

STAT606

Computing for Data Science and Statistics

Lecture 7: Classes, Operators and Inheritance

Classes: programmer-defined types

Sometimes we use a collection of variables to represent a specific object

Example: we used a tuple of tuples to represent a matrix

Example: representing state of a board game

List of players, piece positions, etc.

Example: representing a statistical model

Want to support methods for estimation, data generation, etc.

Important point: these data structures quickly become very complicated, and we want a way to encapsulate them. This is a core motivation (but hardly the only one) for **object-oriented programming**.

Classes encapsulate data types

Example: I want to represent a point in 2-dimensional space \mathbb{R}^2

Option 1: just represent a point by a 2-tuple

Classes encapsulate data types

Example: I want to represent a point in 2-dimensional space \mathbb{R}^2

Option 1: just represent a point by a 2-tuple

Option 2: make a point **class**, so that we have a whole new data type
Additional good reasons for this will become apparent shortly!

Classes encapsulate data types

Example: I want to represent a point in 2-dimensional space \mathbb{R}^2

Option 1: just represent a point by a 2-tuple

Option 2: make a point **class**, so that we have a whole new data type
Additional good reasons for this will become apparent shortly!

```
1 class Point:
2     '''Represents a 2-d point.'''
```

Class header declares a new class, called `Point`.

```
1 print(Point)
<class '__main__.Point'>
```

Docstring provides explanation of what the class represents, and a bit about what it does. This is an ideal place to document your class.

Classes encapsulate data types

Note: By convention, class names are written in **CamelCase**.

Example: I want to represent a point in 2-dimensional space \mathbb{R}^2

Option 1: just represent a point by a 2-tuple

Option 2: make a point **class**, so that we have a whole new data type
Additional good reasons for this will become apparent shortly!

```
1 class Point:
2     '''Represents a 2-d point.'''
```

```
1 print(Point)
```

Class definition creates a **class object**, Point.

```
<class '__main__.Point'>
```

Creating an object: Instantiation

```
class Point:  
    '''Represents a 2-d point.'''
```

```
4 p = Point()  
5 p
```

```
<__main__.Point at 0x10669b940>
```

This defines a class `Point`, and from here on we can create new variables of type `Point`.

Creating an object: Instantiation

```
1 class Point:
2     '''Represents a 2-d point.'''
3
4 p = Point()
5 p
```

Creating a new object is called **instantiation**. Here we are creating an **instance** `p` of the class `Point`.

```
<__main__.Point at 0x10669b940>
```

Indeed, `p` is of type `Point`.

Note: An **instance** is an individual object from a given class. In general, the terms **object** and **instance** are interchangeable: an object is an instantiation of a class.

Assigning Attributes

```
1 p = Point()  
2 p.x = 3.0  
3 p.y = 4.0  
4 (p.x, p.y)
```

```
(3.0, 4.0)
```

This dot notation should look familiar. Here, we are assigning values to **attributes** `x` and `y` of the object `p`. This both creates the attributes, and assigns their values.

Once the attributes are created, we can access them, again with dot notation.

Assigning Attributes

This dot notation should look familiar. Here, we are assigning values to **attributes** `x` and `y` of the object `p`. This both creates the attributes, and assigns their values.

```
1 p = Point()  
2 p.x = 3.0  
3 p.y = 4.0  
4 (p.x, p.y)
```

Once the attributes are created, we can access them, again with dot notation.

```
(3.0, 4.0)
```

```
1 p.goat
```

Attempting to access an attribute that an object doesn't have is an error.

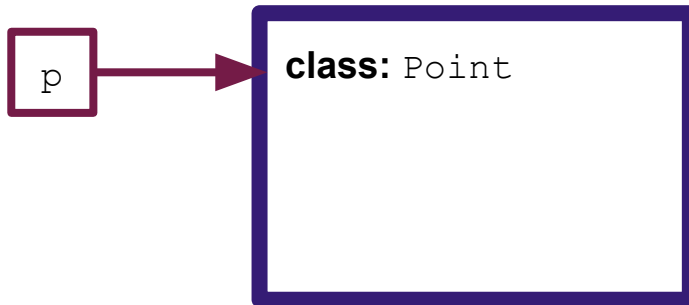
```
-----  
AttributeError                                Traceback (most recent call last)  
<ipython-input-5-f74ee22f01ba> in <module>()  
----> 1 p.goat
```

```
AttributeError: 'Point' object has no attribute 'goat'
```

Thinking about Attributes: Object Diagrams

```
1 class Point:  
2     '''Represents a 2-d point.'''  
3  
4 p = Point()  
5 p.x = 3.0  
6 p.y = 4.0
```

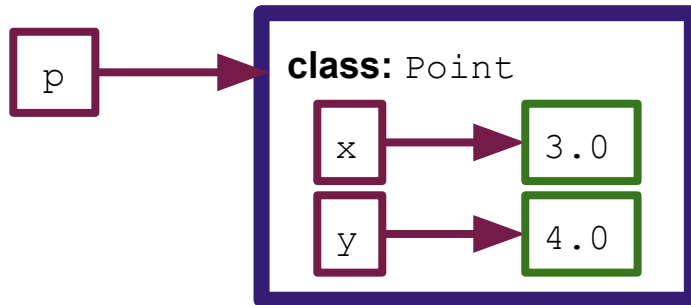
At this point, `p` is just an object with no attributes.



