API Design in Python

Joe Zuntz
Mike Jarvis

Overview

1. Libraries and APIs
2. Essentials of API Design
3. Object Oriented APIs

Part 1: Libraries and APIs

Libraries: Concepts

- Libraries are re-usable collections of connected resources, in our case, mostly code
- Almost all DESC code should be in libraries
- **All should have a clear & documented API**

What should be in a library

- A library should have a limited and well-defined purpose
- If you find your library is doing lots of different things, consider splitting it up
- Science codes may not have clear boundaries
  - e.g. Should code to measure a statistic from data be in the same library as code to predict it from theory?
  - Dependencies can give you an indication

Types of Library

- In compiled languages like C or Fortran, libraries are collections of compiled code
    - Dynamic - programs using library find it each time they run
    - Static - library is delivered as part of programs that use it
- In interpreted languages like Python, libraries are collections of source code

python™ Libraries

- **Module =** single python file
    - Publicly available libraries are only occasionally distributed like this
- **Package =** collection of files under one directory (with subdirectories)
    - Much more common
    - Python treats a directory as a package if it contains a file __init__.py

Naming Libraries

- A good name is the most important thing for libraries.

- “RedMagic” is a great name - descriptive and cool.

- “TJPCosmo” was a bad name. “FireCrown” is a good name.

- Match the name not just to the content but to its social context

Python Packaging

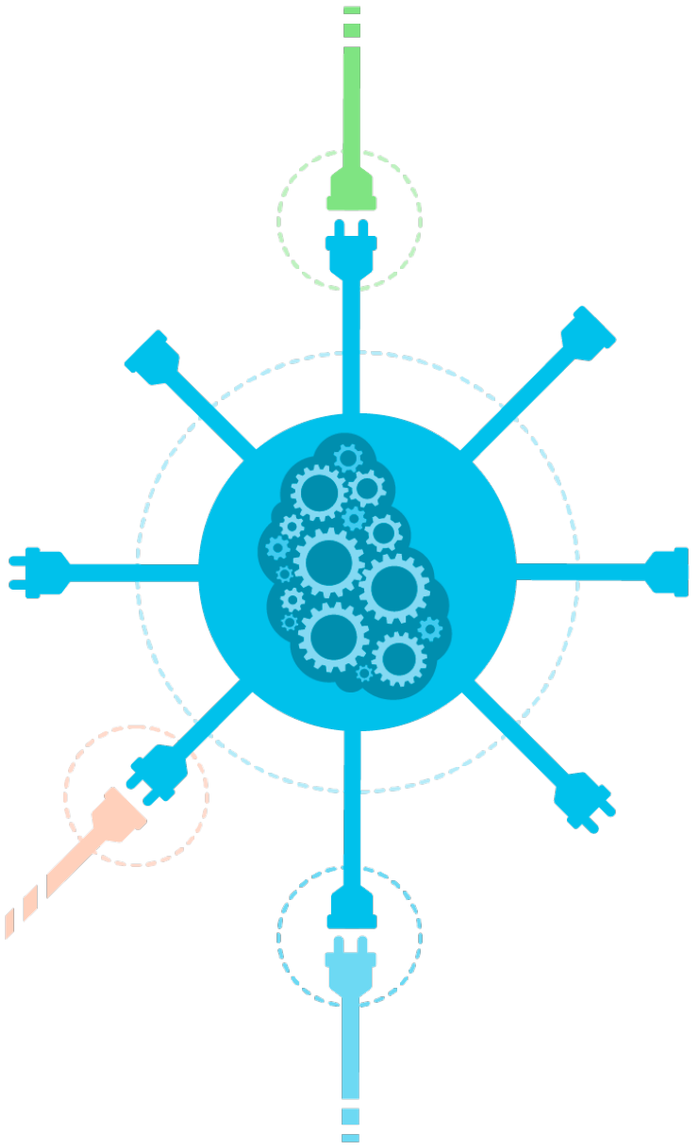
- Setting up your library to install correctly, e.g. with pip, and find dependencies

- Won’t discuss today, since it’s a huge topic

- However, Mike will give a mini-seminar on Friday about repo design, which will touch on this.

