STATS 701
Data Analysis using Python

Lecture 8: Operators and Inheritance

Recap: Objects, so far

Previous lecture: creating classes, attributes, methods

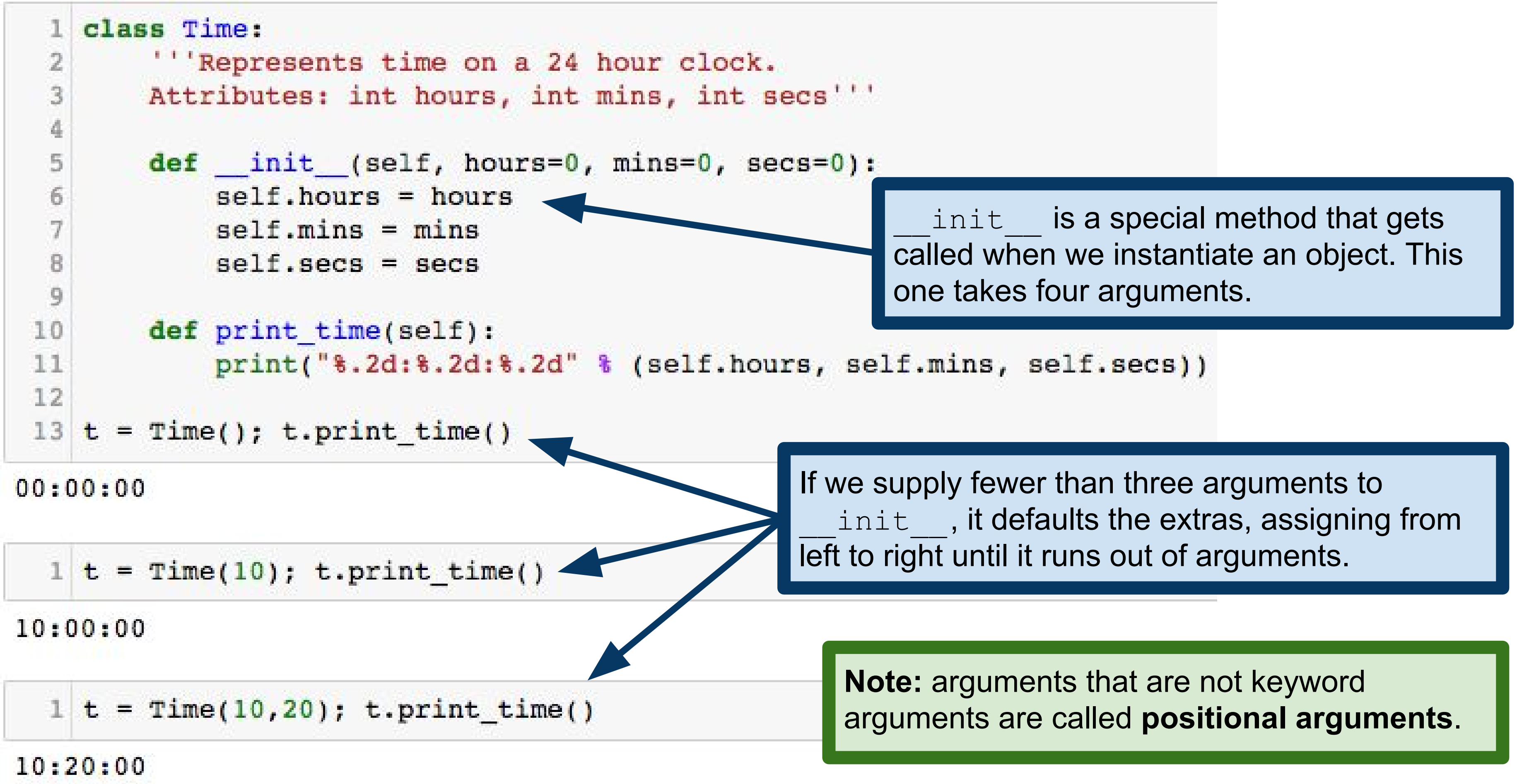
This lecture: next steps

- 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



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

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

```
00:00:00
```

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

```
10:00:00
```

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

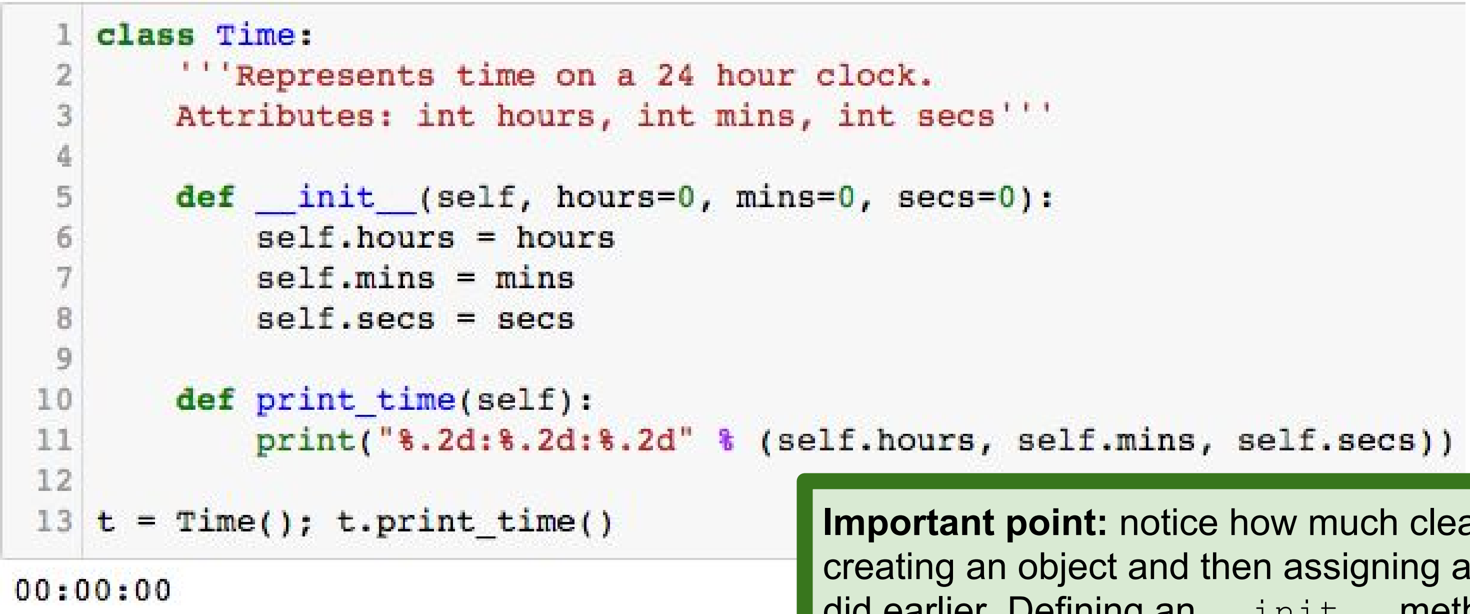
```
10:20:00
```

`__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.

