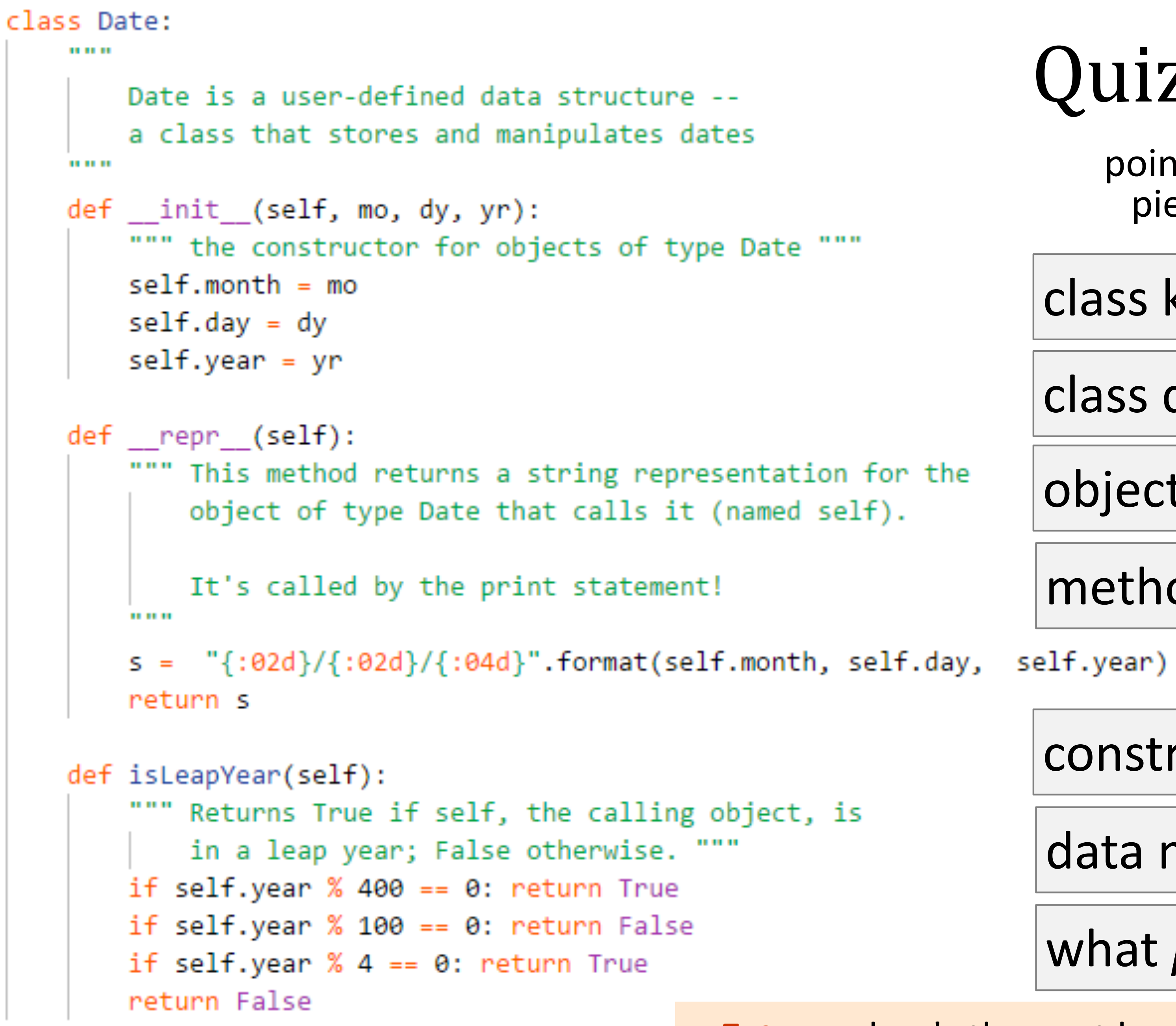
year

d

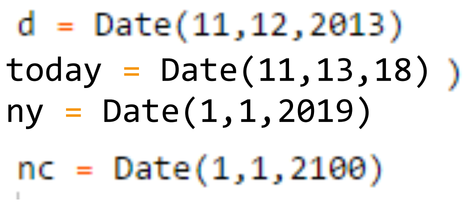
memory location ~ 42042778

Quiz ~ *naming*

point each name to its
piece of the code...



```
class Date:
    """
        Date is a user-defined data structure --
        a class that stores and manipulates dates
    """
    def __init__(self, mo, dy, yr):
        """ the constructor for objects of type Date """
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).

            It's called by the print statement!
        """
        s =  "{:02d}/{:02d}/{:04d}".format(self.month, self.day,  self.year)
        return s

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



