STATS 507
Data Analysis in Python

Lecture 5: Files, Classes, Operators and Inheritance

Persistent data

So far, we only know how to write “transient” programs

- Data disappears once the program stops running

Files allow for **persistence**

- Work done by a program can be saved to disk...
- ...and picked up again later for other uses.

Examples of persistent programs:

- Operating systems
- Databases
- Servers

Key idea: Program information is stored permanently (e.g., on a hard drive), so that we can start and stop programs without losing **state** of the program (values of variables, where we are in execution, etc).

Reading and Writing Files

Underlyingly, every file on your computer is just a string of bits…

0	1	1	0	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	1	0	1	0	0

...which are broken up into (for example) bytes…

0 1 1 0 0 0 1 1	0 1 1 0 0 0 0 1	0 1 1 1 0 1 0 0

...which correspond (in the case of text) to characters.

0 1 1 0 0 0 1 1	0 1 1 0 0 0 0 1	0 1 1 1 0 1 0 0
c	a	t

Reading files

This is the command line. We'll see lots more about this later, but for now, it suffices to know that the command `cat` prints the contents of a file to the screen.

```
keith@Steinhaus:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
keith@Steinhaus:~/demo$
```

```
f = open('demo.txt')
type(f)
```

```
_io.TextIOWrapper
```

Open the file `demo.txt`. This creates a **file object** `f`.
https://docs.python.org/3/glossary.html#term-file-object

```
f.readline()
```

```
'This is a demo file.\n'
```

Provides a method for reading a single line from the file. The string `'\n'` is a **special character** that represents a new line. More on this soon.

Reading files

```
keith@Steinhaus:~/demo$ cat demo.txt
This is a demo file.
It is a text file, containing three lines of text.
Here is the third line.
keith@Steinhaus:~/demo$
```

```
1 f = open('demo.txt')
2 f.readline()
```

```
'This is a demo file.\n'
```

```
1 f.readline()
```

```
'It is a text file, containing three lines of text.\n'
```

```
1 f.readline()
```

```
'Here is the third line.\n'
```

```
1 f.readline()
```

```
''
```

Each time we call `f.readline()`, we get the next line of the file...

...until there are no more lines to read, at which point the `readline()` method returns the empty string whenever it is called.

Reading files

```
f = open('demo.txt')
for line in f:
    for wd in line.split():
        print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

We can treat `f` as an iterator, in which each iteration gives us a line of the file.

Iterate over each word in the line (splitting on ‘ ’ by default).

Remove the trailing punctuation from the words of the file.

`open()` provides a bunch more (optional) arguments, some of which we’ll discuss later.
https://docs.python.org/3/library/functions.html#open

Reading files

```
with open('demo.txt') as f:
    for line in f:
        for wd in line.split():
            print(wd.strip('.,'))
```

```
This
is
a
demo
file
It
is
a
text
file
containing
three
lines
of
text
Here
is
the
third
line
```

You may often see code written this way, using the `with` keyword. We'll see it in detail later. For now, it suffices to know that this is equivalent to what we did on the previous slide.

From the documentation: "It is good practice to use the with keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point."
https://docs.python.org/3/reference/compound_stmts.html#with

In plain English: the `with` keyword does a bunch of error checking and cleanup for you, automatically.

Writing files

Open the file in **write** mode. If the file already exists, this creates it anew, deleting its old contents.

```
f = open('animals.txt', 'w')
f.read()
```

If I try to read a file in write mode, I get an error.

```
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
<ipython-input-29-3b1ef477003a> in <module>()
      1 f = open('animals.txt', 'w')
----> 2 f.read()

UnsupportedOperation: not readable
```

```
f.write('cat\n')
f.write('dog\n')
f.write('bird\n')
f.write('goat\n')
```

```
5
```

Write to the file. This method returns the number of characters written to the file. Note that '\n' counts as a single character, the new line.

Writing files

```python
f = open('animals.txt', 'w')
f.write('cat\n')
f.write('dog\n')
f.write('bird\n')
f.write('goat\n')
f.close()
```

Open the file in **write** mode.
This overwrites the version of the file created in the previous slide.

Each write appends to the end of the file.

When we're done, we close the file. This happens automatically when the program ends, but its good practice to close the file as soon as you're done.

