

OBJECT ORIENTED PROGRAMMING

(download slides and .py files to follow along!)

6.0001 LECTURE 7

ANA BELL

OBJECTS

- Python supports many different kinds of data

```
1234          3.14159      "Hello"      [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- Each is an **object**, and every object has:
 - An internal **data representation** (primitive or composite)
 - A set of procedures for **interaction** with the object
- An object is an **instance** of a **type**
 - `1234` is an instance of an `int`
 - `"hello"` is an instance of a string

OBJECT ORIENTED PROGRAMMING (OOP)

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- Can **create new objects** of some type
- Can **manipulate objects**
- Can **destroy objects**
 - Explicitly using `del` or just “forget” about them
 - Python system will reclaim destroyed or inaccessible objects – called “garbage collection”

WHAT ARE OBJECTS?

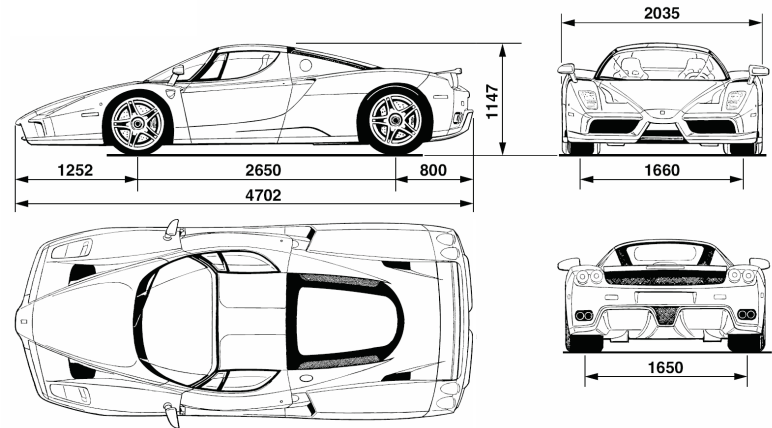
- Objects are **a data abstraction** that captures...

(1) An **internal representation**

- Through data attributes

(2) An **interface** for interacting with object

- Through methods (aka procedures/functions)
- Defines behaviors but hides implementation



EXAMPLE:

[1,2,3,4] has type list

- How are lists **represented internally**? linked list of cells



*follow pointer to
the next index*

- How to **manipulate** lists?
 - `L[i]`, `L[i:j]`, `+`
 - `len()`, `min()`, `max()`, `del(L[i])`
 - `L.append()`, `L.extend()`, `L.count()`, `L.index()`,
`L.insert()`, `L.pop()`, `L.remove()`, `L.reverse()`, `L.sort()`
- Internal representation should be private
- Correct behavior may be compromised if you manipulate internal representation directly

ADVANTAGES OF OOP

- **Bundle data into packages** together with procedures that work on them through well-defined interfaces
- **Divide-and-conquer** development
 - Implement and test behavior of each class separately
 - Increased modularity reduces complexity
- **Classes** make it easy to **reuse** code
 - Many Python modules define new classes
 - Each class has a separate environment (no collision on function names)
 - Inheritance allows subclasses to redefine or extend a selected subset of a superclass' behavior

CREATING AND USING YOUR OWN TYPES WITH CLASSES

- Make a distinction between **creating a class** and **using an instance** of the class
- **Creating** the class involves
 - Defining the class name
 - Defining class attributes
 - *for example, someone wrote code to implement a list class*
- **Using** the class involves
 - Creating new **instances** of the class
 - Doing operations on the instances
 - *for example, $L = [1, 2]$ and $len(L)$*

DEFINE YOUR OWN TYPES

- Use the `class` keyword to define a new type

class definition
`class` *name/type* `Coordinate` (*class parent* `object`) :

`#define attributes here`

- Similar to `def`, indent code to indicate which statements are part of the **class definition**
- The word `object` means that `Coordinate` is a Python object and **inherits** all its attributes (inheritance next lecture)
 - `Coordinate` is a subclass of `object`
 - `object` is a superclass of `Coordinate`

WHAT ARE ATTRIBUTES?

- Data and procedures that “**belong**” to the class
- **Data attributes**
 - Think of data as other objects that make up the class
 - *for example, a coordinate is made up of two numbers*
- **Methods** (procedural attributes)
 - Think of methods as functions that only work with this class
 - How to interact with the object
 - *for example you can define a distance between two coordinate objects but there is no meaning to a distance between two list objects*

DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

- First have to define **how to create an instance** of class
- Use a **special method called `__init__`** to initialize some data attributes or perform initialization operations

```
class Coordinate(object):
```

```
def __init__(self, x, y):
```

```
    self.x = x
```

```
    self.y = y
```

special method to
create an instance
— is double
underscore

two data attributes
for every
Coordinate object

what data initializes a
Coordinate object

parameter to
refer to an
instance of the
class

ACTUALLY CREATING AN INSTANCE OF A CLASS