Thinking about Attributes: Object Diagrams

```
1 class Point:
2     '''Represents a 2-d point.'''
3
4 p = Point()
5 p.x = 3.0
6 p.y = 4.0
```

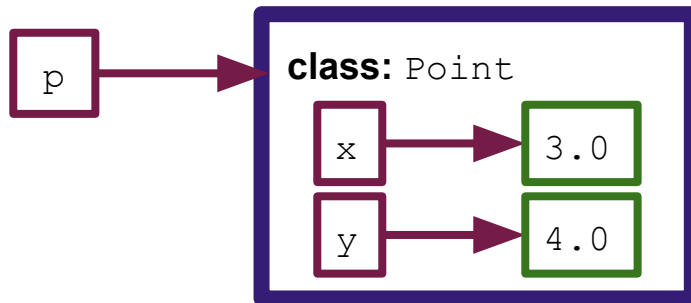
After these two lines, `p` has attributes `x` and `y`.



Thinking about Attributes: Object Diagrams

```
1 class Point:
2     '''Represents a 2-d point.'''
3
4 p = Point()
5 p.x = 3.0
6 p.y = 4.0
```

After these two lines, `p` has attributes `x` and `y`.

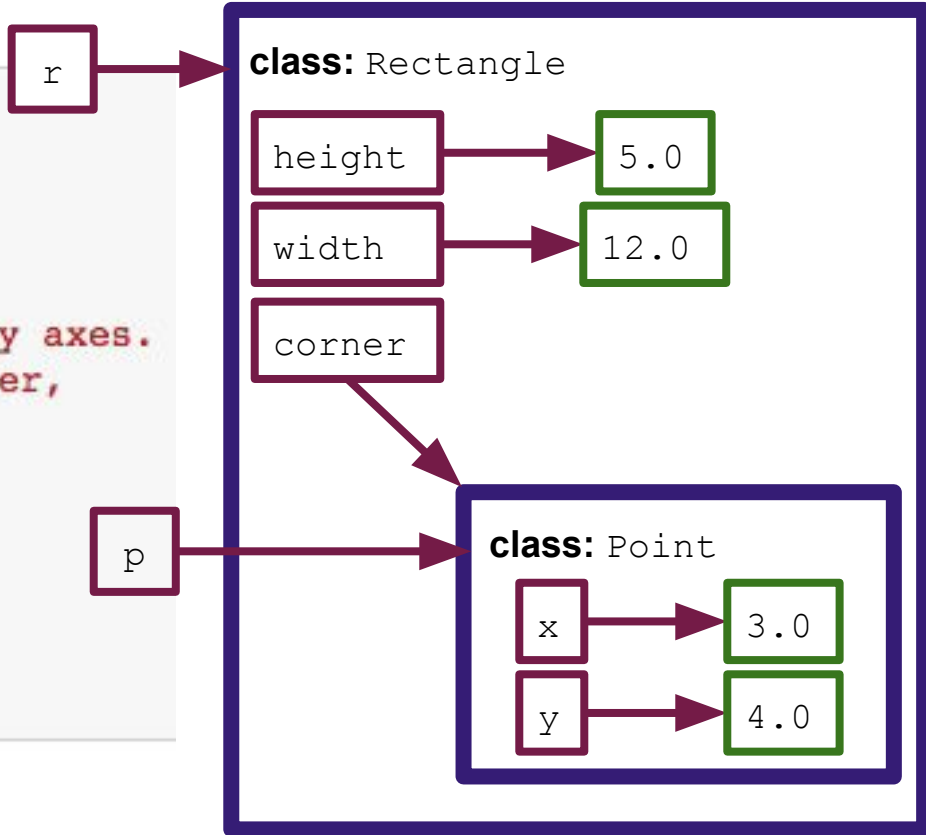


So dot notation `p.x`, essentially says, look inside the object `p` and find the attribute `x`.

Objects can have other objects as their attributes.
We often call the attribute object **embedded**.

Nesting Objects

```
1 class Point:
2     '''Represents a 2-d point.'''
3
4 class Rectangle:
5     '''Represents a rectangle whose
6     sides are parallel to the x and y axes.
7     Specified by its upper-left corner,
8     height, and width.'''
9
10 p = Point(); p.x = 3.0; p.y = 4.0
11 r = Rectangle()
12 r.corner = p
13 r.height = 5.0
14 r.width = 12.0
```



Nesting Objects

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 r1 = Rectangle()
3 r1.corner = p1
4 r1.height = 5.0
5 r1.width = 12.0
6
7 r2 = Rectangle()
8 r2.corner = Point()
9 r2.corner.x = 3.0
10 r2.corner.y = 4.0
11 r2.height = 5.0
12 r2.width = 12.0
```

Both of these blocks of code create equivalent `Rectangle` objects.

Note here that instead of creating a point and then embedding it, we embed a `Point` object and *then* populate its attributes.

Objects are mutable

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 r1 = Rectangle()
3 r1.corner = p1
4 r1.height = 5.0; r1.width = 12.0
5 r1.height = 2*r1.height
6
7 def shift_rectangle(rec, dx, dy):
8     rec.corner.x = rec.corner.x + dx
9     rec.corner.y = rec.corner.y + dy
10
11 shift_rectangle(r1, 2, 3)
12 (r1.corner.x, r1.corner.y)
```

If my `Rectangle` object were immutable, this line would be an error, because I'm making an assignment.