```python
f = open('animals.txt', 'r')
for line in f:
    print(line, end="")
```

```
cat
dog
bird
goat
```

Now, when I open the file for reading, I can print out the lines one by one.

The lines of the file already include newlines on the ends, so override Python's default behavior of printing a newline after each line.

Aside: Formatting Strings

```
1 x = 23
2 print('x = %d' % x)
```

```
x = 23
```

Python provides tools for formatting strings. Example: easier way to print an integer as a string.

```
1 animal = 'unicorn'
2 print('My pet %s' % animal)
```

```
My pet unicorn
```

`%d` : integer
`%s` : string
`%f` : floating point
More information:
https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

```
1 x = 2.718; y = 1.618
2 print('%f divided by %f is %f' % (x,y,x/y))
```

```
2.718000 divided by 1.618000 is 1.679852
```

```
1 print('%.3f divided by %.3f is %.8f' % (x,y,x/y))
```

```
2.718 divided by 1.618 is 1.67985167
```

Can further control details of formatting, such as number of significant figures in printing floats.

Newer features for similar functionality:
https://docs.python.org/3/reference/lexical_analysis.html#f-strings
https://docs.python.org/3/library/stdtypes.html#str.format

Aside: Formatting Strings

Note: Number of formatting arguments must match the length of the supplied tuple!

```
x = 2.718; y = 1.618
print('%f divided by %f is %f' % (x,y,x/y,1.0))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-eb736fce3612> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y,x/y,1.0))

TypeError: not all arguments converted during string formatting
```

```
x = 2.718; y = 1.618
print('%f divided by %f is %f' % (x,y))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-47-b2e6a26d3415> in <module>()
      1 x = 2.718; y = 1.618
----> 2 print('%f divided by %f is %f' % (x,y))

TypeError: not enough arguments for format string
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
import pickle
t1 = [1,'two',3.0]
s = pickle.dumps(t1)
s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

`pickle.dumps()` (short for "dump string") creates a **binary string** representing an object.

This is a raw binary string that encodes the list `t1`. Each symbol encodes one byte. More detail later in the course.
https://docs.python.org/3.6/library/functions.html#func-bytes
https://en.wikipedia.org/wiki/ASCII

```
t2 = pickle.loads(s)
t1==t2
```

```
True
```

```
t1 is t2
```

```
False
```

Saving objects to files: `pickle`

Sometimes it is useful to be able to turn an object into a string

```
import pickle
t1 = [1,'two',3.0]
s = pickle.dumps(t1)
s
```

```
b'\x80\x03]q\x00(K\x01X\x03\x00\x00\x00twoq\x01G@\x08\x00\x00\x00\x00\x00\x00e.'
```

We can now use this string to store (a representation of) the list referenced by `t1`. We can write it to a file for later reuse, use it as a key in a dictionary, etc.

```
t2 = pickle.loads(s)
t1==t2
```

```
True
```

Later on, to “unpickle” the string and turn it back into an object, we use `pickle.loads()` (short for “load string”).

Important point: pickling stores a representation of the value, not the variable! So after this assignment, `t1` and `t2` are equivalent...

```
t1 is t2
```

```
False
```

...but not identical.

Locating files: the `os` module

```
import os
cwd = os.getcwd()
cwd
```

```
'/Users/keith/demo/L6_Files'
```

```
os.listdir()
```

```
['data', 'scripts']
```

```
os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

```
os.chdir('data')
os.getcwd()
```

```
'/Users/keith/demo/L6_Files/data'
```

`os` module lets us interact with the operating system.
https://docs.python.org/3.6/library/os.html

`os.getcwd()` returns a string corresponding to the **current working directory**.

`os.listdir()` lists the contents of its argument, or the current directory if no argument.

`os.chdir()` changes the working directory. After calling `chdir()`, we're in a different cwd.

Locating files: the `os` module

```
import os
cwd = os.getcwd()
cwd
```

```
'/Users/keith/demo/L6_Files'
```

This is called a **path**. It starts at the **root directory**, `'/'`, and describes a sequence of nested directories.