API: Overview

- API = Application Programming Interface
- The user interface to your library, where the “user” is other code
- Think of your code split into public and private sections
    - The private section is how things are done internally - implementation.
    - The public section is the functions, classes, or tools that are the entry point to the library - API



Examples of APIs

- Operating System provides APIs (e.g. POSIX)
- Graphics libraries (e.g. OpenGL)
- Python standard library
- Web API - Not discussing today, but dominates web searches for “API design”
- **User libraries like yours should have APIs**

Eating your own dog food

- If there's a main script along with your library, it should use your API in the same way as any other user

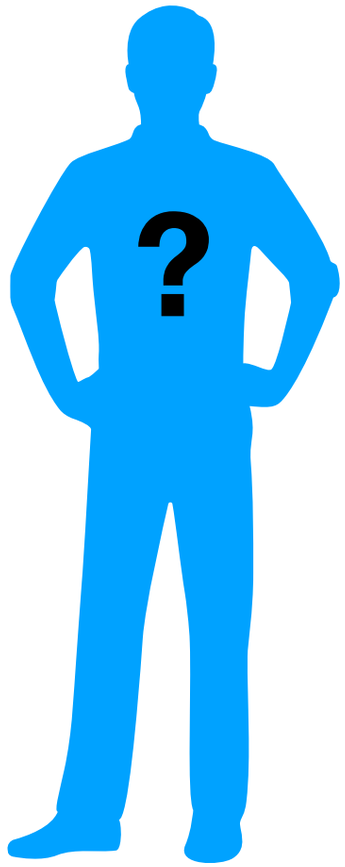
- Similarly, most of your unit tests should use the public API. Only rarely should you need to test private implementation details.

Today: APIs for python

- Function-based
- Object-oriented

First steps in API design

- Design your API before writing your code - the API defines what the code must be able to do under the hood



- **Step 1:** Gather detailed *use cases* for the library - specific things that people will want to do with it

- Ideally do this in consultation with actual users

User Stories

- It can be useful to think in terms of *user stories*:

  “A weak lensing analyst needs to access the shear estimator and response for all galaxies in a specific tomographic bin."

  “A SAWG investigator runs the code repeatedly to check the impact of an image anomaly”

  “A modified gravity theorist is changing gravity and seeing the effects on the theory spectra”

Activity 1

- Form small groups of 4-5

- We'll focus on applications for interacting with a **catalog of transient objects**.

- Make a list of as many use cases and corresponding user stories as you can think of.

Activity 1

- Discussion
  - Search the catalog for lensed quasars.
  - Query catalog for externally known transient objects (e.g. kilonova)
  - Find supernova, get spectra, look for weird stuff
  - Estimate SED model from light curves
  - Find host galaxies in DB corresponding to transients
  - Get all images that correspond to transient detections
  - Get information about the environment near transient
  - Look for microlensing objects (from template maybe)
  - Find non-transient correlate of transient detection (if any)
  - Get orbits of asteroid to look for modified gravity implications of orbits
  - Add a new item to the catalog
  - Access what are all the column names available in the catalog
  - Get documentation about how each column was built

Activity 1

- Some that we thought of:
  - Update the catalog as new objects are detected
  - Export (a subset of) the catalog to file
  - Get ra, dec, time, mag for all z-band-only transients
  - Find isolated transients that are within 1 arcmin of another isolated transient
  - Get light curves for transients with at least 5 observations in r
  - Find transients that match a Gaia star
  - Find transients that are within 1 arcsec of an $i < 21$ galaxy
  - Get r-band light curves for transients that match SN Ia light curve profile
  - Mark some sets of transients as each being the same SN
  - Mark some transients as being eclipsing binary stars
  - Mark some transients as unclassified cosmic rays

Part 2

Starting API Design

Design Approach

- Often easier to write code that will use the library first, to illustrate what the interface should look like
- Take a user story and think about what steps the user would want to take - try to be specific:
  - What parameters will the user need to specify?
  - What options are important?
  - What kind of output value would be expected?
  - Does this use case seem to conflict with the requirements of some other use case? How might this be reconciled?