Since objects are mutable, I can change attributes of an object inside a function and those changes remain in the object in the `__main__` namespace.

(5.0, 7.0)

Returning Objects

```
1 def double_sides(r):
2     rdouble = Rectangle()
3     rdouble.corner = r.corner
4     rdouble.height = 2*r.height
5     rdouble.width = 2*r.width
6     return(rdouble)
7
8 p1 = Point(); p1.x = 3.0; p1.y = 4.0
9 r1 = Rectangle()
10 r1.corner = p1
11 r1.height = 5.0
12 r1.width = 12.0
13
14 r2 = double_sides(r1)
15 r2.height, r2.width
```

(10.0, 24.0)

Functions can return objects. Note that this function is implicitly assuming that `rdouble` has the attributes `corner`, `height` and `width`. We will see how to do this soon.

The function creates a *new* Rectangle and returns it. Note that it doesn't change the attributes of its argument.

Copying and Aliasing

Recall that aliasing is when two or more variables have the same referent
i.e., when two variables are identical

Aliasing can often cause unexpected problems

Solution: make **copy** of object; variables equivalent, but not identical

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 import copy
3 p2 = copy.copy(p1)
4 p1 is p2
```

False

The `copy` module provides functions for copying objects. `p2` is a copy of `p1`, so they should **not** be identical...

Copying and Aliasing

Recall that aliasing is when two or more variables have the same referent
i.e., when two variables are identical

Aliasing can often cause unexpected problems

Solution: make **copy** of object; variables equivalent, but not identical

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 import copy
3 p2 = copy.copy(p1)
4 p1 is p2
```

False

The `copy` module provides functions for copying objects. `p2` is a copy of `p1`, so they should **not** be identical...

```
1 p1 == p2
```

False

...but they **should** be equivalent.

Copying and Aliasing

Documentation for the `copy` module:
<https://docs.python.org/3/library/copy.html>

Recall that aliasing is when two or more variables have the same referent
i.e., when two variables are identical

Aliasing can often cause unexpected problems

Solution: make **copy** of object; variables equivalent, but not identical

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 import copy
3 p2 = copy.copy(p1)
4 p1 is p2
```

False

```
1 p1 == p2
```

False

The
cop
the

Hey, those were supposed to be equivalent! What's up with that? **Answer:** by default, for programmer-defined types, `==` and `is` are the same. It's up to you, the programmer, to tell Python how to tell if two objects are equivalent, by defining a method `object.__eq__`. We'll come back to this.

...b

Copying and Aliasing

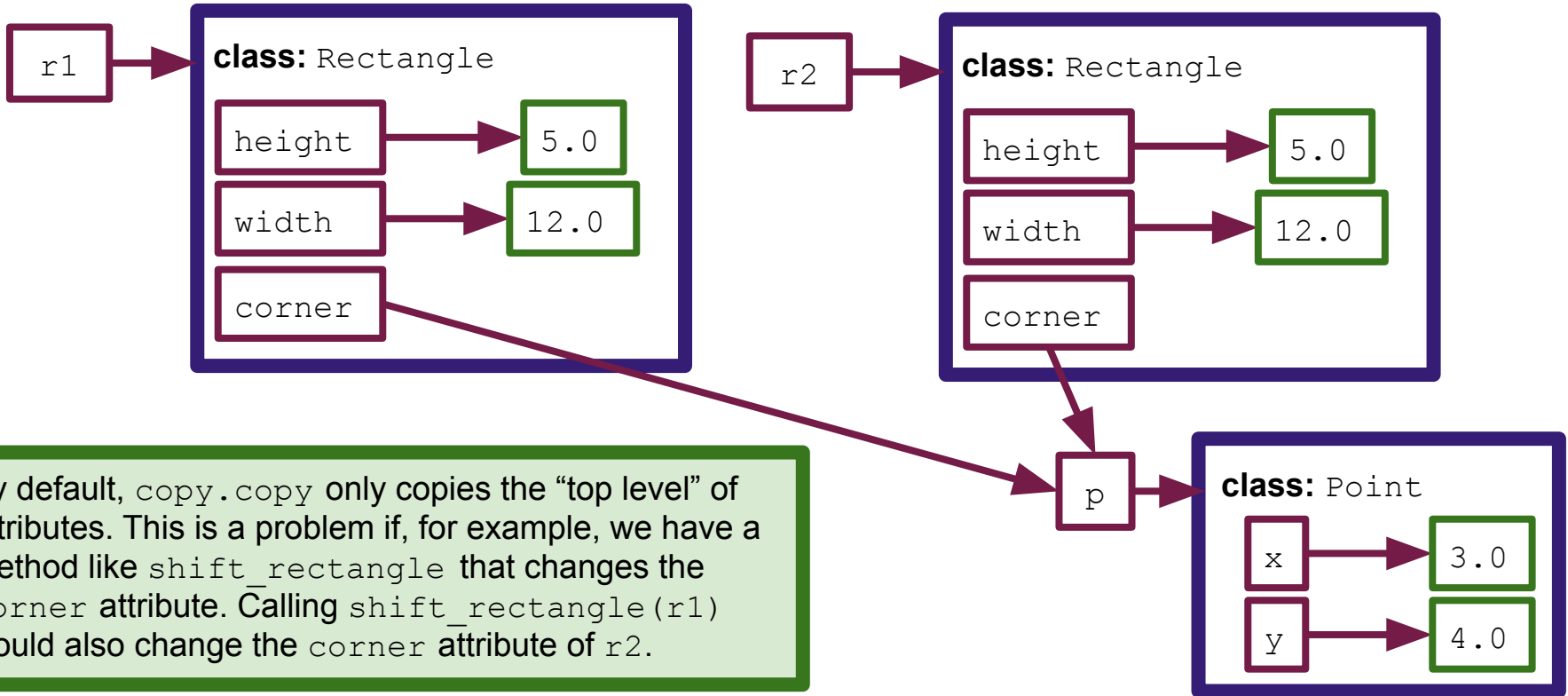
```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 r1 = Rectangle()
3 r1.corner = p1
4 r1.height = 5.0; r1.width = 12.0
5 r2 = copy.copy(r1)
6
7 r1.corner is r2.corner
```

True

Here we construct a Rectangle, and then copy it. Expected behavior is that mutable attributes should **not** be identical, and yet...

...evidently our copied objects still have attributes that are identical.

Copying and Aliasing



Copying and Aliasing