```
 d = Date(11,12,2013)
today = Date(11,13,18) )
ny = Date(1,1,2019)
 nc = Date(1,1,2100)
```

class keyword (keyword)

class definition (end)

object creation (4)

methods (3)

constructor

data member (3)

what *prints* Dates?

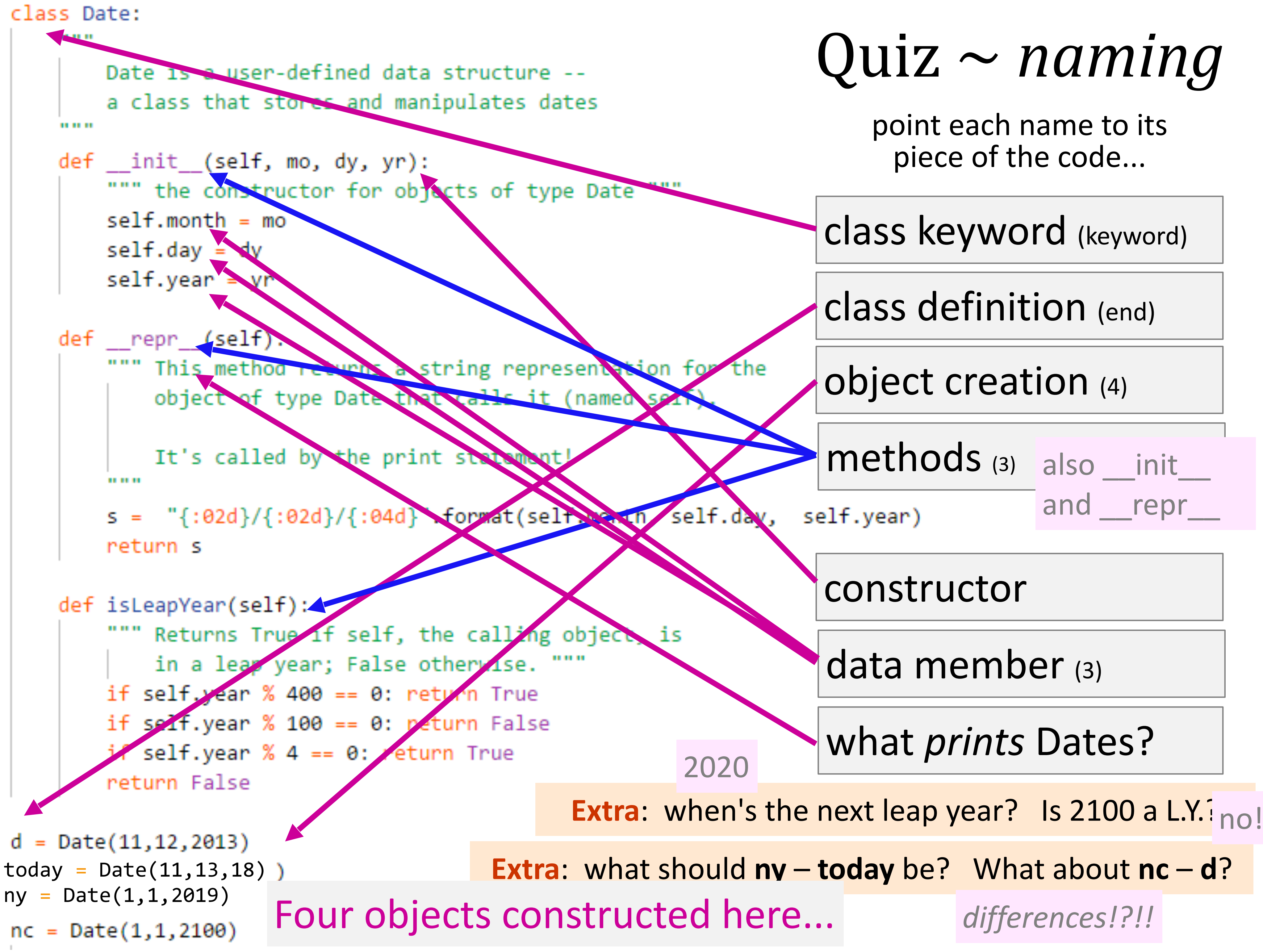
Extra: when's the next leap year? Is 2100 a L.Y.?

Extra: what should **ny – today** be? What about **nc – d**?

Your name(s) ______________________

Quiz ~ *naming*

point each name to its
piece of the code...



```
class Date:
    """
        Date is a user-defined data structure --
        a class that stores and manipulates dates
    """

    def __init__(self, mo, dy, yr):
        """ the constructor for objects of type Date """
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__(self):
        """ This method returns a string representation for the
            object of type Date that calls it (named self).