Design Approach

- Be ambitious in the high-level API design.
- Try to think of everything some user might want to do, even if you won't implement all (or even most) of it yet.
- E.g. if it is conceivable that you might some day add some feature, don't design it out of your API.
- Usually this means making things as generic as possible.
- This can sometimes be in tension with computational efficiency. Making appropriate compromises regarding flexibility vs efficiency is tricky.

Science-Specific API Considerations

- Generally want much more customisation, so generally want to
- Implementation must be clear to users for numerical code
- Tend to push code to unreasonable limits
- Less focus on stability
- Less user/developer distinction

Function APIs

- Collections of functions that work together, e.g.

```
prepare_calculation()
run_calculation()
save_calculation()
load_calculation()
```

- There should be a clear order that users should run these in, and a clear reason why they are separated

- If a function is always run after another with no changes in between, they should be one call

Function Questions

- Will users want to do something between two parts of a calculation?
    - Modify data, change options, etc.
    - If so, consider splitting into two functions
- Will users want to replace some part of a workflow with their own function?
    - If so, make sure this is possible

Life Cycles

- A subset of your API functions are often life-cycle functions

- Creation
    - From scratch
    - From file
    - From configuration options

- Destruction
    - Write data to file
    - Close open resources

Naming

- For functions and methods, prefer verbs & questions to nouns & adjectives
    - `open` is better than `file`
    - `get_resultant` is better than `resultant`
    - `is_prime` is better than `prime`
- Avoid cryptic names - be verbose if necessary:
    - In python standard library you'll find `NamedTemporaryFile`, not `nmdtmpfile`
    - Code is read far more often than it is written. Therefore, choose clarity over brevity.

Namespace

- C & Fortran functions usually have a single *namespace*
  - Can’t have two publicly accessible functions with same name
- Convention is to use common prefix for all lib functions
  - e.g. the CFITSIO library defines `fits_open_file`, `fits_create_file`, `fits_get_img_size`, …
- **Not** needed in python, because:
  - ```
    from astropy.io import fits
    fits.open  # no clash with built-in open
    ```
    ```

Python Keywords

