

# OBJECT ORIENTED PROGRAMMING

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

---

6.0001 LECTURE 8

# 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:
  - a **type**
  - 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 objects
  - doing operations on the instances
  - *for example,  $L = [1, 2]$  and  $\text{len}(L)$*

# DEFINE YOUR OWN TYPES

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

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

`#define attributes here`

such like.  
def function  
-----  
⇒ def name( )  
Code.

- 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 object
- use a **special method called `__init__`** to initialize some data attributes

```
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)
```

The diagram illustrates the creation and use of two objects of the `Coordinate` class. A green circle highlights the variable `c`, and a blue triangle highlights the variable `origin`. Red boxes highlight the object creation `Coordinate(3,4)` and `Coordinate(0,0)`, and the attribute access `c.x` and `origin.x`. Arrows indicate that `c` points to the first object and `origin` points to the second object.

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 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

# 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 &lt; 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, subtract
  - print representation, convert to a float
  - invert the fraction
- the code for this is in the handout, check it out!

# 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

MIT OpenCourseWare

<https://ocw.mit.edu>

6.0001 Introduction to Computer Science and Programming in Python

Fall 2016

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.