            It's called by the print statement!
        """
        s =  "{:02d}/{:02d}/{:04d}".format(self.month,  self.day,  self.year)
        return s

    def isLeapYear(self):
        """ Returns True if self, the calling object, is
            in a leap year; False otherwise. """
        if self.year % 400 == 0: return True
        if self.year % 100 == 0: return False
        if self.year % 4 == 0: return True
        return False

d = Date(11,12,2013)
today = Date(11,13,18) )
ny = Date(1,1,2019)
nc = Date(1,1,2100)
```

class keyword (keyword)

class definition (end)

object creation (4)

methods (3)

also __init__ and __repr__

constructor

data member (3)

what *prints* Dates?

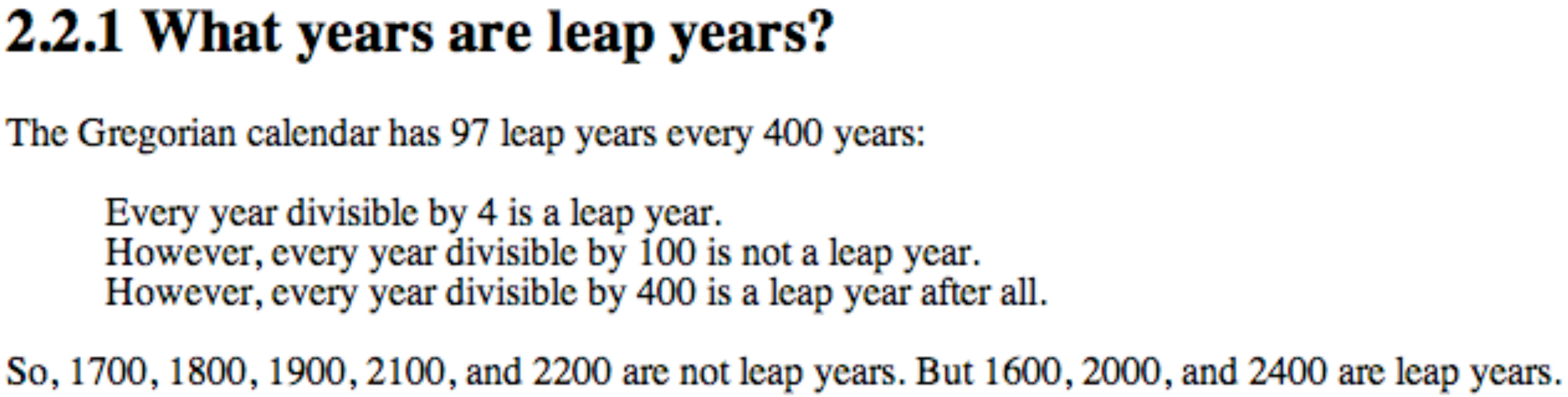
Four objects constructed here...

2020

Extra: when's the next leap year? Is 2100 a L.Y.? no!

Extra: what should **ny – today** be? What about **nc – d**?

differences!?!!



2.2.1 What years are leap years?

The Gregorian calendar has 97 leap years every 400 years:

> Every year divisible by 4 is a leap year.
> However, every year divisible by 100 is not a leap year.
> However, every year divisible by 400 is a leap year after all.

So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years.

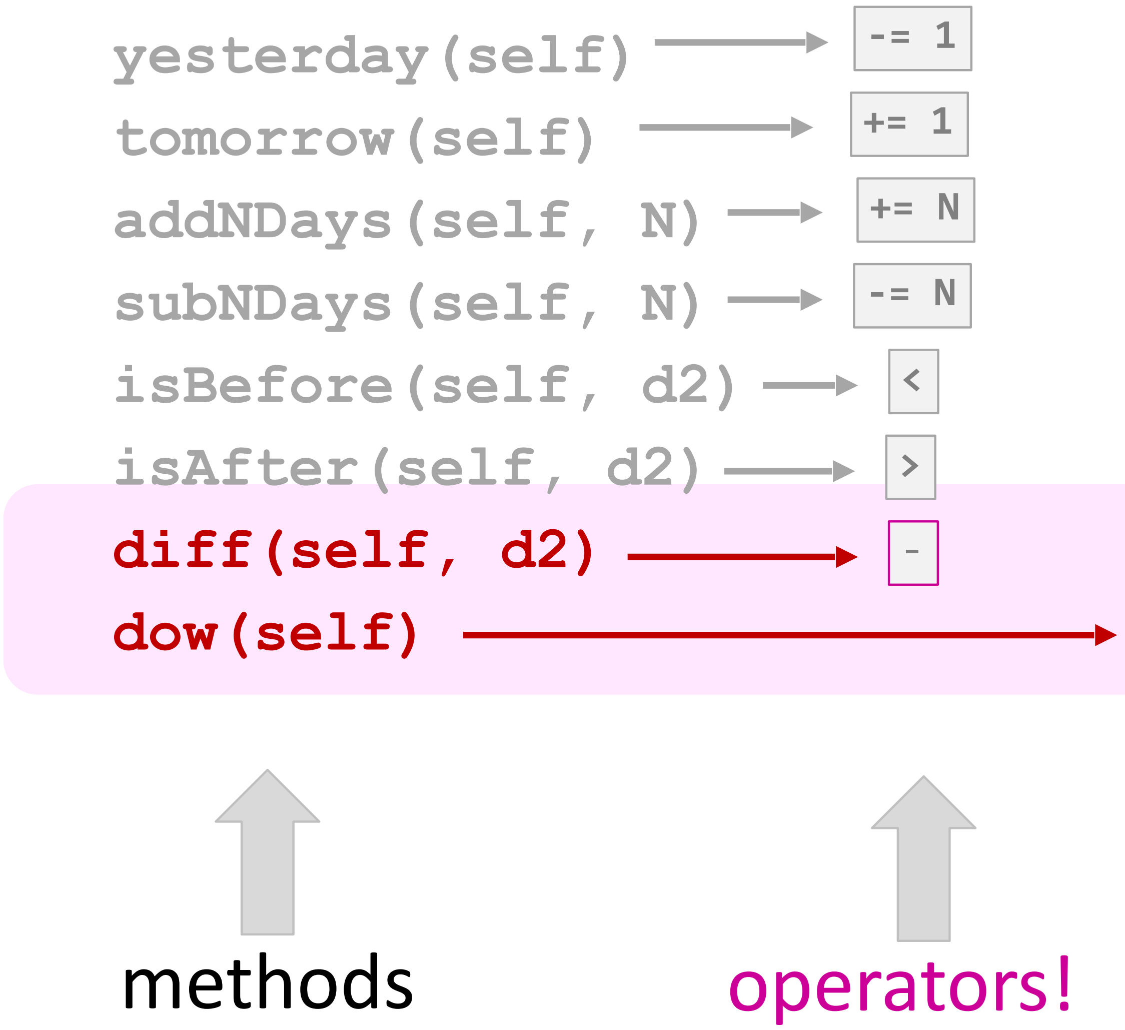
```
class Date:
    def __init__( self, mo, dy, yr ):   (constructor)
    def __repr__(self):   (for printing)