```
1 p1 = Point(); p1.x = 3.0; p1.y = 4.0
2 r1 = Rectangle()
3 r1.corner = p1
4 r1.height = 5.0; r1.width = 12.0
5 r2 = copy.deepcopy(r1)
6
7 r1.corner is r2.corner
```

False

`copy.deepcopy` is a recursive version of `copy.copy`. So it recursively makes copies of all attributes, and their attributes, and so on.

We often refer to `copy.copy` as a **shallow copy** in contrast to `copy.deepcopy`.

Now when we test for identity we get the expected behavior. Python has created a copy of `r1.corner`.

`copy.deepcopy` documentation explains how the copying operation is carried out:
<https://docs.python.org/3/library/copy.html#copy.deepcopy>

Pure functions vs modifiers

A **pure function** is a function that returns an object
...and **does not** modify any of its arguments

A **modifier** is a function that changes attributes of one or more of its arguments

```
1 def double_sides(r):
2     rdouble = Rectangle()
3     rdouble.corner = r.corner
4     rdouble.height = 2*r.height
5     rdouble.width = 2*r.width
6     return(rdouble)
7
8 def shift_rectangle(rec, dx, dy):
9     rec.corner.x = rec.corner.x + dx
10    rec.corner.y = rec.corner.y + dy
```

`double_sides` is a **pure function**. It creates a new object and returns it, without changing the attributes of its argument `r`.

`shift_rectangle` changes the attributes of its argument `rec`, so it is a **modifier**. We say that the function has **side effects**, in that it causes changes outside its scope.

[https://en.wikipedia.org/wiki/Side_effect_\(computer_science\)](https://en.wikipedia.org/wiki/Side_effect_(computer_science))

Pure functions vs modifiers

Why should one prefer one over the other?

Pure functions

Are often easier to debug and verify (i.e., check correctness)

https://en.wikipedia.org/wiki/Formal_verification

Common in **functional programming**

Modifiers

Often faster and more efficient

Common in **object-oriented programming**

Modifiers vs Methods

A modifier is a **function** that changes attributes of its arguments

A **method** is *like* a function, but it is provided by an object.

Define a class representing a 24-hour time.

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5     def print_time(self):
6         print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
7
8 t = Time()
9 t.hours=12; t.mins=34; t.secs=56
10 t.print_time()
```

Class supports a **method** called `print_time`, which prints a string representation of the time.

Every method must include `self` as its first argument. The idea is that the object is, in some sense, the object on which the method is being called.

12:34:56

Credit: Running example adapted from A. B. Downey, *Think Python*

More on Methods

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5     def print_time(self):
6         print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
7
8     def time_to_int(self):
9         return(self.secs + 60*self.mins + 3600*self.hours)
10
11 def int_to_time(seconds):
12     '''Convert a number of seconds to a Time object.'''
13     t = Time()
14     (minutes, t.secs) = divmod(seconds, 60)
15     (hrs, t.mins) = divmod(minutes, 60)
16     t.hours = hrs % 24 #military time!
17     return t
18
19 t = int_to_time(1337)
20 t.time_to_int()
```

`int_to_time` is a pure function that creates and returns a new `Time` object.

`Time.time_to_int` is a method, but it is still a pure function in that it has no side effects.

More on Modifiers

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5
6
7
8
9     def increment_pure(self, seconds):
10        '''Return new Time object representing this time
11        incremented by the given number of seconds.'''
12        t = Time()
13        t = int_to_time(self.time_to_int() + seconds)
14        return t
15
16    def increment_modifier(self, seconds):
17        '''Increment this time by the given
18        number of seconds.'''
19        (mins, self.secs) = divmod(self.secs+seconds, 60)
20        (hours, self.mins) = divmod(self.mins+mins, 60)
21        self.hours = (self.hours + hours)%24
22
23 t1 = int_to_time(1234)
24 t1.increment_modifier(1111)
25 t1.time_to_int()
```

I cropped out `time_to_int` and `print_time` for space.