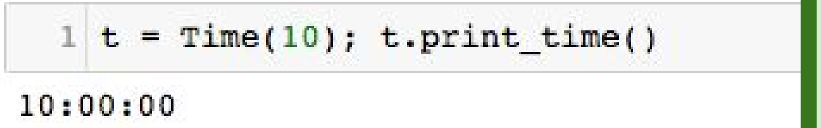
Note: arguments that are not keyword arguments are called **positional arguments**.

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()
```

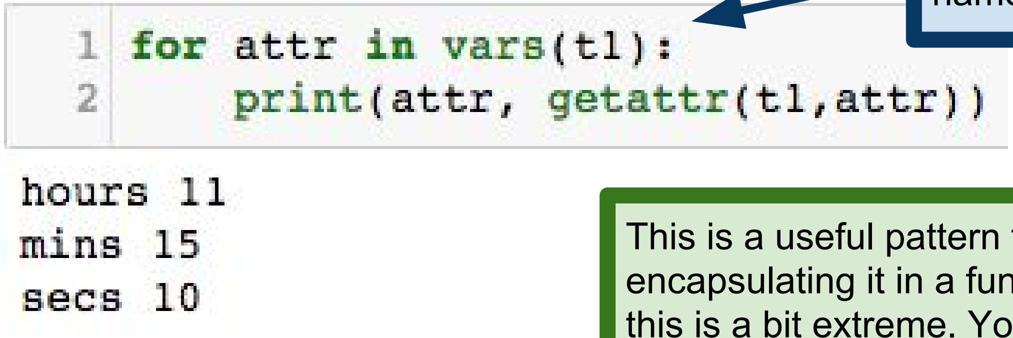
```
10:20:00
```

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`

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

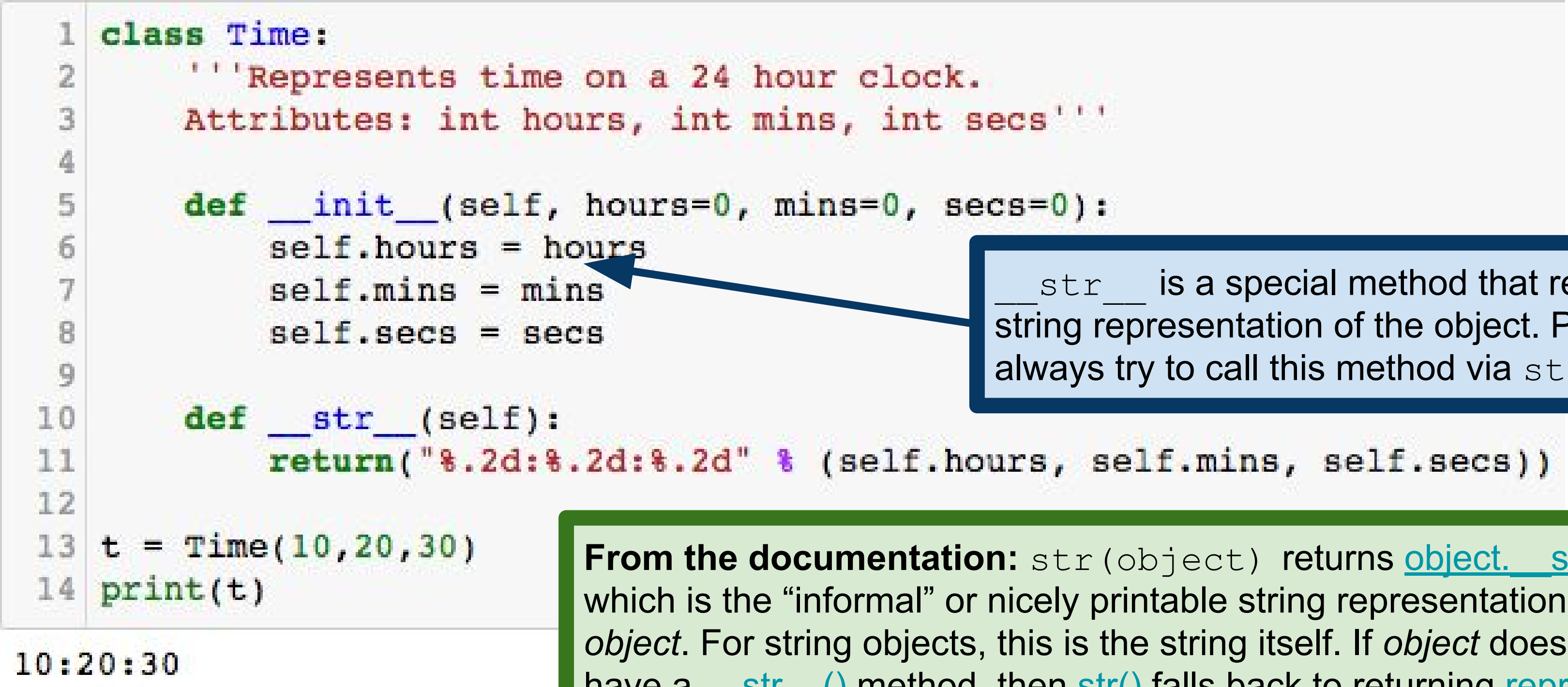


```python
for attr in vars(t1):
    print(attr, getattr(t1,attr))
```

```
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



```
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)
```