```
os.listdir()
```

```
['data', 'scripts']
```

```
os.listdir('data')
```

```
['numbers.txt', 'pi.txt']
```

A path from the root to a file or directory is called an **absolute path**. A path from the current directory is called a **relative path**.

```
os.path.abspath('data/pi.txt')
```

```
'/Users/keith/demo/L6_Files/data/pi.txt'
```

Use `os.path.abspath` to get the absolute path to a file or directory.

Locating files: the `os` module

```
import os
os.chdir('/Users/keith/demo/L6_Files')
os.listdir('data')
```

```
['extra', 'numbers.txt', 'pi.txt']
```

```
os.path.exists('data/pi.txt')
```

```
True
```

```
os.path.exists('data/nonsense.txt')
```

```
False
```

Check whether or not a file/directory exists.

```
os.path.isdir('data/extra')
```

```
True
```

```
os.path.isdir('data/numbers.txt')
```

```
False
```

Check whether or not this is a directory. `os.path.isfile()` works analogously.

Handling errors: try/catch statements

Sometimes when an error occurs, we want to try and recover

Rather than just giving up and having Python yell at us.

Python has a special syntax for this: `try:... except:...`

Basic idea: try to do something, and if an error occurs, try something else.

Example: try to open a file for reading.

If that fails (e.g., because the file doesn’t exist) look for the file elsewhere

Handling errors: try/catch statements

```
import os
os.listdir()
```

```
['backup_file.txt', 'data', 'scripts']
```

```
try:
    f = open('nonsense.txt')
except:
    f = open('backup_file.txt')
f.read()
```

```
'This is a backup file.\n'
```

Python attempts to execute the code in the `try` block. If that runs successfully, then we continue on.

If the `try` block fails (i.e., if there's an **exception**), then we run the code in the `except` block.

Programmers call this kind of construction a **try/catch statement**, even though the Python syntax uses `try/except` instead.

Handling errors: try/catch statements

```
import os
os.listdir()
```

```
['backup_file.txt',
```

```
try:
    f = open('nonsense.txt')
except:
    f = open('backup_file.txt')
f.read()
```

```
'This is a backup file.\n'
```

Note: this pattern is really only necessary in particular situations where you know how you want to recover from the error. Otherwise, it's better to just raise an error. I show it here because you'll see this pattern frequently "in the wild".

…cute the code in …ns successfully, then we continue on.

If the `try` block fails (i.e., if there's an **exception**), then we run the code in the `except` block.

Programmers call this kind of construction a **try/catch statement**, even though the Python syntax uses `try/except` instead.

Writing modules

Python provides modules (e.g., `math`, `os`, `time`)

But we can also write our own, and import from them with same syntax

```
import prime
prime.is_prime(2)
```

```
True
```

```
prime.is_prime(3)
```

```
True
```

```
prime.is_prime(1)
```

```
False
```

```
prime.is_prime(23)
```

```
True
```

prime.py

```python
import math

def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True
```

Writing modules

```
from prime import *
is_prime(7)
```

```
True
```

```
is_square(7)
```

```
False
```

```
is_prime(373)
```

```
True
```

Import everything defined in `prime`, so we can call it without the prefix. Can also import specific functions:
`from prime import is_square`

prime.py

```
import math

def is_prime(n):
    if n <= 1:
        return False
    elif n==2:
        return True
    else:
        ulim = math.ceil(math.sqrt(n))
        for k in range(2,ulim+1):
            if n%k==0:
                return False
        return True
def is_square(n):
    r = int(math.sqrt(n))
    return(r*r==n or (r+1)*(r+1)==n)
```

Caution: be careful that you don't cause a collision with an existing function or a function in another module!

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

Option 2: make a point **class**, so that we have a whole new data type

Additional good reasons for this will become apparent shortly!

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

Class header declares a new class, called `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.