Two different versions of the same operation. One is a pure function (pure method?), that does not change attributes of the caller. The second method is a modifier.

The modifier method does indeed change the attributes of the caller.

More on Modifiers

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4     def time_to_int(self):
5         return(self.secs + 60*self.mins + 3600*self.hours)
6     def print_time(self):
7         print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
8
9     def increment_pure(self, seconds):
10        '''Return new Time object representing this time
11        incremented by the given number of seconds.'''
12        t = Time()
13        t = int_to_time(self.time_to_int() + seconds)
14        return t
15
16 t1.increment_pure(100, 200)
```

Here's an error you may encounter.
How the heck did `increment_pure`
get 3 arguments?!

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-55-1d8fb5e5c628> in <module>()
     14         return t
     15
----> 16 t1.increment_pure(100, 200)
```

TypeError: increment_pure() takes 2 positional arguments but 3 were given

More on Modifiers

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4     def time_to_int(self):
5         return(self.secs + 60*self.mins + 3600*self.hours)
6     def print_time(self):
7         print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
8
9     def increment_pure(self, seconds):
10        '''Return new Time object representing this time
11        incremented by the given number of seconds.'''
12        t = Time()
13        t = int_to_time(self.time_to_int() + seconds)
14        return t
15
16 t1.increment_pure(100, 200)
```

Here's an error you may encounter.
How the heck did `increment_pure`
get 3 arguments?!

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-55-1d8fb5e5c628> in <module>()
     14         return t
     15
----> 16 t1.increment_pure(100, 200)
```

Answer: the caller is considered an
argument (because of `self`!)

TypeError: increment_pure() takes 2 positional arguments but 3 were given

Recap: Objects, so far

So far: creating classes, attributes, methods

Next steps:

- How to implement operators (+, *, string conversion, etc)

- More complicated methods

- Inheritance

We will not come anywhere near covering OOP in its entirety

- My goal is only to make sure you see the general concepts

- Take a software engineering course to learn the deeper principles of OOP

Creating objects: the `__init__` method

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5     def __init__(self, hours=0, mins=0, secs=0):
6         self.hours = hours
7         self.mins = mins
8         self.secs = secs
9
10    def print_time(self):
11        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
12
13 t = Time(); t.print_time()
```

00:00:00

```
1 t = Time(10); t.print_time()
```

10:00:00

```
1 t = Time(10,20); t.print_time()
```

10:20:00

`__init__` is a special method that gets called when we instantiate an object. This one takes four arguments.

If we supply fewer than three arguments to `__init__`, it defaults the extras, assigning from left to right until it runs out of arguments.

Note: arguments that are not keyword arguments are called **positional arguments**.

Creating objects: the `__init__` method

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5     def __init__(self, hours=0, mins=0, secs=0):
6         self.hours = hours
7         self.mins = mins
8         self.secs = secs
9
10    def print_time(self):
11        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))
12
13    t = Time(); t.print_time()
```

00:00:00

```
1 t = Time(10); t.print_time()
```

10:00:00

```
1 t = Time(10,20); t.print_time()
```

10:20:00

Important point: notice how much cleaner this is than creating an object and then assigning attributes like we did earlier. Defining an `__init__` method also lets us ensure that there are certain attributes that are **always** populated in an object. This avoids the risk of an `AttributeError` sneaking up on us later. **Best practice** is to create all of the attributes that an object is going to have **at initialization**. Once again, Python allows you to do something, but it's best never to do it!

While we're on the subject...

Useful functions to know for debugging purposes: `vars` and `getattr`

```
1 for attr in vars(t1):  
2     print(attr, getattr(t1,attr))
```

`vars` returns a dictionary keyed on attribute names, values are attribute values.

```
hours 11  
mins 15  
secs 10
```

This is a useful pattern for debugging. Downey recommends encapsulating it in a function like `print_attrs(obj)`. I think this is a bit extreme. You should be using test cases and sanity checks to debug rather than examining the contents of objects.

Objects to strings: the `__str__` method

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5     def __init__(self, hours=0, mins=0, secs=0):
6         self.hours = hours
7         self.mins = mins
8         self.secs = secs
9
10    def __str__(self):
11        return "%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs)
12
13 t = Time(10,20,30)
14 print(t)
```

10:20:30

`__str__` is a special method that returns a string representation of the object. Print will always try to call this method via `str()`.

From the documentation: `str(object)` returns `object.__str__()`, which is the “informal” or nicely printable string representation of *object*. For string objects, this is the string itself. If *object* does not have a `__str__()` method, then `str()` falls back to returning `repr(object)`.
<https://docs.python.org/3.5/library/stdtypes.html#str>

Overloading operators

We can get other operators (+, *, /, comparisons, etc) by defining special functions

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
5
6
7
8
9
10
11
12
13     def time_to_int(self):
14         return(self.secs + 60*self.mins + 3600*self.hours)
15
16     def __add__(self, other):
17         '''Add other to this time, return result.'''
18         s = self.time_to_int() + other.time_to_int()
19         return(int_to_time(s))
20
21 t1 = Time(11,15,10); t2 = Time(1,5,1)
22 print(t1+t2)
```

`__init__` and `__str__`
cropped for space.