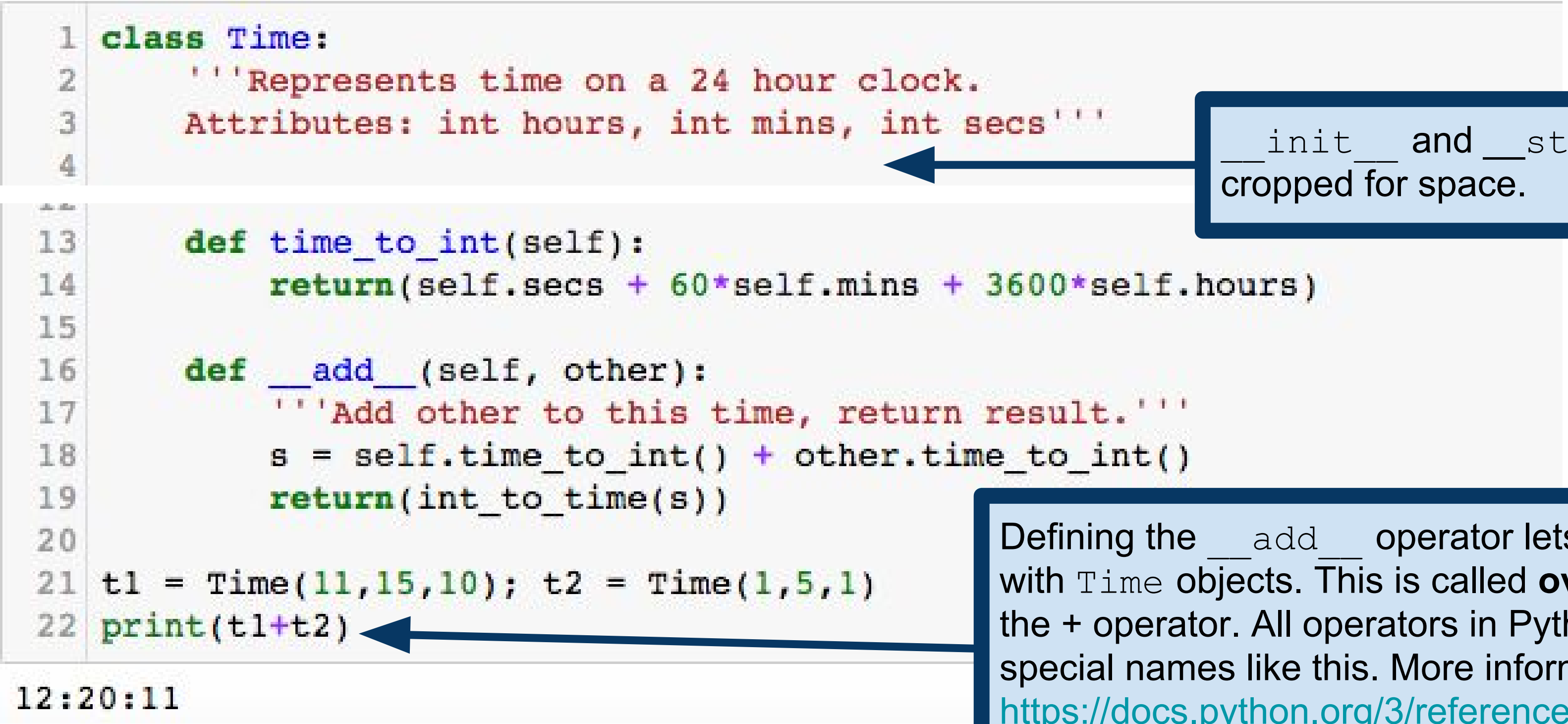
```
10:20:30
```

`__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



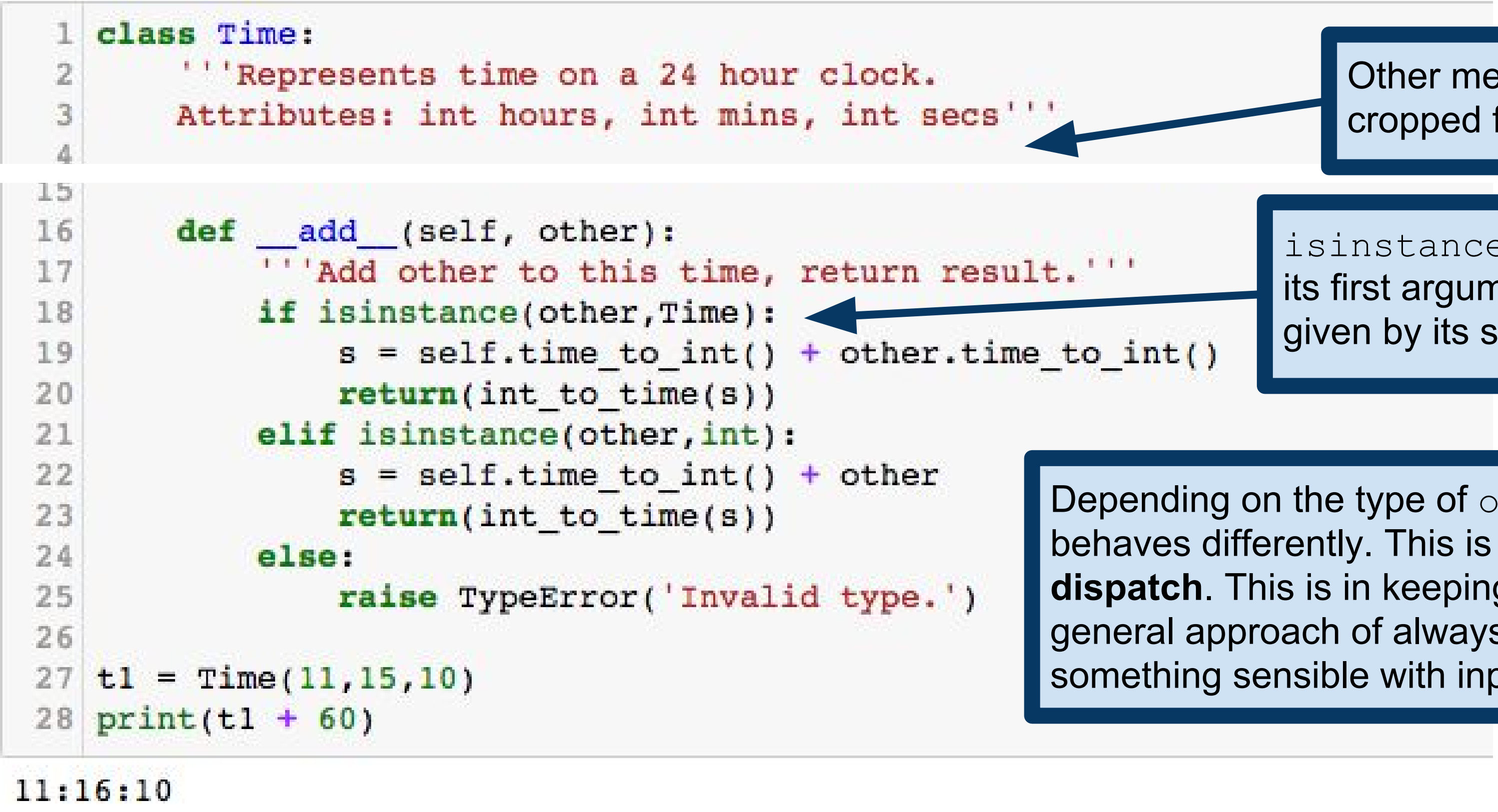
```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''


    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



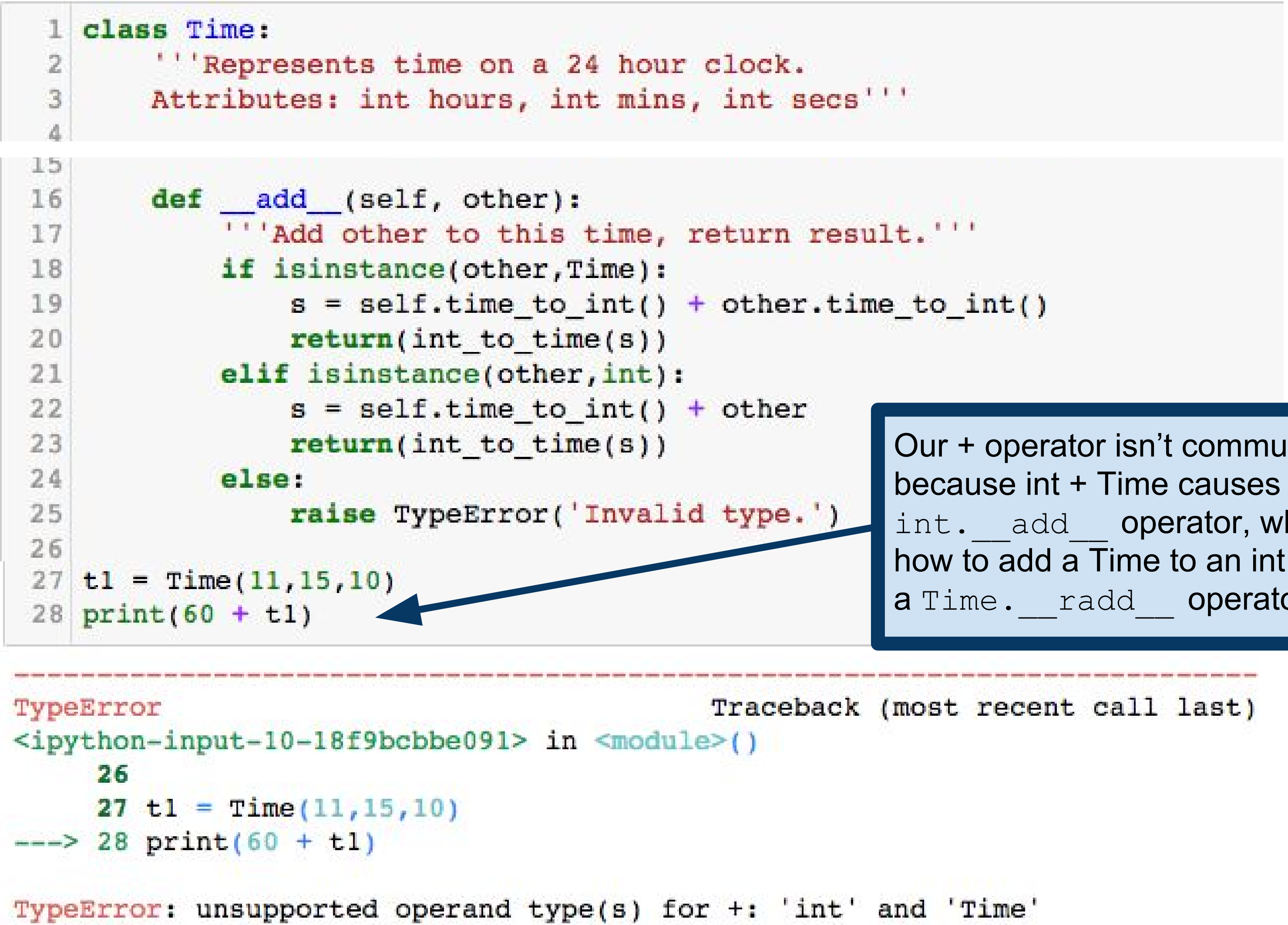
```
class Time:
    '''Represents time on a 24 hour clock.
    Attributes: int hours, int mins, int secs'''