```
print(Point)
```

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

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

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!

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

```
print(Point)
```

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

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

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

Creating an object: Instantiation

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

p = Point()
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

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

p = Point()
p
```

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

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

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

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.

```
p = Point()
p.x = 3.0
p.y = 4.0
(p.x,p.y)
```

```
(3.0, 4.0)
```

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

```
p.goat
```

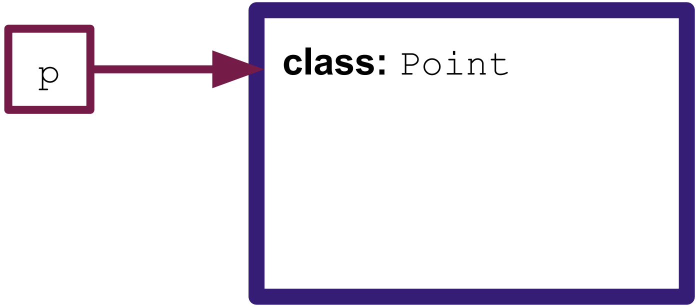
Attempting to access an attribute that an object doesn't have is an error.

```
---------------------------------------------------------------------------
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

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

p = Point()
p.x = 3.0
p.y = 4.0
```

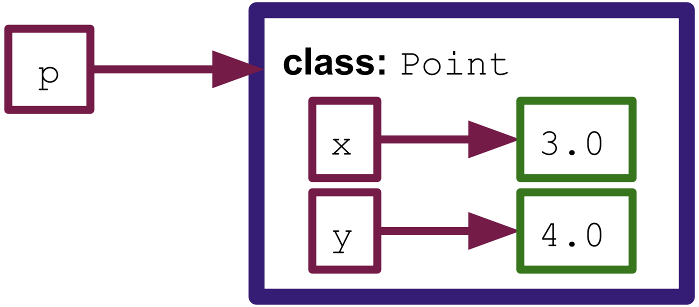
At this point, `p` is just an object with no attributes.



Thinking about Attributes: Object Diagrams

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

p = Point()
p.x = 3.0
p.y = 4.0
```

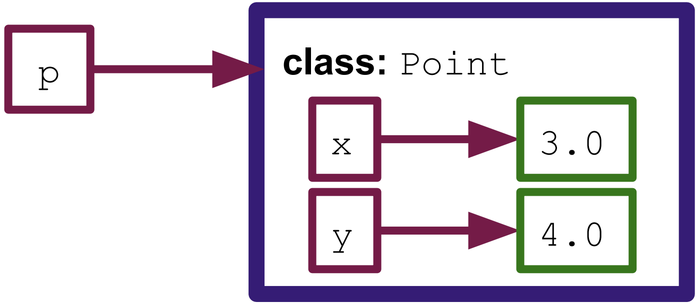
After these two lines, `p` has attributes `x` and `y`.



Thinking about Attributes: Object Diagrams

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

p = Point()
p.x = 3.0
p.y = 4.0
```