```
def function(arg1, arg2, new_keyword=False)
```

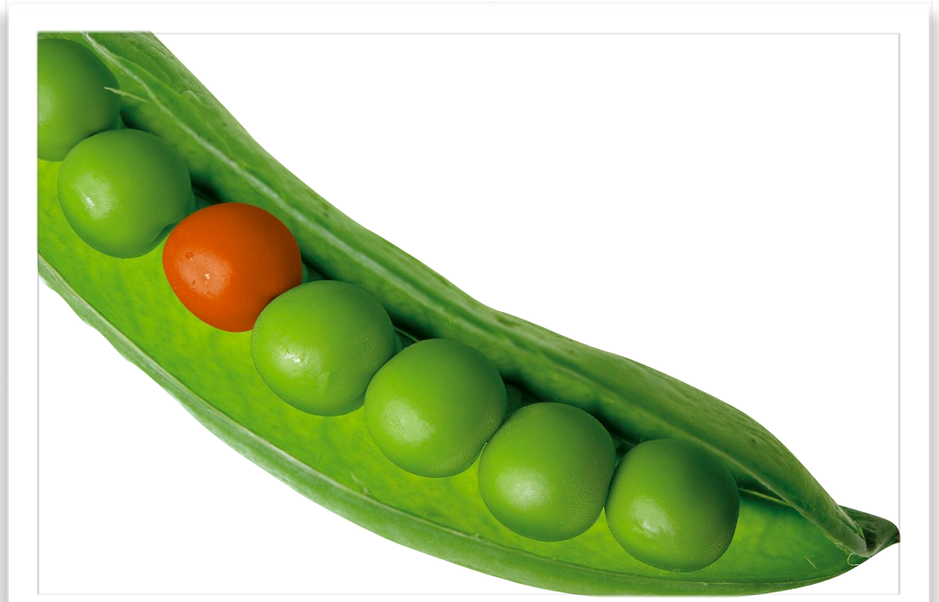
- Allow for a default value to be used in most cases, but allow the user to change it if they need to.

- Allows for potentially lots of customization about how something works without a cumbersome interface for most users who just use the defaults.

- Can be used to extend an existing API without breaking old code by adding a new keyword, but leaving the default as the old behaviour.

Consistency

- Consistency is more important than perfection for APIs in large projects



- If you already have functions called `powerOfTwo()` and `powerOfTen()` don't add `is_power_of_n()`

- If all your classes have methods called `save`, don't add a new one with `write`

Stability

- Changing existing API will break any code using on the old one. VERY BAD!
- Minimize how often this happens!
    - Ideally, only at major version numbers**
      e.g. 1.7.3 -> 2.0.0
- Before a major version, should "deprecate" old API
    - should still work, but give a warning and instructions about converting to new API.
- Adding new features (extending the API) is fine, since it doesn't break old code.

** This doesn't apply prior to version 1.0. 0.x are considered development versions, and stability is less important.

Documentation

- Documentation of the API is for users
    - This should be included in public documentation e.g. at readthedocs.io

- Documentation of other code is for developers
    - If generated, this should be clearly marked as not being part of the public API

Activity 2

- Pick one of the stories we came up with in Activity 1. (Does not have to be one from your group.)
- What function call(s) would make sense for this action?
- What parameters should the function(s) take?
- What are good defaults for some/any of the parameters?
- What parameters should be required?
- What will be returned (if anything)?
- Will the catalog be modified in any way? What are the implications of this that might need some care?

Activity 2

- Discussion

Part 3: Object Oriented APIs

Part 3: Object Oriented APIs

- Main interface is in the form of one or more classes.
- Class encapsulates a set of values that all work together as a single concept.
- Users create instances of these, and then call methods to perform various actions with the data.
- E.g. `image = galaxy.drawImage(scale=0.2)`
- Objects hold a "state" that may be different for different instances of the same class, depending on construction arguments or what methods have been called previously.

When To Create Classes

- There is a trade-off in choosing to upgrade functionality to a class.
- Too many classes makes things awkward (developer must learn class API and code must accept it)
- Do you find that a long list of arguments is always passed around together?
- Are you passing around a dictionary and manipulating its contents?
- Are you writing lots of functions with the same first argument?

Class names

- OO APIs are popular in science code, because they allow for an obvious connection between real physical objects, concepts, processes, etc. and data types in the code.
- Just as functions should almost always be verbs, classes should almost always be nouns.
  - `Galaxy`
  - `TransientObject`
  - `Redshift`
  - `Tracer`
  - `PhotonArray`

Method names

- Methods are a kind of function, so they should often be verbs. These are the actions that the objects know how to do.
  - `image.addNoise(noise_generator)`
  - `psf.draw(image)`
  - `cosmology.calculate_sigma8()`
  - `data.write(output_file)`
- Some methods are clearer as nouns or phrases, so use your best judgement about what makes sense
  - `coord.angleTo(other_coord)`
  - `wcs.jacobian(image_pos)`
  - `galaxy.atRedshift(new_redshift)`

Method names

- Method names should generally give a clue about how complicated a calculation is required.
  - `gal.shape` should be trivial -- usually accessing a value already stored in memory.
  - `gal.getShape()` should be either very fast or amortized fast (i.e. result is stored, so second and later calls are fast).
  - `gal.calculateShape()` or `gal.measureShape()` may require some significant calculation.
  - `gal.queryShape()` or `gal.retrieveShape()` might involve some potentially slow I/O or DB operation.

Arithmetic

- Python has special methods for defining how arithmetic should work on objects.