    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.

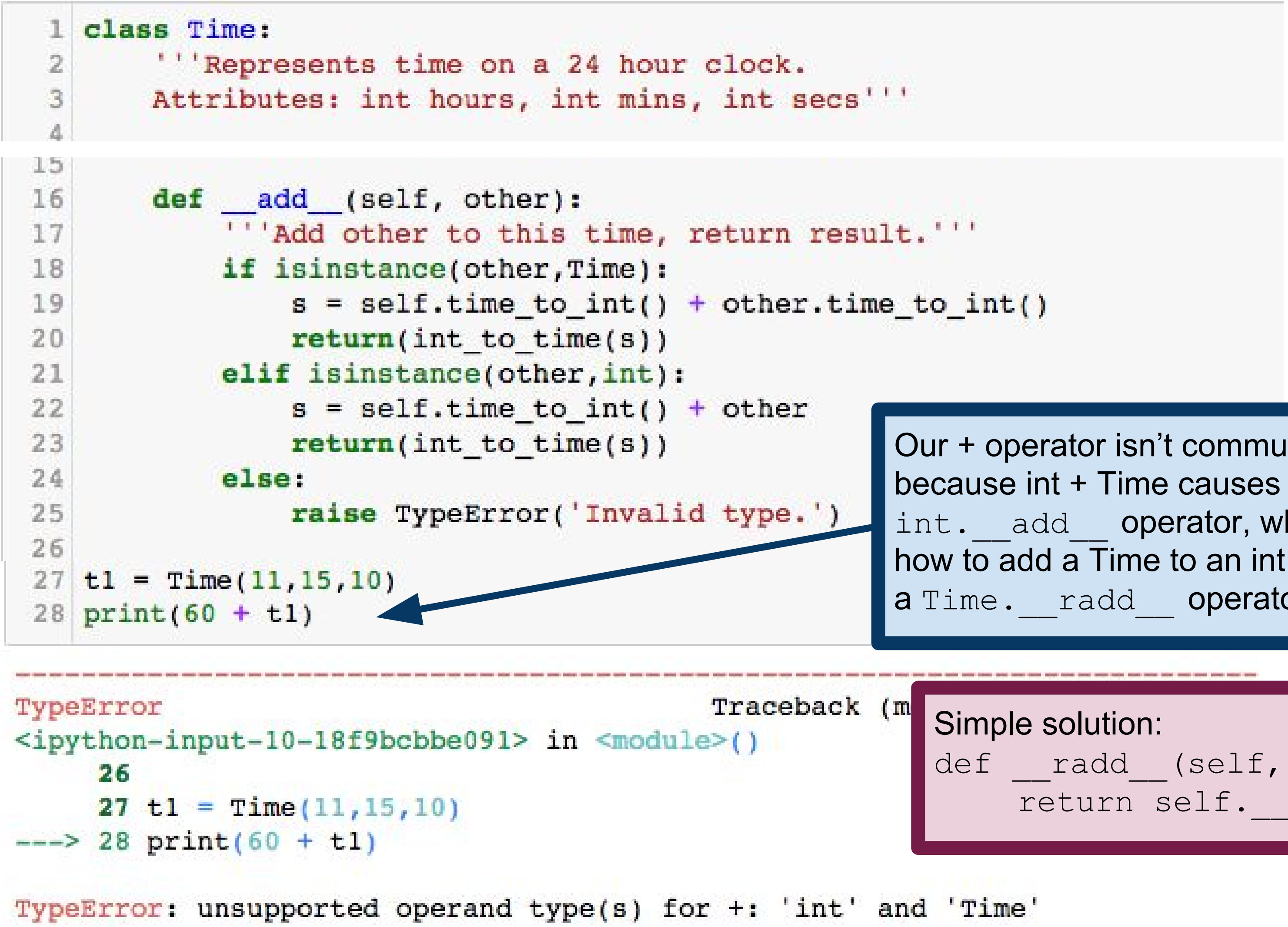
Depending on the type of `other`, our method behaves differently. This is called **type-based dispatch**. This is in keeping with Python's general approach of always trying to do something sensible with inputs.



```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.