```
c = Coordinate(3,4)
origin = Coordinate(0,0)
print(c.x)
print(origin.x)
```

use the dot to
access an attribute
of instance `c`

create a new object
of type
`Coordinate` and
pass in 3 and 4 to
the `__init__`

- Data attributes of an instance are called **instance variables**
- Don't provide argument for `self`, Python does this automatically

WHAT IS A METHOD?

- Procedural attribute, like a **function that works only with this class**
- Python always passes the object as the first argument
 - Convention is to use **self** as the name of the first argument of all methods
- The “.” **operator** is used to access any attribute
 - A data attribute of an object
 - A method of an object

DEFINE A METHOD FOR THE `Coordinate` CLASS

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def distance(self, other):  
        x_diff_sq = (self.x - other.x) ** 2  
        y_diff_sq = (self.y - other.y) ** 2  
        return (x_diff_sq + y_diff_sq) ** 0.5
```

use it to refer to any instance

another parameter to method

dot notation to access data

- Other than `self` and dot notation, methods behave just like functions (take params, do operations, return)

HOW TO USE A METHOD

```
def distance(self, other):  
    # code from prev slide here
```

method def

Using the class:

■ Conventional way

```
c = Coordinate(3,4)  
zero = Coordinate(0,0)  
print(c.distance(zero))
```

*object to call
method on*

*name of
method*

*parameters not
including self
(self is
implied to be c)*

■ Equivalent to

```
c = Coordinate(3,4)  
zero = Coordinate(0,0)  
print(Coordinate.distance(c, zero))
```

*name of
class*

*name of
method*

*parameters, including an
object to call the method
on, representing self*

EXAMPLE: FRACTIONS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
 - Numerator
 - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
 - Add, subtract
 - Invert the fraction
- Let's write it together!

PRINT REPRESENTATION OF AN OBJECT

```
>>> c = Coordinate(3,4)
>>> print(c)
<__main__.Coordinate object at 0x7fa918510488>
```

- **Uninformative** print representation by default
- Define a **`__str__` method** for a class
- Python calls the `__str__` method when used with `print` on your class object
- You choose what it does! Say that when we print a `Coordinate` object, want to show

```
>>> print(c)
<3,4>
```

DEFINING YOUR OWN PRINT METHOD

```
class Coordinate(object):  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
    def distance(self, other):  
        x_diff_sq = (self.x-other.x)**2  
        y_diff_sq = (self.y-other.y)**2  
        return (x_diff_sq + y_diff_sq)**0.5  
    def __str__(self):  
        return "<" + str(self.x) + ", " + str(self.y) + ">"
```

name of
special
method

must return
a string

WRAPPING YOUR HEAD AROUND TYPES AND CLASSES

- Can ask for the type of an object instance

```
>>> c = Coordinate(3,4)
```

```
>>> print(c)
```

```
<3,4>
```

```
>>> print(type(c))
```

```
<class __main__.Coordinate>
```

return of the `__str__` method
the type of object `c` is a class `Coordinate`

- This makes sense since

```
>>> print(Coordinate)
```

```
<class __main__.Coordinate>
```

```
>>> print(type(Coordinate))
```

```
<type 'type'>
```

a `Coordinate` is a class
a `Coordinate` class is a type of object

- Use `isinstance()` to check if an object is a `Coordinate`

```
>>> print(isinstance(c, Coordinate))
```

```
True
```

SPECIAL OPERATORS

- `+`, `-`, `==`, `<`, `>`, `len()`, `print`, and many others

<https://docs.python.org/3/reference/datamodel.html#basic-customization>

- Like `print`, can override these to work with your class
- Define them with double underscores before/after

<code>__add__(self, other)</code>	→	<code>self + other</code>
<code>__sub__(self, other)</code>	→	<code>self - other</code>
<code>__eq__(self, other)</code>	→	<code>self == other</code>
<code>__lt__(self, other)</code>	→	<code>self < other</code>
<code>__len__(self)</code>	→	<code>len(self)</code>
<code>__str__(self)</code>	→	<code>print self</code>

... and others

EXAMPLE: FRACTIONS

- Create a **new type** to represent a number as a fraction
- **Internal representation** is two integers
 - Numerator
 - Denominator
- **Interface** a.k.a. **methods** a.k.a **how to interact** with `Fraction` objects
 - Add, sub, mult, div to work with `+`, `-`, `*`, `/`
 - Print representation, convert to a float
 - Invert the fraction
- Let's write it together!

THE POWER OF OOP

- **Bundle together objects** that share
 - Common attributes and
 - Procedures that operate on those attributes
- Use **abstraction** to make a distinction between how to implement an object vs how to use the object
- Build **layers** of object abstractions that inherit behaviors from other classes of objects
- Create our **own classes of objects** on top of Python's basic classes

5 Min Break, then Quiz Time!

- Sit at a seat, **not on the floor**
- No aids allowed, **only MITx and your IDE**
- If you finish early, **stay in your seat** (no phones, external websites, etc)
- **Checkout password given in the last 2 mins of exam**

Exam link:

bit.ly/60001-mq2-f19