A scenario

- Suppose we have data from Twitter:
  - Tweets, users, who follows whom, likes, etc.
- Suppose we want to compute, stats e.g.,
  - Date with the most tweets
  - A particular user’s average tweet’s per day
- How will we store the data?
- Let’s start with storing a Tweet.
v0, with lists or dicts

• Each tweet is represented by a list, e.g.:
  ['David', 'Hello, I am so cool.', '2017-09-19', 0]

• Or by a dict, e.g.:
  {
    'userid': 'David',
    'content': 'Hello, I am so cool.',
    'created_at': '2017-09-19',
    'likes': 0
  }

• Pass these to functions that compute stats.

• Thoughts?
v1, with classes
Special method `__init__`

- `__init__` is a special method. We don’t have to call it explicitly.
- It is called an initializer.
- Two surprises:
  - It has one more parameter than argument.
  - It returns `None`.
- Make sense once you know:
  - Creating a new object involves *three* steps.
  - The initializer is just step 2.
Special method `__init__`

- When we create a new object, for example
  
  ```python
  t = Tweet(‘David’, date(2017, 9, 18), ‘hey’)
  ```

  *three* things happen:

1. The object is created.
2. `__init__` is called. The object’s id is passed to the first parameter.
3. The object’s id is returned.
Recap: v1, with classes

- A tweet is represented by an instance of a class.
  - Every instance has a userid, content, created_at, and likes
  - These are called instance attributes
  - The initializer initializes them
  - We use dot notation to access them

- We have added a new type to Python!
Best practices

1. Method `__init__` should initialize every instance attribute.

2. No other code should create attributes.
   - In fact, it is possible for other code to add attributes.
     Example:
     ```python
     t = Tweet('David', date(2017, 9, 18), 'hey')
     t.topping = 'bacon'
     ```
   - But this should never be done.

Effect?
Terminology

- Class
- Object, or instance of the class
- Instance attribute, instance variable, or data member
- Initializer
- Dot notation
Representation Invariants

• A **representation invariant** is
  – a boolean statement about the attributes of the class
  – which must be true for any instance of the class

• The types of the class attributes are simple representation invariants.

• But we can generalize this idea to express other kinds of things.
  – For example: ?
Methods and \texttt{self}

• A function defined in a class is called a \texttt{method}.

• The first parameter is passed implicitly. E.g.:
  \begin{verbatim}
  david = User(‘davidLiu’, ‘Lover of chocolate.’)
  david.verbosity(2016)
  \end{verbatim}

  Method \texttt{verbosity} receives
  \begin{itemize}
  \item The value of \texttt{david} into the first parameter
  \item The value \texttt{2016} into the second parameter
  \end{itemize}

• By convention, we always call that first parameter \texttt{self}. 
self and how we use objects

• We’re used to doing things like this:
  >>> word = 'supercalifragilisticexpealidocious'
  >>> word.count('i')
  6

• How does Python know which string to count?
  – The value of `word` is passed to the first parameter of `count`.

• These are equivalent:
  
  ```python
  word.count('i')
  str.count(word, 'i')
  ```

• (We can, but don’t, call `count` like that.)
Preconditions

• If arguments to the function must satisfy requirements (other than the type contract), say so.
• We call these requirements **preconditions**.
• Label them as such in the docstring.
Declare precondition vs check value?

• If we check that parameter values are sensible
  – The onus is on the method.

• If we declare a precondition
  – The onus is on the client code.
  – The method gets to be simpler and perhaps significantly faster.

• This is a design decision.

• Example where a precondition makes clear sense: binary search of a list.
Class User

• Notice that an instance of User contains references to 0 or more instances of Tweet.

• We say that:
  – Class User is in a contains relationship to class Tweet.
  – User has-a Tweet.
Choose method or function?

• Methods are for behaviours that
  – are essential to any client of the class
  – are integral to the very definition of that thing

• Functions are for behaviours that
  – are more specialized
  – not every client of the class would need
  – it feels reasonable to expect the client code to implement

• The choice may not be clear cut.
RIs are helpful with methods

• Representation invariants are very helpful now that we have methods.

• They help you write your methods:
  – You can assume the RIs are true when the method begins.
  – (They may become temporarily false during the method.)
  – You know you must restore the RIs by the end.
    This gives you a target that may suggest part of your algorithm.
All attributes are openly available

• The Python philosophy is to welcome client code to access the data unless there is good reason not to.

• Some good reasons not to welcome client code to access the data:
  – The structures storing the data are complex. Client code shouldn’t have to think about that.
  – There are important constraints on the data. We don’t want client code to mess it up!
When we want data to be private

• Indicate that intention with an underscore. Example: _cars

• This tells client code:
  – You shouldn’t access this attribute directly.
  – You should get the information by calling methods.

• The underscore does not prevent access; it only signals our intention.

• This approach defines an interface that:
  – Provides an abstraction for the client code.
  – Ensures the structures storing the data are not messed up.
Separating Interface from Implementation

• The distinction helps with two critical things.

• Management of complexity.
  – The class *encapsulates* all details about the implementation.
  – Client code can be written at a higher level.
  – Cognitive psychology: limited working memory; “chunking” helps.
  – This is essentially chunking.

• Plug-out / plug-in compatibility of code.
  – We can change the implementation of a class and client code runs the same.
Modelling a domain with classes

• When we tackle a new problem we must
  – Choose what the classes should be
  – Decide what each class will be responsible for

• In other words: we must decide how to model the domain.

• Simple rule of thumb:
  – An important noun suggests a class
  – A verb suggests a method
Class Design Recipe

• The Function Design Recipe taught us to
  – Separate interface from implementation.
  – Define the interface and test cases *before* implementing.

• The Class Design Recipe does also.
  – Homework: read it (It’s posted on the website.)
  – The process is not critical.
  – The produce *is* critical: an “API” for the class that does not reveal implementation details.