```
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 (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'
```

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.

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

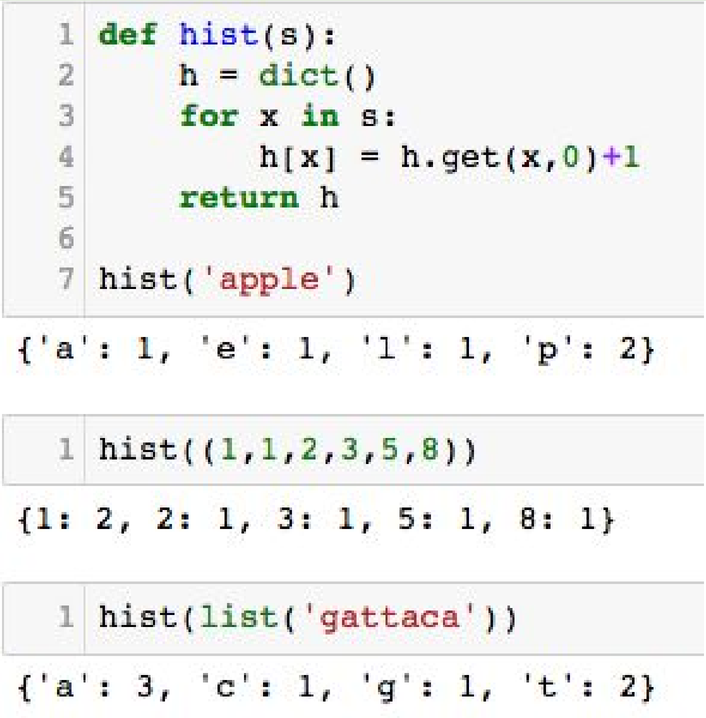
Examples:

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!



```
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

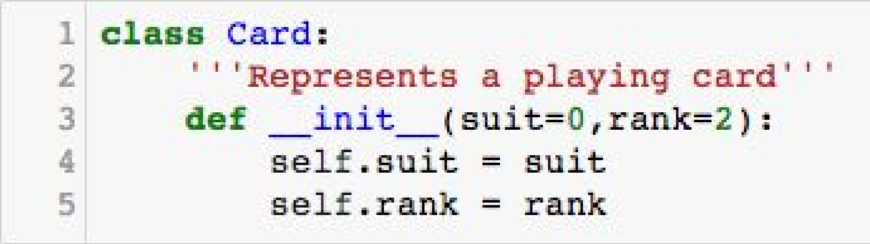
    Objects are playing cards, hands and decks

    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

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.



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

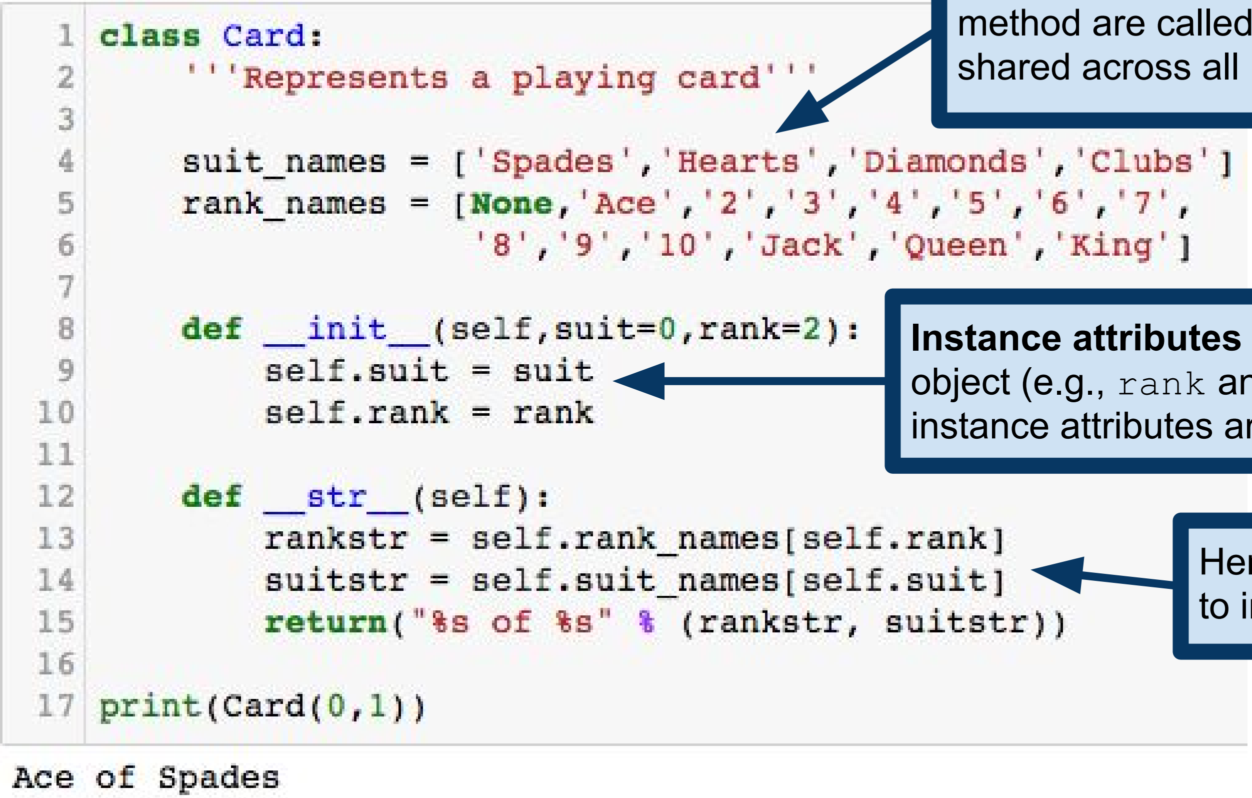
This stage of choosing how you will represent objects (and what objects to represent) is often the most important part of the coding process. It’s well worth your time to carefully plan and design your objects, how they will be represented and what methods they will support.

