A whole new **class** of programming

ONTARIO MILLS
For your convenience, please enter through the mall entrance.
WOLFGANG PUCK CAFE
Thank you.

Whose convenience?

Read 6.1-6.3

CS 5 overview

✓ CS's building blocks: functions and composition

✓ behind CS's curtain: *circuits, assembly, loops*

NOW! *Designing Data!* The Date class

Coming soon…

CS: theory + practice

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.

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 its methods, objects refer to themselves as **`self`**

(4) Python signals special methods with double 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!*

The **`Date`** class

```
class Date:
    """A user-defined data structure that
       stores and manipulates dates."""
```

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

This is the <u>**constructor**</u> for Date objects
As is typical, it assigns its arguments to the data members.

```
    def __init__(self, month, day, yr):
        """Construct a Date with
           the given month, day, and year."""
        self.month = month
        self.day = day
        self.year = yr
```

Names don't *have* to match!

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

Date

This is a class. It is a user-defined data type that you'll build in Lab 10 this week...

```
>>> d = Date(4, 6, 2021)
```

Constructor!

Add a print statement to see that this is OUR OWN code...

```
>>> d.month
4
>>> d.day
4
```

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

```
>>> d
04/06/2021
```

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 user-defined data structure that
       stores and manipulates dates."""

    def __init__(self, month, day, yr):
        """Construct a Date with
           the given month, day, and year."""
        self.month = month
        self.day = day
        self.year = yr

    def __repr__(self):
        """Display a date in a nice format."""
        s = f"{self.month:02d}/{self.day:02d}/{self.year:04d}"
        return s
```

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

Why is everyone so far away?!

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

self

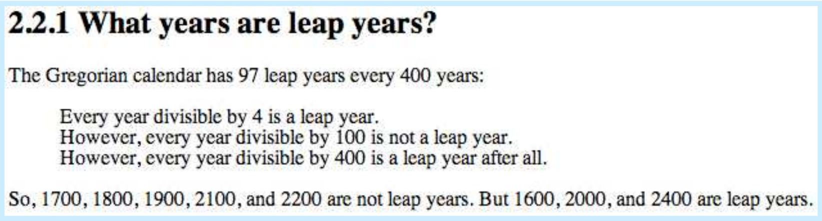
is the variable calling a method

```
>>> d = Date(4, 6, 2021)
>>> print(d)
04/06/2021

>>> d.isLeapYear()
False

>>> ny = Date(1, 1, 2024)
>>> print(ny)
01/01/2024
>>> ny.isLeapYear()
True
```

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



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, month, day, yr): # (constructor)
    def __repr__(self): # (for printing)

    def isLeapYear(self):
        """Here it is in all its glory"""
        if self.year % 400 == 0: return True
        elif self.year % 100 == 0: return False
        elif self.year % 4 == 0: return True
        else: return False
```

== vs. equals

UFO license
Area 51, CA

What **id** is on your Date?

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

This constructs a different Date

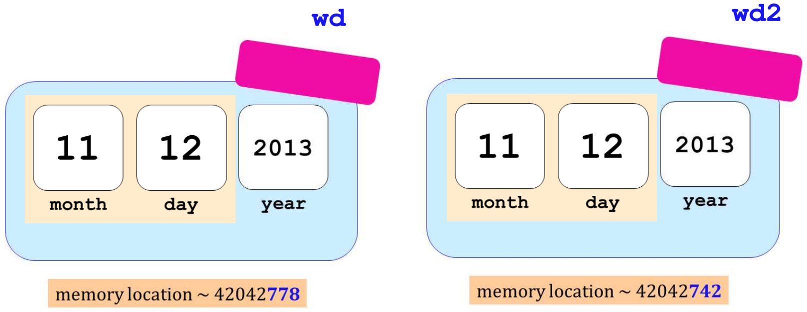
```
>>> wd2 = Date(11, 12, 2013)
>>> wd2
11/12/2013

>>> wd == wd2
False
```

How can this be False ?

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

Two Date objects:



wd

wd2

11 month | 12 day | 2013 year

11 month | 12 day | 2013 year

memory location ~ 42042778

memory location ~ 42042742

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

originals underneath...

== vs. equals

UFO license
Area 51, CA

What date is on your **id**?
What **id** is on your Date?

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

This constructs a different Date

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

>>> d.equals(d2)
True
```

Python objects are handled by reference...
.equals compares contents

equals

```
class Date:

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

    def __eq__(self, d2):
        """Returns True if self and d2
           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

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)
```

in-place arithmetic

```
__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)
```

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

__lt__ <

LESS !

```
class Date:

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

```
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 [                    ]
```

Don't hand this in... *Use for hw11pr1 this 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, only if needed

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"""
      if self.isLeapYear(): 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
```

Name(s) ____________________ *Quiz*

```
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
```

Challenge #1 If

d prints as 4/1/2021
d2 prints as 4/6/2021

What does **d.isBefore(d2)** return?
Which of the *4* return statements is used?
Is this the *correct* value?

Challenge #2

Find *different* dates, d and d2, for which **d.isBefore(d2)** returns an INCORRECT value...

This Date is Late!

Extra! Above, show how to *fix* the **isBefore** method ...