- e.g. `__add__(self, x)` defines what `obj + x` does.
- For many scientific objects, arithmetic is intuitive:
  - `image *= 2.`
  - `total_shear = shear1 + shear2.`
- Sometimes using arithmetic leads to a nicer design:
  - `angle = 90 * coord.degrees`
- Don't surprise your user: `a=a*b` and `a*=b` should be equivalent, no matter what `a` and `b` are.

Capitalization

- Class names are traditionally capitalized.
- CamelCase is common when two or more words are needed.
- Instances of the class (objects) are traditionally lower case. e.g. `star = Star()`
- Method names are traditionally either camelCase or with_underscores_between_words. Be consistent.

Private methods

- Often it is helpful to break up parts of the implementation into several helper functions/methods.
- These are generally not functions you want/expect the user to call directly.
- To indicate that, start these names with a single underscore.
- This doesn’t actually stop users calling these methods. It just indicates that they probably shouldn’t.
- They won’t appear when you press TAB in iPython or Jupyter.

Immutable Objects

- In Python, methods are allowed to change the internal state of an object.
- Functional programming languages like Haskell don't allow that. All changes are returned as new objects.
- Compare:
  - `galaxy.set_redshift(1.5)`
  - `high_z_galaxy = galaxy.at_redshift(1.5)`

Immutable Objects

- Advantages:
  - Easier to reason about your (both developer and user) code.
  - Always safe to pass objects as function parameters, and know that they won't be changed.
  - Usually leads to fewer bugs in code.
  - Efficient to keep as members of other objects, since you don't need to make a "defensive copy" to ensure your version doesn't get changed by some other code.
  - Easier to parallelize, since there are no race conditions.

Immutable Objects

- Disadvantages:
  - Can be inefficient to return new copies of objects with large memory footprints.
    - e.g. We could generate a new transient catalog each time we remove objects, but this would likely be very inefficient.
- Some objects are inherently stateful (e.g. random number generators) so cannot be treated as immutable.

Immutable Objects

- It is generally helpful to design classes to be functionally immutable whenever possible.
- In Python, you cannot really force objects to be truly immutable, but you can design the public API such that no public methods ever change the object's state.
- `@property` can be used to make attributes functionally immutable.
- E.g. in GalSim, the only classes with mutating methods are `Image`, `PhotonArray`, and various random number generators.

Factory Functions

- Factory functions are function that return a new instance of a class.
- Useful when there are several possible ways you might make a new instance.
- Keeps the regular class initialization method simpler.
- In Python, it is a good idea to make these class methods.
  - `coord = CelestialCoord.from_galactic(el,b)`
  - `psf = PSF.read(piff_file)`

Subclasses

- Use subclasses when there are several similar concepts that share some common functionality.
- Subclasses have an "is a" relationship to the base class.
    - `SpiralGalaxy` is a `Galaxy`.
- Functionality common to all classes can be defined in the base class.
- Subclasses may add more methods, and may define private methods used by the base class implementations.
- Base classes that are not constructible are called "abstract data types".

Subclasses

- Example - the GalSim WCS functionality:
  - Lots of specific WCS classes with different back end implementations and different levels of complexity.
  - **`PixelScale`**, **`AffineTranform`**, **`AstropyWCS`**, **`PyAstWCS`**, etc.
  - Most of the public API is defined in **`BaseWCS`**.
  - **`EuclideanWCS`** and **`CelestialWCS`** are mid-level classes for which some of the API is qualitatively different.
  - Subclasses define private methods to implement the functionality used by the higher level classes.

Activity 3

- Get in your groups again.
- What are some classes we would want to define to facilitate the functionality discussed earlier?
- Should some of the functions we discussed be methods of one of these classes?
- What other methods would be helpful?
- Would some be better as free functions rather than methods?
- Would any subclasses be helpful?

Activity 3

- Discussion