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



```
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))
```

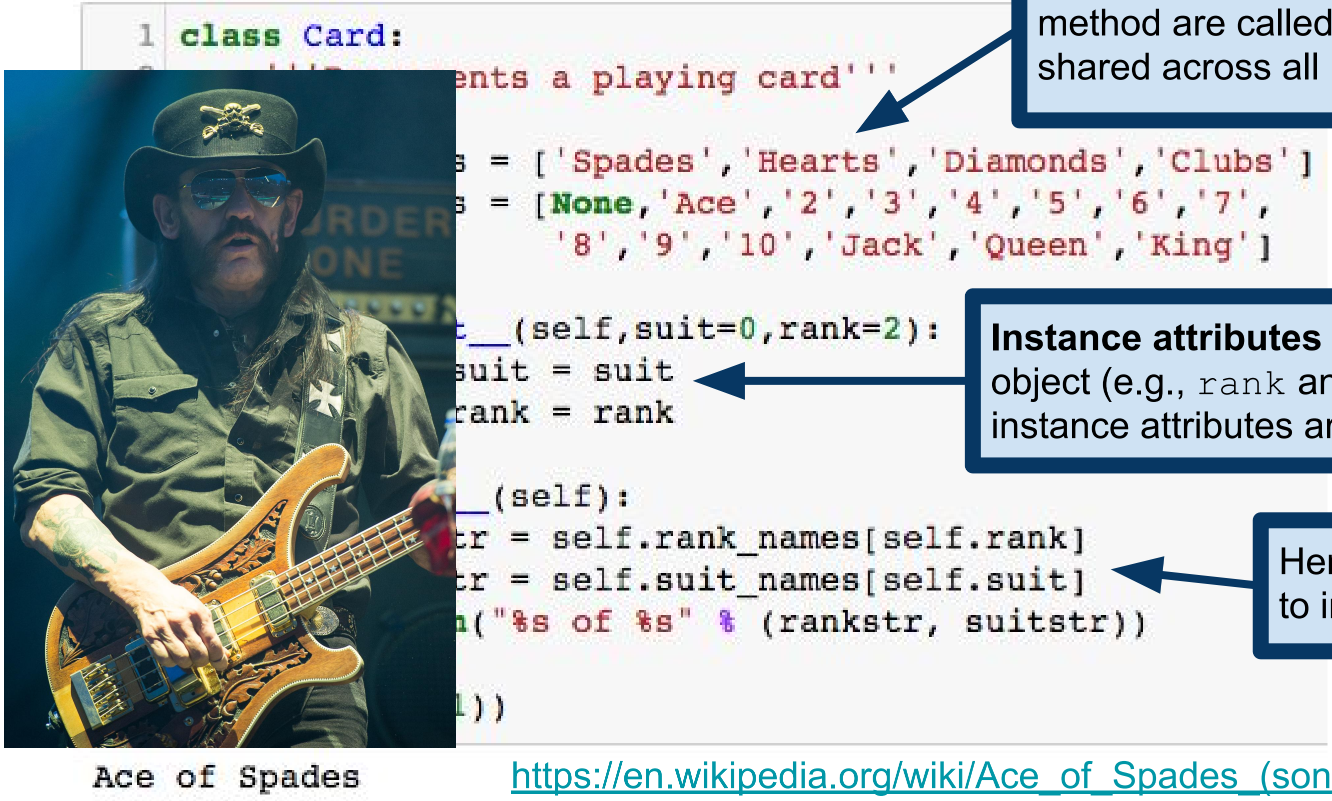
```
Ace of Spades
```

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:
                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
```

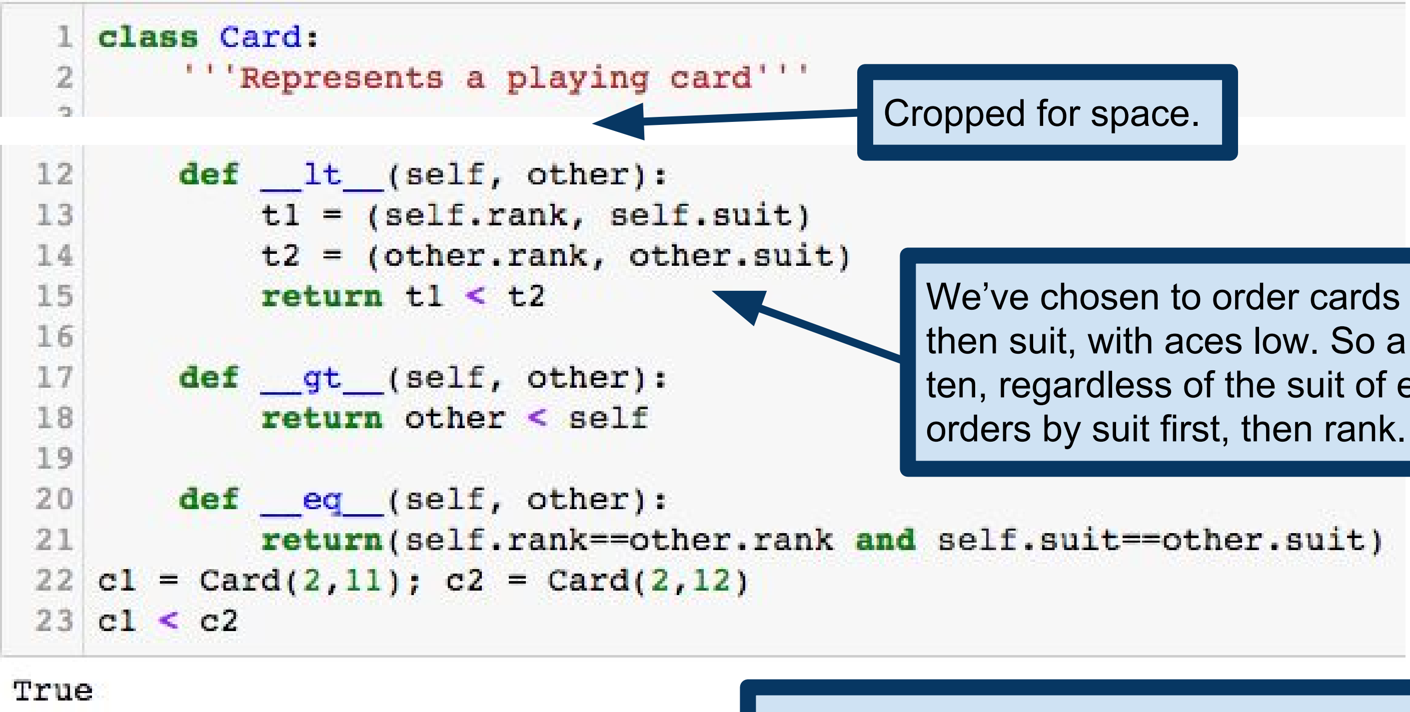
Variables defined in a class but outside any method are called **class attributes**. They are shared across all instances of the class.

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

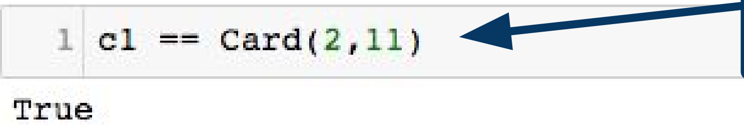