    def isLeapYear( self ):
        """ here it is """
        if self.year%400 == 0: return True
        if self.year%100 == 0: return False
        if self.year%4 == 0: return True
        return False
```

```
In : wd = Date(11,12,2013)

In : wd.isLeapYear()
Out: False
```

```
In : d = Date(1,1,2020)

In : d.isLeapYear()
Out: True
```

hw10pr1

You'll create a **Date** class with



yesterday(self) → -= 1

tomorrow(self) → += 1

addNDays(self, N) → += N

subNDays(self, N) → -= N

isBefore(self, d2) → <

isAfter(self, d2) → >

diff(self, d2) → -

dow(self) →

methods

operators!

Prof. Benjamin !

no computer required…

What's the **diff**?

```
In : today = Date(11,13,2018)
In : wd = Date(11,12,2013)
In : today.diff(wd)
Out: 1827
```

method

```
In : today - wd
Out: 1827
```

operator

```
In : wd - today
Out: -1827
```

operator

```
In : eraday = Date(1,1,1)
In : today.diff(eraday)
Out: 737010
```

method

```
In : today - eraday
Out: 737010
```

operator

This gives me pause

Where's the dow?

The dow looks down to me!

```
In : sm1 = Date(10,28,1929)
In : sm2 = Date(10,19,1987)

In : sm1.dow()
Out: 'Monday'
```

uses a *named* object...

```
In : sm2.dow()
Out: 'Monday'
```

uses a *named* object...

```
In : Date(1,1,1).dow()
Out: 'Monday'
```

unnamed!

```
In : Date(1,1,2100).dow()
Out: 'Friday'
```

unnamed!

```
In : Date(10,10,2010).dow()
Out: 'Sunday'
```

popular!

The Date class

```
class Date:
    """ a blueprint (class) for objects
        that represent calendar days
    """
```

This is the start of a new type called Date
It begins with the keyword **class**

This is the **constructor** for Date objects
As is typical, it assigns input data to the data members.