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

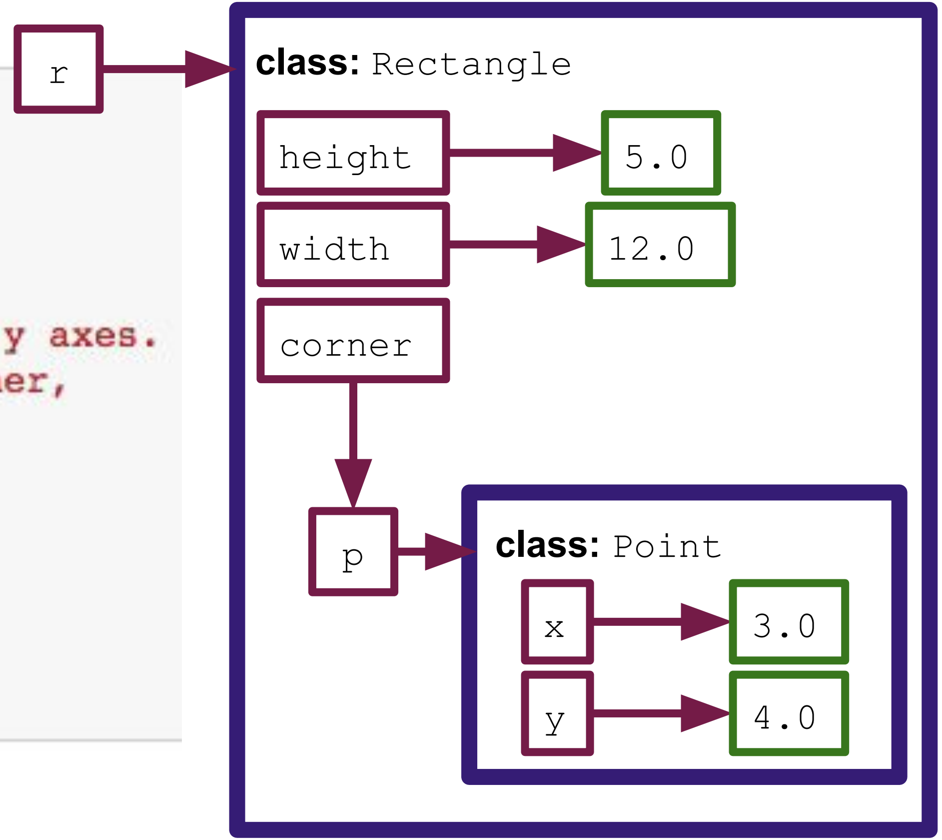


So dot notation `p.x`, essentially says, look inside the object `p` and find the attribute `x`.

Nesting Objects

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

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

class Rectangle:
    '''Represents a rectangle whose
    sides are parallel to the x and y axes.
    Specified by its upper-left corner,
    height, and width.'''

p = Point(); p.x = 3.0; p.y = 4.0
r = Rectangle()
r.corner = p
r.height = 5.0
r.width = 12.0
```



Nesting Objects

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0
r1.width = 12.0

r2 = Rectangle()
r2.corner = Point()
r2.corner.x = 3.0
r2.corner.y = 4.0
r2.height = 5.0
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

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r1.height = 2*r1.height

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dx

shift_rectangle(r1, 2, 3)
(r1.corner.x, r1.corner.y)
```

```
(5.0, 6.0)
```

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.

Returning Objects

```
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0
r1.width = 12.0

r2 = double_sides(r1)
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

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

```
False
```

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

```
p1 == p2
```

```
False
```

...but they should **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

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

```
False
```

The
cop
they

```
p1 == p2
```

```
False
```

...bu

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.

Copying and Aliasing

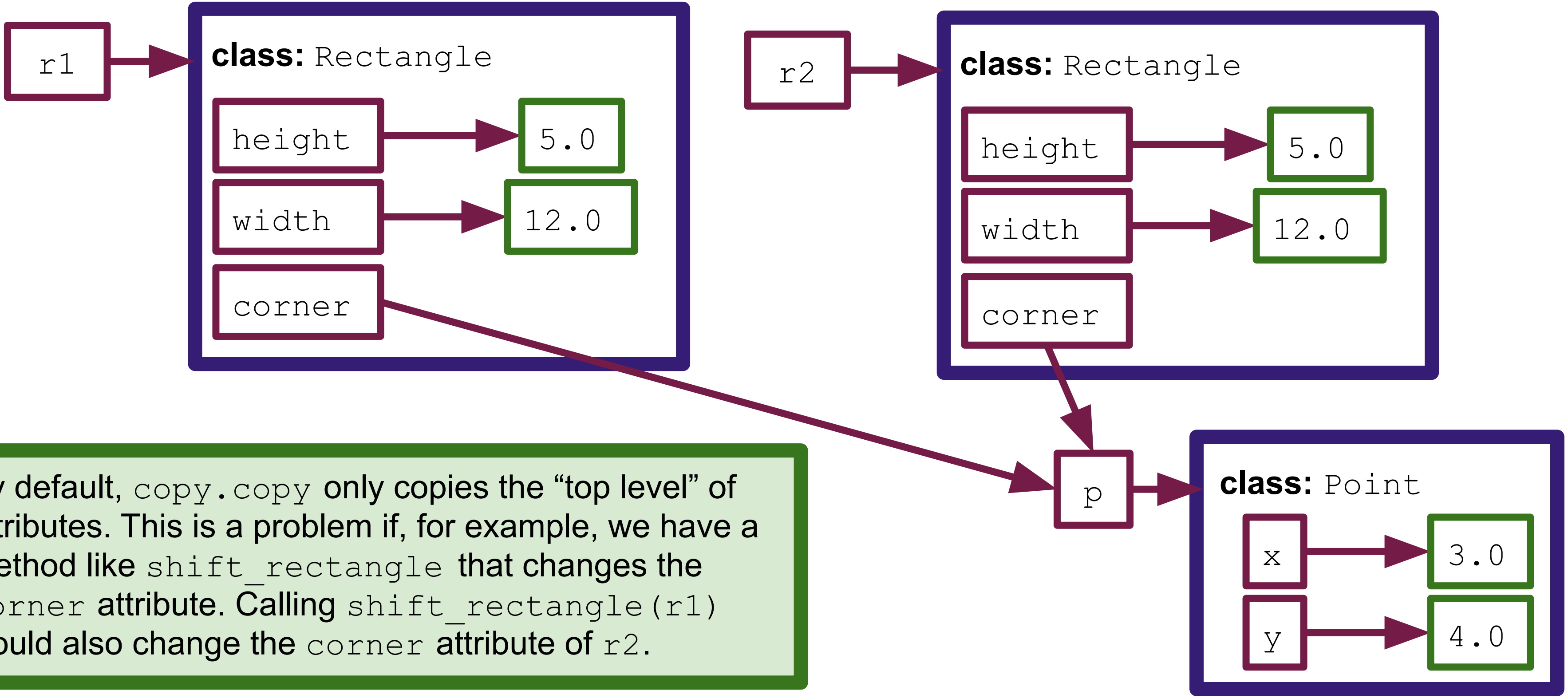
```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r2 = copy.copy(r1)