```python
class Card:
    '''Represents a playing card'''

    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.

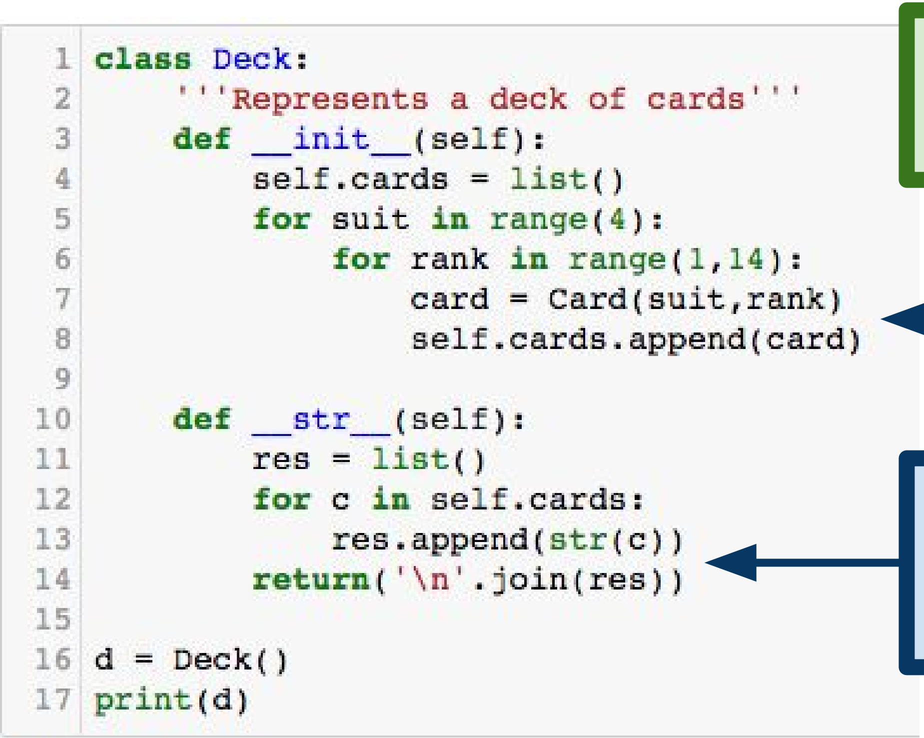


```python
c1 == Card(2,11)
```

```
True
```

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

Objects with other objects

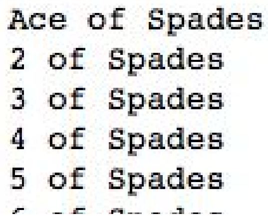


```
class Deck:
    '''Represents a deck of cards'''
    def __init__(self):
        self.cards = list()
        for suit in range(4):
            for rank in range(1,14):
                card = Card(suit,rank)
                self.cards.append(card)

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

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

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.

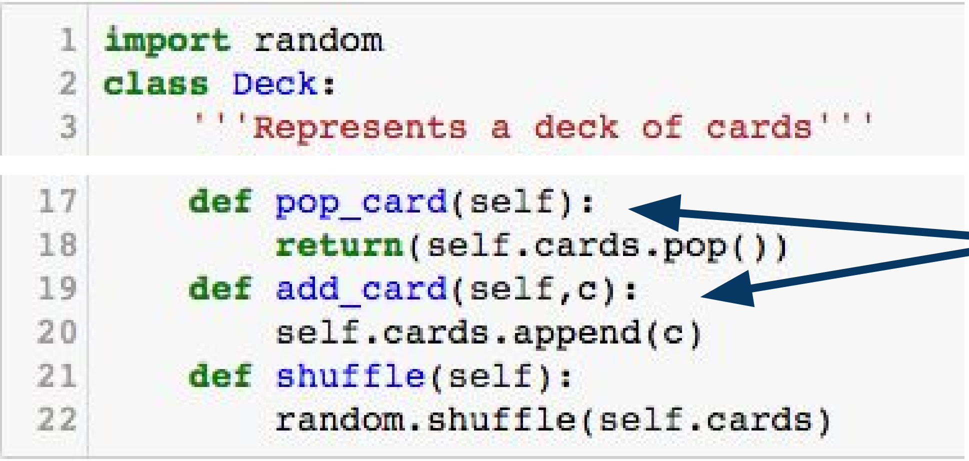
String representation of a deck will just be the cards in the deck, in order, one per line. Note that this produces a **single string**, but it includes newline characters.



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

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

Providing additional methods

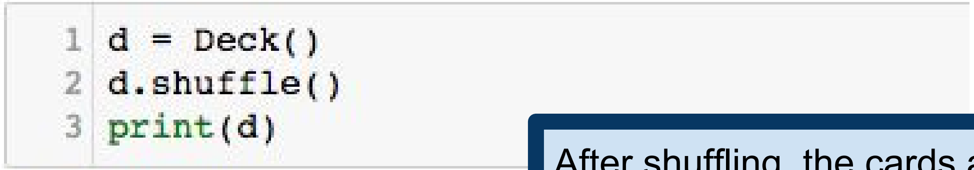