```
    def __init__( self, mo, dy, yr ):
        """ the Date constructor """
        self.month = mo
        self.day = dy
        self.year = yr
```

These are data members – they are the information inside every Date object.

Date

This is a **class**. It is a user-defined datatype that you'll finish building in Lab 10 next week…

```
In : d = Date(11,12,2013)
In : d.isLeapYear()
False
```

Constructor!

add a print statement to see that this is OUR OWN code…

d contains data members named **day**, **month**, and **year**

```
>>> d
11/12/2013
```

The repr!

the **repr**esentation of an object of type Date

```
>>> d.isLeapYear()
False
```

The **isLeapYear** method returns **True** or **False**.
How does it know ***what year to check***?

The Date class

```
class Date:
    """ a blueprint (class) for objects
        that represent calendar days
    """
    def __init__( self, mo, dy, yr ):
        """ the Date constructor """
        self.month = mo
        self.day = dy
        self.year = yr

    def __repr__( self ):
        """ used for printing Dates """
        s = "{:02d}/{:02d}/{:04d}".format(self.month, self.day, self.year)
        return s
```

Why is everything
so far away?!

This is the **repr** for Date objects
It tells Python how to print these objects.

Why **`self`** instead of **`d`** ?

self is the variable calling a method

```
>>> d = Date(11,12,2013)
>>> print d
11/12/2013

>>> d.isLeapYear()
False
```

```
>>> nd = Date(1,1,2020)
>>> print nd
01/01/2020
>>> nd.isLeapYear()
True
```

These methods need access to the object that calls them: it's **self**

Problems with ==

```
>>> wd = Date(11,12,2013)
>>> wd
11/12/2013
```

this constructs a different Date

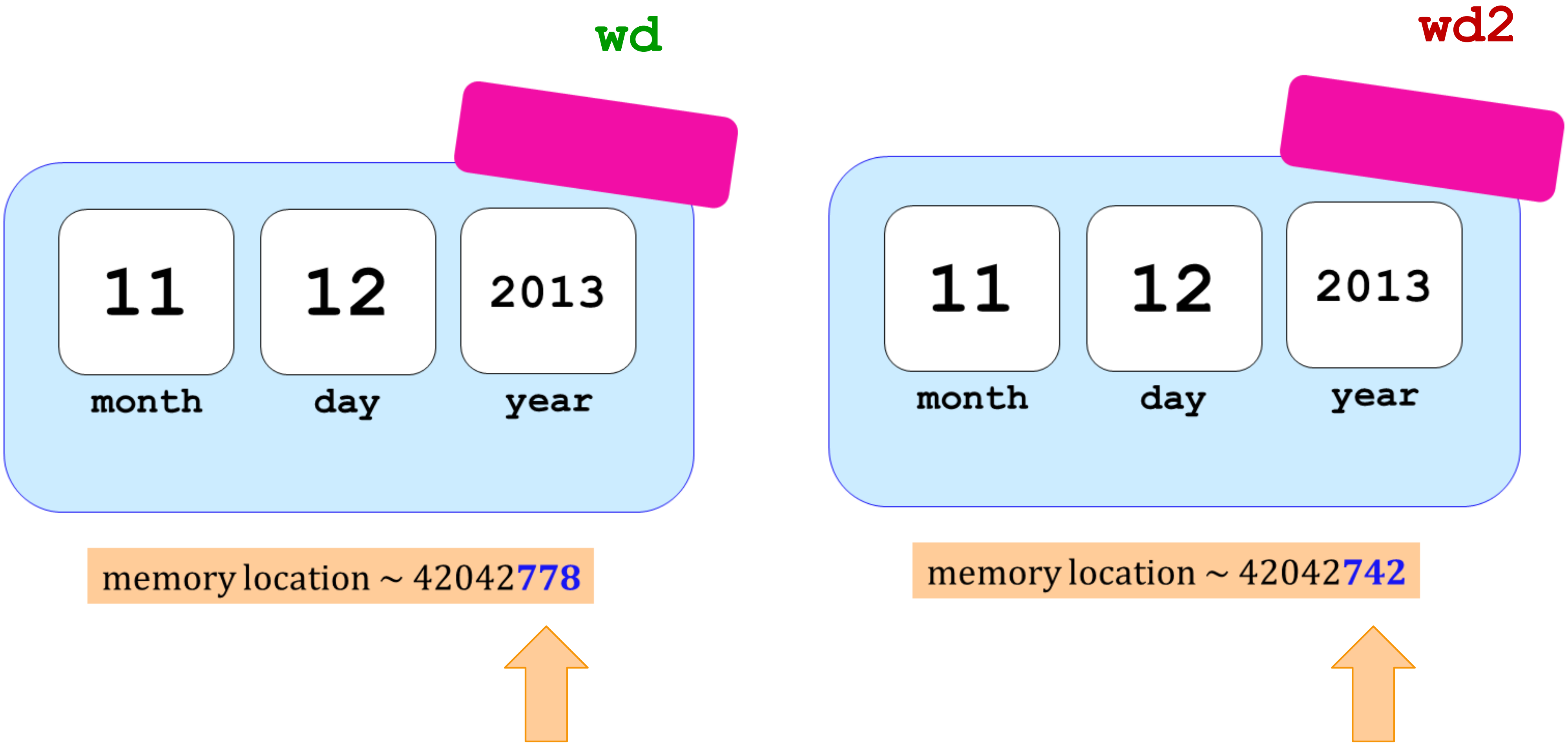
```
>>> wd2 = Date(11,12,2013)
>>> wd2
11/12/2013
```