Defining the `__add__` operator lets us use + with `Time` objects. This is called **overloading** the + operator. All operators in Python have special names like this. More information: <https://docs.python.org/3/reference/datamodel.html#specialnames>

12:20:11

Type-based dispatch

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
```

Other methods
cropped for space.

```
15
16 def __add__(self, other):
17     '''Add other to this time, return result.'''
18     if isinstance(other, Time):
19         s = self.time_to_int() + other.time_to_int()
20         return(int_to_time(s))
21     elif isinstance(other, int):
22         s = self.time_to_int() + other
23         return(int_to_time(s))
24     else:
25         raise TypeError('Invalid type.')
26
27 t1 = Time(11,15,10)
28 print(t1 + 60)
```

`isinstance` returns True iff
its first argument is of the type
given by its second argument.

Depending on the type of `other`, our method
behaves differently. This is called **type-based
dispatch**. This is in keeping with Python's
general approach of always trying to do
something sensible with inputs.

```

1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
15
16     def __add__(self, other):
17         '''Add other to this time, return result.'''
18         if isinstance(other, Time):
19             s = self.time_to_int() + other.time_to_int()
20             return(int_to_time(s))
21         elif isinstance(other, int):
22             s = self.time_to_int() + other
23             return(int_to_time(s))
24         else:
25             raise TypeError('Invalid type.')
26
27 t1 = Time(11,15,10)
28 print(60 + t1)

```

Our + operator isn't commutative! This is because `int + Time` causes Python to call the `int.__add__` operator, which doesn't know how to add a `Time` to an `int`. We have to define a `Time.__radd__` operator for this to work.

```

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-10-18f9bcbbe091> in <module>()
     26
     27 t1 = Time(11,15,10)
--> 28 print(60 + t1)

TypeError: unsupported operand type(s) for +: 'int' and 'Time'

```

```
1 class Time:
2     '''Represents time on a 24 hour clock.
3     Attributes: int hours, int mins, int secs'''
4
```

```
15
16 def __add__(self, other):
17     '''Add other to this time, return result.'''
18     if isinstance(other, Time):
19         s = self.time_to_int() + other.time_to_int()
20         return(int_to_time(s))
21     elif isinstance(other, int):
22         s = self.time_to_int() + other
23         return(int_to_time(s))
24     else:
25         raise TypeError('Invalid type.')
26
27 t1 = Time(11,15,10)
28 print(60 + t1)
```

Our + operator isn't commutative! This is because `int + Time` causes Python to call the `int.__add__` operator, which doesn't know how to add a `Time` to an `int`. We have to define a `Time.__radd__` operator for this to work.

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-10-18f9bcbbe091> in <module>()
     26
     27 t1 = Time(11,15,10)
----> 28 print(60 + t1)
```

TypeError: unsupported operand type(s) for +: 'int' and 'Time'

Simple solution:

```
def __radd__(self, other):
    return self.__add__(other)
```

Polymorphism

Type-based dispatch is useful, but tedious

Better: write functions that work for many types

Examples:

String functions often work on tuples

int functions often work on floats or complex

Functions that work for many types are called **polymorphic**. Polymorphism is useful because it allows code reuse.

`hist` below is a good example of polymorphism. Works for all sequences!

```
1 def hist(s):
2     h = dict()
3     for x in s:
4         h[x] = h.get(x,0)+1
5     return h
6
7 hist('apple')
```

```
{'a': 1, 'e': 1, 'l': 1, 'p': 2}
```

```
1 hist((1,1,2,3,5,8))
```

```
{1: 2, 2: 1, 3: 1, 5: 1, 8: 1}
```

```
1 hist(list('gattaca'))
```

```
{'a': 3, 'c': 1, 'g': 1, 't': 2}
```


Interface and Implementation

Key distinction in object-oriented programming

Interface is the set of methods supplied by a class

Implementation is how the methods are actually carried out

Important point: ability to change implementation **without** affecting interface

Example: our `Time` class was represented by hour, minutes and seconds

Could have equivalently represented as seconds since midnight

In either case, we can write all the same methods (addition, conversion, etc)

Interface and Implementation

Key distinction in object-oriented programming

Interface is the set of methods supplied by a class

Implementation is how the methods are actually carried out

Important point: ability to change implementation **without** affecting interface

Example: our `Time` class was represented by hour, minutes and seconds

Could have equivalently represented as seconds since midnight

In either case, we can write all the same methods (addition, conversion, etc)

Certain implementations make certain operations easier than others.

Example: comparing two times in our hours, minutes, seconds representation is complicated, but if `Time` were represented as seconds since midnight, comparison becomes trivial. On the other hand, printing hh:mm:ss representation of a `Time` is complicated if our implementation is seconds since midnight.