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



By default, `copy.copy` only copies the "top level" of attributes. This is a problem if, for example, we have a method like `shift_rectangle` that changes the `corner` attribute. Calling `shift_rectangle(r1)` would also change the `corner` attribute of `r2`.

Copying and Aliasing

```
p1 = Point(); p1.x = 3.0; p1.y = 4.0
r1 = Rectangle()
r1.corner = p1
r1.height = 5.0; r1.width = 12.0
r2 = copy.deepcopy(r1)

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

```
def double_sides(r):
    rdouble = Rectangle()
    rdouble.corner = r.corner
    rdouble.height = 2*r.height
    rdouble.width = 2*r.width
    return(rdouble)

def shift_rectangle(rec, dx, dy):
    rec.corner.x = rec.corner.x + dx
    rec.corner.y = rec.corner.y + dx
```

`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)

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.

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

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time()
t.hours=12; t.mins=34; t.secs=56
t.print_time()
```

```
12:34:56
```

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.

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

More on Methods

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

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)

def int_to_time(seconds):
    '''Convert a number of seconds to a Time object.'''
    t = Time()
    (minutes, t.secs) = divmod(seconds,60)
    (hrs, t.mins) = divmod(minutes,60)
    t.hours = hrs % 24 #military time!
    return t

t = int_to_time(1337)
t.time_to_int()
```

```
1337
```

`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

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

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

    def increment_modifier(self, seconds):
        '''Increment this time by the given
        number of seconds.'''
        (mins, self.secs) = divmod(self.secs+seconds, 60)
        (hours, self.mins) = divmod(self.mins+mins, 60)
        self.hours = (self.hours + hours)%24

t1 = int_to_time(1234)
t1.increment_modifier(1111)
t1.time_to_int()
```

```
2345
```

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

```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''
    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)
    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

    def increment_pure(self, seconds):
        '''Return new Time object representing this time
        incremented by the given number of seconds.'''
        t = Time()
        t = int_to_time(self.time_to_int() + seconds)
        return t

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
```

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

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

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

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

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

```
00:00:00
```

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

```
10:00:00
```

```
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

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

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def print_time(self):
        print("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

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

```
00:00:00
```

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

```
10:00:00
```

```
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`

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

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

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

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

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __init__(self, hours=0, mins=0, secs=0):
        self.hours = hours
        self.mins = mins
        self.secs = secs

    def __str__(self):
        return("%.2d:%.2d:%.2d" % (self.hours, self.mins, self.secs))

t = Time(10,20,30)
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

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

    def time_to_int(self):
        return(self.secs + 60*self.mins + 3600*self.hours)

    def __add__(self, other):
        '''Add other to this time, return result.'''
        s = self.time_to_int() + other.time_to_int()
        return(int_to_time(s))

t1 = Time(11,15,10); t2 = Time(1,5,1)
print(t1+t2)
```

```
12:20:11
```

`__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

Type-based dispatch

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''


    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
print(t1 + 60)
```

```
11:16:10
```

Other methods cropped for space.