```
>>> wd == wd2
False
```

How can this be False ?

Python objects are handled by reference...
== compares references!

Two Date objects:



memory location ~ 42042778

memory location ~ 42042742

== compares **memory locations**, not contents

originals underneath…

equals

```
class Date:

    def __init__( self, mo, dy, yr ):
    def __repr__(self):
    def isLeapYear(s

    def equals(
        """ returns
            represent
            False oth
        """
        if self.year =
           self.month
           self.day ==
               return
        else:
               return
```

Let's write *our own* equality-tester

To use this, write `wd.equals(wd2)`

which goes where?

equals

```
class Date:

    def __init__( self, mo, dy, yr ):
    def __repr__(self):
    def isLeapYear(self):

    def equals(self, d2):
        """ returns True if they
            represent the same date;
            False otherwise
        """
        if self.year == d2.year and \
           self.month == d2.month and \
           self.day == d2.day:
                return True
        else:
                return False
```

To use this, write `wd.equals(wd2)`

which goes where?

Solution: **equals**

```
>>> wd = Date(11,12,2013)
>>> wd
11/12/2013
```

```
>>> wd2 = Date(11,12,2013)
>>> wd2
11/12/2013
```

this constructs a different Date object, but with the same mo/dy/yr

```
>>> wd.equals(wd2)
True
```

.equals compares mo/dy/yr – because *we asked it to!*

But *who* is this convenient for?!

__eq__

```
class Date:

    def __init__( self, mo, dy, yr ):
    def __repr__(self):
    def isLeapYear(self):