Inheritance

Inheritance is perhaps the most useful feature of object-oriented programming

Inheritance allows us to create new Classes from old ones

Our running example for this will follow Downey's chapter 18

Objects are playing cards, hands and decks

Assumes some knowledge of Poker <https://en.wikipedia.org/wiki/Poker>

52 cards in a deck

4 suits: Spades > Hearts > Diamonds > Clubs

13 ranks: Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King

Creating our class

A card is specified by its suit and rank, so those will be the attributes of the card class. The default card will be the two of clubs.

```
1 class Card:
2     '''Represents a playing card'''
3     def __init__(suit=0,rank=2):
4         self.suit = suit
5         self.rank = rank
```

This stage of choosing how you will represent objects (and what objects to represent) is often the most important part of the coding process. It's well worth your time to carefully plan and design your objects, how they will be represented and what methods they will support.

We will encode suits and ranks by numbers, rather than strings. This will make comparison easier.

Suit encoding

0 : Clubs
1 : Diamonds
2 : Hearts
3 : Spades

Rank encoding

0 : None
1 : Ace
2 : 2
3 : 3
...
10 : 10
11 : Jack
12 : Queen
13 : King

Creating our class

```
1 class Card:
2     '''Represents a playing card'''
3
4     suit_names = ['Spades', 'Hearts', 'Diamonds', 'Clubs']
5     rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
6                   '8', '9', '10', 'Jack', 'Queen', 'King']
7
8     def __init__(self, suit=0, rank=2):
9         self.suit = suit
10        self.rank = rank
11
12    def __str__(self):
13        rankstr = self.rank_names[self.rank]
14        suitstr = self.suit_names[self.suit]
15        return "%s of %s" % (rankstr, suitstr)
16
17 print(Card(0,1))
```

Variables defined in a class but outside any method are called **class attributes**. They are shared across all instances of the class.

Instance attributes are assigned to a specific object (e.g., `rank` and `suit`). Both class and instance attributes are accessed via dot notation.

Here we use instance attributes to index into class attributes.

Ace of Spades

Creating our class



Ace of Spades

```
1 class Card:
    """Represents a playing card"""
    suits = ['Spades', 'Hearts', 'Diamonds', 'Clubs']
    ranks = [None, 'Ace', '2', '3', '4', '5', '6', '7',
             '8', '9', '10', 'Jack', 'Queen', 'King']

    def __init__(self, suit=0, rank=2):
        self.suit = suits[suit]
        self.rank = ranks[rank]

    def __str__(self):
        rankstr = self.rank_names[self.rank]
        suitstr = self.suit_names[self.suit]
        return "%s of %s" % (rankstr, suitstr)

    def __repr__(self):
        return "Card(%s, %s)" % (self.rank, self.suit)

    def __eq__(self, other):
        return self.rank == other.rank and self.suit == other.suit
```

Variables defined in a class but outside any method are called **class attributes**. They are shared across all instances of the class.

Instance attributes are assigned to a specific object (e.g., `rank` and `suit`). Both class and instance attributes are accessed via dot notation.

Here we use instance attributes to index into class attributes.

[https://en.wikipedia.org/wiki/Ace_of_Spades_\(song\)](https://en.wikipedia.org/wiki/Ace_of_Spades_(song))

More operators

```
1 class Card:  
2     '''Represents a playing card'''  
3
```

Cropped for space.

```
12     def __lt__(self, other):  
13         t1 = (self.rank, self.suit)  
14         t2 = (other.rank, other.suit)  
15         return t1 < t2  
16  
17     def __gt__(self, other):  
18         return other < self  
19  
20     def __eq__(self, other):  
21         return (self.rank==other.rank and self.suit==other.suit)  
22 c1 = Card(2,11); c2 = Card(2,12)  
23 c1 < c2
```

We've chosen to order cards based on rank and then suit, with aces low. So a jack is bigger than a ten, regardless of the suit of either one. Downey orders by suit first, then rank.

True

```
1 c1 == Card(2,11)
```

Now that we've defined the `__eq__` operator, we can check for equivalence correctly.

True

Objects with other objects

```
1 class Deck:
2     '''Represents a deck of cards'''
3     def __init__(self):
4         self.cards = list()
5         for suit in range(4):
6             for rank in range(1,14):
7                 card = Card(suit,rank)
8                 self.cards.append(card)
9
10    def __str__(self):
11        res = list()
12        for c in self.cards:
13            res.append(str(c))
14        return('\n'.join(res))
15
16 d = Deck()
17 print(d)
```

Define a new object representing a deck of cards. A standard deck of playing cards is 52 cards, four suits, 13 ranks per suit, etc.

Represent cards in the deck via a list. To populate the list, just use a nested for-loop to iterate over suits and ranks.

String representation of a deck will just be the cards in the deck, in order, one per line. Note that this produces a **single string**, but it includes newline characters.

There's another 45 or so more strings down there...

```
Ace of Spades
2 of Spades
3 of Spades
4 of Spades
5 of Spades
6 of Spades
```


Providing additional methods

```
1 import random
2 class Deck:
3     '''Represents a deck of cards'''
```

```
17     def pop_card(self):
18         return(self.cards.pop())
19     def add_card(self,c):
20         self.cards.append(c)
21     def shuffle(self):
22         random.shuffle(self.cards)
```

```
1 d = Deck()
2 d.shuffle()
3 print(d)
```

```
2 of Hearts
9 of Clubs
Ace of Spades
3 of Clubs
6 of Spades
```

One method for dealing a card off the “top” of the deck, and one method for adding a card back to the “top” of the deck.