```python
import random
class Deck:
    '''Represents a deck of cards'''
```

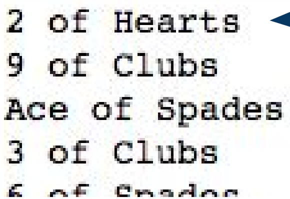
```python
    def pop_card(self):
        return(self.cards.pop())
    def add_card(self,c):
        self.cards.append(c)
    def shuffle(self):
        random.shuffle(self.cards)
```

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**.



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



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

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.

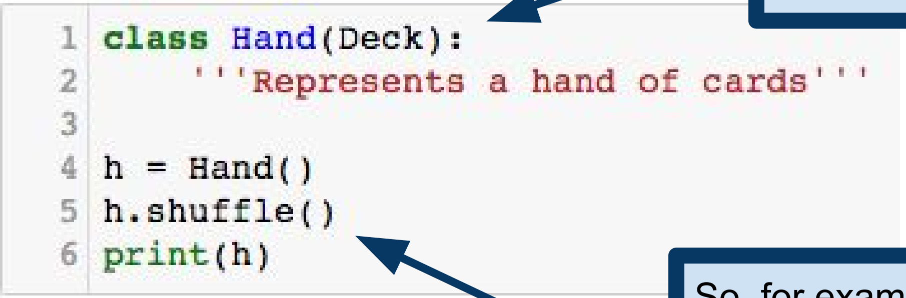
The naïve approach would be to create a new class Hand from scratch

- 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...

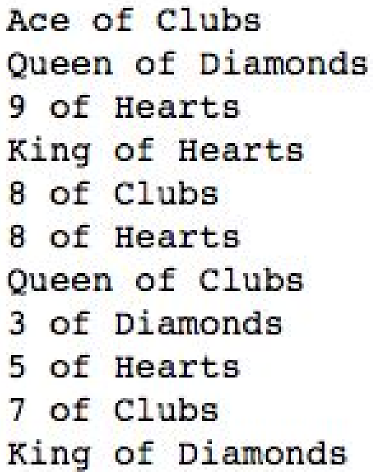
Inheritance



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

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

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`.

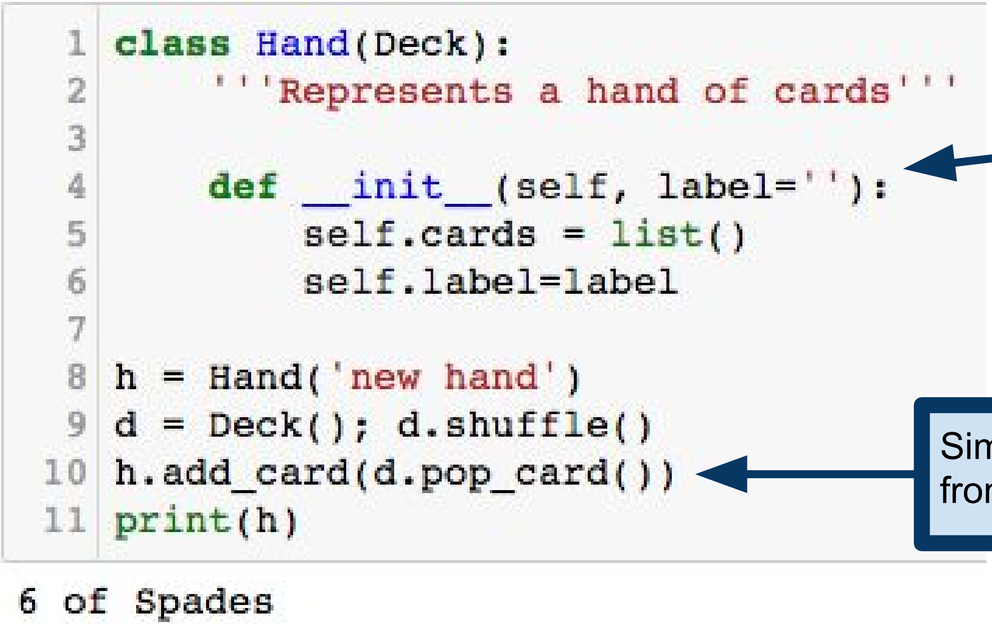


```
Ace of Clubs
Queen of Diamonds
9 of Hearts
King of Hearts
8 of Clubs
8 of Hearts
Queen of Clubs
3 of Diamonds
5 of Hearts
7 of Clubs
King of Diamonds
```

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



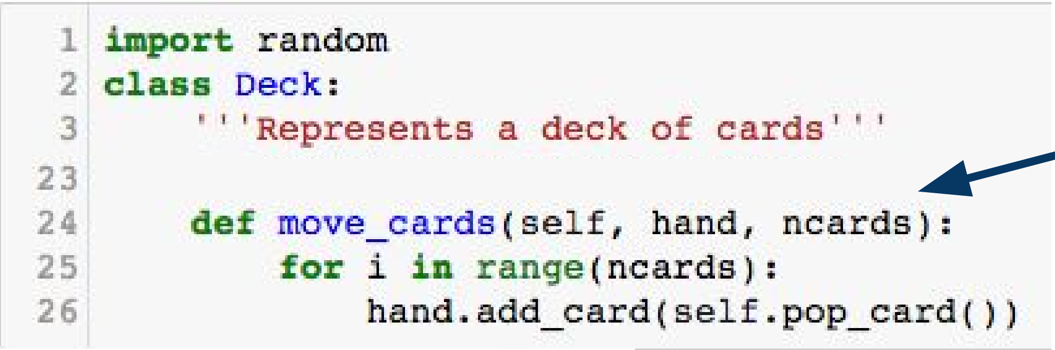
```
class Hand(Deck):
    '''Represents a hand of cards'''

    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`.

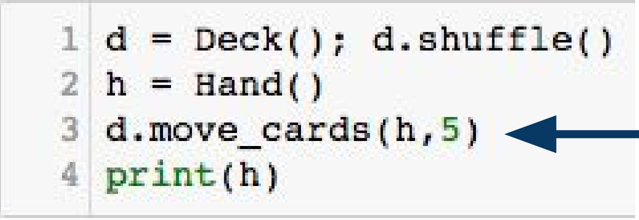
Simple way to deal a single card from the deck to the hand.

Inheritance: methods and overriding

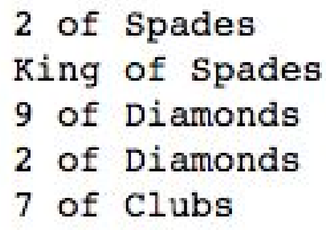


```
import random
class Deck:
    '''Represents a deck of cards'''

    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.



```
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)

Readings (this lecture)

Required:

Downey Chapters 17 and 18

Recommended:

Python documentation on operators

https://docs.python.org/3/reference/datamodel.html#specialnames

Coding style guides

https://google.github.io/styleguide/pyguide.html

https://www.python.org/dev/peps/pep-0008/

Readings (next lecture)

Required:

Python `itertools` documentation

https://docs.python.org/3/library/itertools.html

A. M. Kuchling. *Functional Programming HOWTO*

https://docs.python.org/3/howto/functional.html

Recommended:

M. R. Cook. *A Practical Introduction to Functional Programming*

https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming

D. Mertz. *Functional Programming in Python*.

http://www.oreilly.com/programming/free/functional-programming-python.csp