    def __eq__(self, d2):
        """ returns True if they
            represent the same date;
            False otherwise
        """
        if self.year == d2.year and \
           self.month == d2.month and \
           self.day == d2.day:
                return True
        else:
                return False
```

L==k! This is T== C==L!

redefined for *our convenience*!

To use this, write `d == d2`

DIY operators …

__eq__(self, other) defines the equality operator, ==
__ne__(self, other) defines the inequality operator, !=
__lt__(self, other) defines the less-than operator, <
__gt__(self, other) defines the greater-than operator, >
__le__(self, other) defines the less-or-equal-to operator, <=
__ge__(self, other) defines the gr.-or-equal-to operator, >=

__add__(self, other) defines the addition operator, +
__sub__(self, other) defines the subtraction operator, -

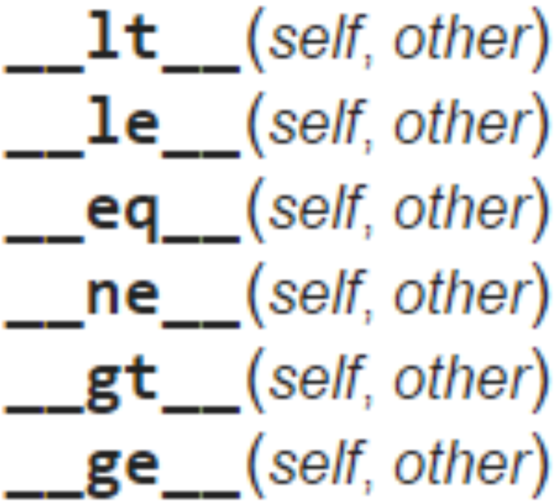
… and many more! Use dir('')

there are two under-scores on each side here

I should **underscore** this unusual syntax!

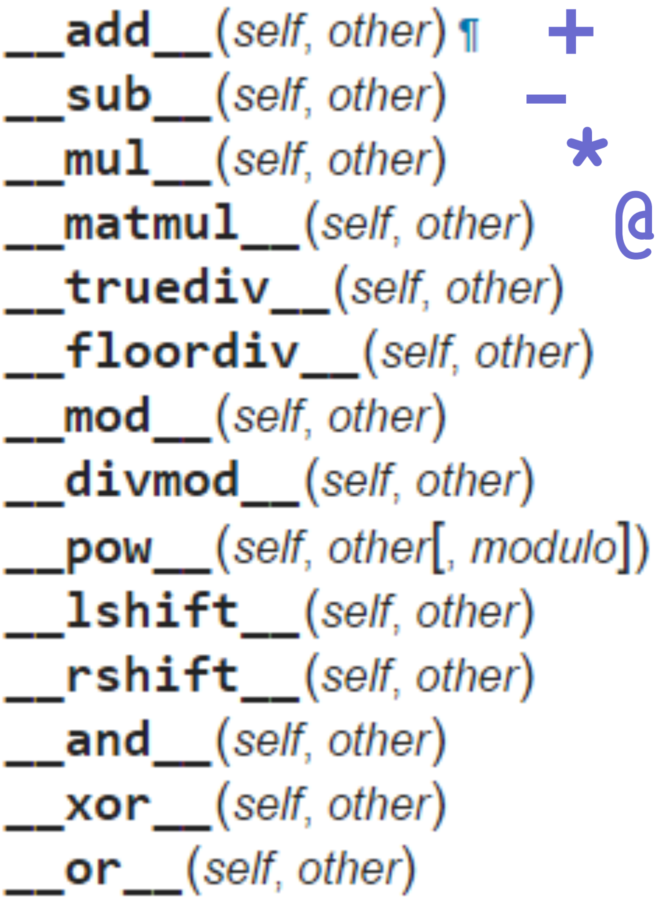
More operators!

Booleans

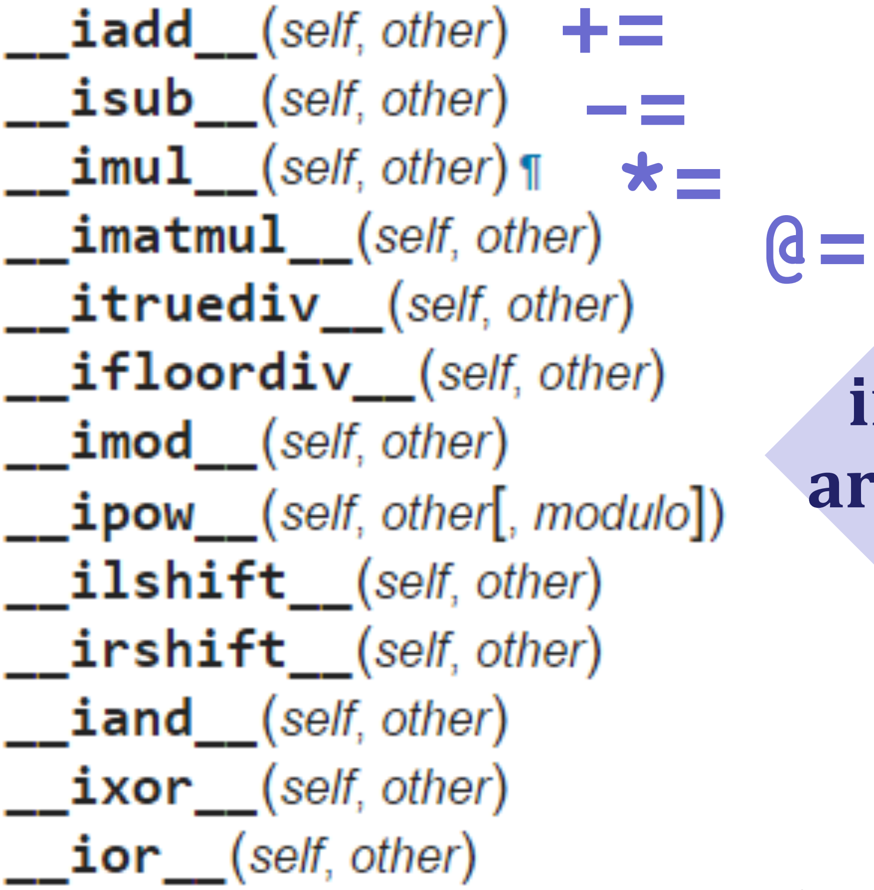


```
__lt__(self, other)
__le__(self, other)
__eq__(self, other)
__ne__(self, other)
__gt__(self, other)
__ge__(self, other)
```

arithmetic



```
__add__(self, other) ¶      +
__sub__(self, other)       -
__mul__(self, other)        *
__matmul__(self, other)      @
__truediv__(self, other)
__floordiv__(self, other)
__mod__(self, other)
__divmod__(self, other)
__pow__(self, other[, modulo])
__lshift__(self, other)
__rshift__(self, other)
__and__(self, other)
__xor__(self, other)
__or__(self, other)
```



```
__iadd__(self, other)        +=
__isub__(self, other)         -=
__imul__(self, other) ¶        *=
__imatmul__(self, other)          @=
__itruediv__(self, other)
__ifloordiv__(self, other)
__imod__(self, other)
__ipow__(self, other[, modulo])
__ilshift__(self, other)
__irshift__(self, other)
__iand__(self, other)
__ixor__(self, other)
__ior__(self, other)
```

in-place arithmetic

https://docs.python.org/3/reference/datamodel.html#special-method-names

hw10pr1

Add these to your **Date** class!

`yesterday(self)`	`-= 1`
`tomorrow(self)`	`+= 1`
`addNDays(self, N)`	`+= N`
`subNDays(self, N)`	`-= N`
`isBefore(self, d2)`	`<`
`isAfter(self, d2)`	`>`
`diff(self, d2)`	`-`
`dow(self)`	
	`abs?`

Prof. Benjamin !

no computer required…

and use your **Date** class to analyze our calendar a bit…

isBefore