Note: methods like this that are really just wrappers around other existing methods are often called **veneer** or **thin methods**.

After shuffling, the cards are not in the same order as they were on initialization.

Let's take stock

We have:

- a class that represents playing cards (and some basic methods)

- a class that represents a deck of cards (and some basic methods)

Now, the next logical thing we want is a class for representing a hand of cards

So we can actually represent a game of poker, hearts, bridge, etc.

The naïve approach would be to create a new class Hand from scratch

But a more graceful solution is to use **inheritance**

Key observation: a hand is a lot like a deck (it's a collection of cards)

...of course, a hand is also different from a deck in some ways...

Inheritance

This syntax means that the class `Hand` **inherits** from the class `Deck`. Inheritance means that `Hand` has all the same methods and class attributes as `Deck` does.

```
1 class Hand(Deck):  
2     '''Represents a hand of cards'''  
3  
4 h = Hand()  
5 h.shuffle()  
6 print(h)
```

We say that the **child** class `Hand` inherits from the **parent** class `Deck`.

```
Ace of Clubs  
Queen of Diamonds  
9 of Hearts  
King of Hearts  
8 of Clubs  
8 of Hearts  
Queen of Clubs  
3 of Diamonds  
5 of Hearts  
7 of Clubs  
King of Diamonds
```

Inheritance

This syntax means that the class `Hand` **inherits** from the class `Deck`. Inheritance means that `Hand` has all the same methods and class attributes as `Deck` does.

```
1 class Hand(Deck):  
2     '''Represents a hand of cards'''  
3  
4 h = Hand()  
5 h.shuffle()  
6 print(h)
```

We say that the **child** class `Hand` inherits from the **parent** class `Deck`.

```
Ace of Clubs  
Queen of Diamonds  
9 of Hearts  
King of Hearts  
8 of Clubs  
8 of Hearts  
Queen of Clubs  
3 of Diamonds  
5 of Hearts  
7 of Clubs  
King of Diamonds
```

So, for example, `Hand` has `__init__` and `shuffle` methods, and they are identical to those in `Deck`. Of course, we quickly see that the `__init__` inherited from `Deck` isn't quite what we want for `Hand`. A hand of cards isn't usually the entire deck...

So we already see the ways in which inheritance can be useful, but we also see immediately that there's no free lunch here. We will have to **override** the `__init__` function inherited from `Deck`.

Inheritance: methods and overriding

```
1 class Hand(Deck):  
2     '''Represents a hand of cards'''  
3  
4     def __init__(self, label=''):  
5         self.cards = list()  
6         self.label=label
```

Redefining the `__init__` method overrides the one inherited from `Deck`.

Inheritance: methods and overriding

```
1 class Hand(Deck):
2     '''Represents a hand of cards'''
3
4     def __init__(self, label=''):
5         self.cards = list()
6         self.label=label
7
8 h = Hand('new hand')
9 d = Deck(); d.shuffle()
10 h.add_card(d.pop_card())
11 print(h)
```

Redefining the `__init__` method overrides the one inherited from `Deck`.

Now, when we initialize a `Hand` object, it starts out empty.

Simple way to deal a single card from the deck to the hand: pop a card off of the deck, add it to the hand.

6 of Spades

Inheritance: methods and overriding

```
1 import random
2 class Deck:
3     '''Represents a deck of cards'''
23
24     def move_cards(self, hand, ncards):
25         for i in range(ncards):
26             hand.add_card(self.pop_card())
```

Encapsulate this pattern in a method supplied by `Deck`, and we have a method that deals cards to a hand.

```
1 d = Deck(); d.shuffle()
2 h = Hand()
3 d.move_cards(h,5)
4 print(h)
```

The `move_cards` method is supplied by `Deck` but it modifies both the caller and the `Hand` object in the first argument.

```
2 of Spades
King of Spades
9 of Diamonds
2 of Diamonds
7 of Clubs
```

Note: `Hand` also inherits the `move_cards` method from `Deck`, so we have a way to move cards from one hand to another (e.g., as at the beginning of a round of hearts)

Inheritance: pros and cons

Pros:

- Makes for simple, fast program development

- Enables code reuse

- Often reflects some natural structure of the problem

Cons:

- Can make debugging challenging (e.g., where did this method come from?)

- Code gets spread across multiple classes

- Can accidentally override (or forget to override) a method

A Final Note on OOP

Object-oriented programming is ubiquitous in software development

Useful when designing large systems with many interacting parts

As a statistician, most systems you build are... not so complex

(At least not in the sense of requiring lots of interacting subsystems)

We've only scratched the surface of OOP

Not covered: factories, multiple inheritance, abstract classes...

Take a software engineering course to learn more about this

In my opinion, OOP isn't especially useful for data scientists, anyway.

This isn't to say that *objects* aren't useful, only OOP as a paradigm

Understanding functional programming is far more important (next lecture)