`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.

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''

    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
print(60 + t1)
```

```
---------------------------------------------------------------------------
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'
```

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.

```python
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''


    def __add__(self, other):
        '''Add other to this time, return result.'''
        if isinstance(other,Time):
            s = self.time_to_int() + other.time_to_int()
            return(int_to_time(s))
        elif isinstance(other,int):
            s = self.time_to_int() + other
            return(int_to_time(s))
        else:
            raise TypeError('Invalid type.')

t1 = Time(11,15,10)
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 (m
<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:

```python
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!

```
def hist(s):
    h = dict()
    for x in s:
        h[x] = h.get(x,0)+1
    return h

hist('apple')
```

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

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

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

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

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

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.

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

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

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.

Creating our class

```
class Card:
    '''Represents a playing card'''

    suit_names = ['Spades','Hearts','Diamonds','Clubs']
    rank_names = [None,'Ace','2','3','4','5','6','7',
                  '8','9','10','Jack','Queen','King']

    def __init__(self,suit=0,rank=2):
        self.suit = suit
        self.rank = rank

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

print(Card(0,1))
```

```
Ace of Spades
```

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.

Creating our class

```
class Card:
    '''R       ents a playing card'''

          s = ['Spades','Hearts','Diamonds','Clubs']
          s = [None,'Ace','2','3','4','5','6','7',
                 '8','9','10','Jack','Queen','King']

          t__(self,suit=0,rank=2):
          suit = suit
          rank = rank

          __(self):
          tr = self.rank_names[self.rank]
          tr = self.suit_names[self.suit]
          n("%s of %s" % (rankstr, suitstr))

          1))
```

```
Ace of Spades
```

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)

More operators

```
class Card:
    '''Represents a playing card'''

    def __lt__(self, other):
        t1 = (self.rank, self.suit)
        t2 = (other.rank, other.suit)
        return t1 < t2

    def __gt__(self, other):
        return other < self

    def __eq__(self, other):
        return(self.rank==other.rank and self.suit==other.suit)
c1 = Card(2,11); c2 = Card(2,12)
c1 < c2
```

```
True
```

Cropped for space.

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.

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

```
True
```

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

Objects with other objects

```
class Deck:
    '''Represents a deck of cards'''
    def __init__(self):
        self.cards = list()
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit,rank)
                self.cards.append(card)

    def __str__(self):
        res = list()
        for c in self.cards:
            res.append(str(c))
        return('\n'.join(res))

d = Deck()
print(d)
```

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

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...

Providing additional methods

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

```
    def pop_card(self):
        return(self.cards.pop())
    def add_card(self,c):
        self.cards.append(c)
    def shuffle(self):
        random.shuffle(self.cards)
```

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

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

```
d = Deck()
d.shuffle()
print(d)
```

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

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

```
class Hand(Deck):
    '''Represents a hand of cards'''

h = Hand()
h.shuffle()
print(h)
```

```
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
```

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.

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

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

```
class Hand(Deck):
    '''Represents a hand of cards'''

    def __init__(self, label=''):
        self.cards = list()
        self.label=label

h = Hand('new hand')
d = Deck(); d.shuffle()
h.add_card(d.pop_card())
print(h)
```

```
6 of Spades
```

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

Simple way to deal a single card from the deck to the hand.

Inheritance: methods and overriding

```python
import random
class Deck:
    '''Represents a deck of cards'''

    def move_cards(self, hand, ncards):
        for i in range(ncards):
            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.

```python
d = Deck(); d.shuffle()
h = Hand()
d.move_cards(h,5)
print(h)
```

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

Note that this method is supplied by `Deck` but it modifies both the caller and the `Hand` object in the first argument.

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)