```python
class Date:

    def isBefore(self, d2):
        """ True if self is before d2, else False """
        if self.year  < d2.year:
            return True
        elif self.month < d2.month:
            return True
        elif self.day   < d2.day:
            return True
        else: return False
```

```python
Date(12,31,1999).isBefore(Date(11,13,2018))
```

```python
Date(11,13,2018).isBefore(Date(12,31,1999))
```

Why doesn't this function work correctly?!

isBefore

```
class Date:

    def isBefore(self, d2):
        """ True if self is before d2, else False """
        if self.year  < d2.year:
            return True

        elif self.month < d2.month and self.year == d2.year :
            return True

        elif self.day < d2.day and self.year == d2.year \
                                 and self.month == d2.month :

            return True
        else:
            return False
```

I <3 Elf! But what about Elif?

__lt__ <

I want *LESS* !

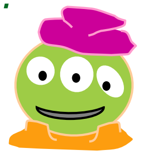
```
class Date:

    def __lt__(self, d2):
        """ if self is before d2, this should
            return True; else False """

        if self.isBefore(d2) == True:
            return True
        else:
            return False
```

__lt__

<

```
class Date:

    def __lt__(self, d2):
        """ this is less than most code! """
        return self.isBefore(d2)
```

__lt__ <



```
class Date:

    def __lt__(self, d2):
        """ this is less than most code! """
        return self.isBefore(d2)
```

__gt__ >

```
    def __gt__(self, d2):
        """ this is less than most code! """
        return _____.isBefore(_____)
```

The 2 most essential *methods*

```
>>>  wd = Date(11,12,2013)
>>> print wd
11/12/2013
```

construct with the **CONSTRUCTOR** ...

print uses **__repr__**

```
>>> wd.tomorrow()
>>> print wd
11/13/2013
```

the **tomorrow** method returns nothing at all. Is it doing anything?

```
d += 1
```

wd has changed!

```
>>> wd.yesterday()
>>> print wd
11/12/2013
```

yesterday is pretty much just like **tomorrow** (is this a good thing!?)

```
d -= 1
```

Some methods return a value; others change the object that call it!

```
class Date:

    def tomorrow(self):
        """ moves the self date ahead 1 day """

        DIM = [0,31,28,31,30,31,30,31,31,30,31,30,31]

        self.day += 1

        if

            self.day
            self.month
            if
```

Don't hand this in... *Use for hw10pr1 next week!*

first, add 1 to **`self.day`**

DIM looks pretty bright to me!

test if we have gone "out of bounds!"

then, adjust the month and year, but only as needed

Use another if!

Don't return anything. This **CHANGES** the date object that calls it.

Extra How could we make this work for leap years, too?

```
class Date:

  def tomorrow(self):
    """ moves the self date ahead 1 day """
```

better as a *variable*!

```
    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]

    self.day += 1     # add 1 to the day!

    if self.day > DIM[self.month]:   # check day
        self.month += 1
        self.day = 1

        if self.month > 12:              # check month
            self.year += 1
            self.month = 1
```

```
class Date:

  def tomorrow(self):
    """ moves the self date ahead 1 day """

    if self.isLeapYear() == True:   fdays = 29
    else: fdays = 28

    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]

    self.day += 1     # add 1 to the day!

    if self.day > DIM[self.month]:   # check day
        self.month += 1
        self.day = 1

        if self.month > 12:          # check month
            self.year += 1
            self.month = 1
```

```
class Date:

  def tomorrow(self):
    """ moves the self date ahead 1 day """

    fdays = 28 + self.isLeapYear()     # What ?!

    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]

    self.day += 1       # add 1 to the day!

    if self.day > DIM[self.month]:    # check day
        self.month += 1
        self.day = 1

        if self.month > 12:           # check month
            self.year += 1
            self.month = 1
```

the "Luke trick"!

Don't hand this in... *Use for hw10pr1 this week!*

```
class Date:

  def yesterday(self):
    """ moves the self date backwards 1 day """

    fdays = 28 + self.isLeapYear()    # Yay!

    DIM = [0,31,fdays,31,30,31,30,31,31,30,31,30,31]
   self.day -= 1     # sub 1 from the day!

       if self.day < 1:   # check day
           self.month -= 1
           self.day = DIM[self.month-1]

           if self.month > 12:          #
   check month
               self.year += 1
               self.month = 1
```

For lab: how will "wrap-around" work in this case? *What cases do we need to worry about?!*