Project Structure

Peter Hill

Outline

- Higher-level things
- Low-level things

Projects

- Individual piece of software
- Collected set of components, e.g.:
  - Simulation
  - Post-processing
  - Data sets
  - Website

The central ideas

- Abstract over related things
- Give things names
- Group things in namespaces
- Keep it simple

Software directory layout

A typical C++ project layout

```
my_code
|-- docs
|-- examples
|-- include
|   \-- my_code_public.hxx
|-- src
|   |-- my_code_private.hxx
|   \-- my_code.cxx
|-- tests
|   \-- test_my_code.cxx
|-- LICENCE
\-- README.md
```

Software directory layout

A typical Python project layout

```
my_code
|-- docs
|-- examples
|-- my_code
|   |-- __init__.py
|   \-- my_code.py
|-- tests
|   \-- test_my_code.py
|-- LICENCE
|-- README.md
\-- setup.py
```

Software directory layout

Essential

- README
  - What, why and how

Very good to have

- Separate source directory (“src”)
- Separate documentation directory
- Examples for libraries
- Licence! What am I allowed to do?

Up to you

- Separate tests directory
- Examples directory

The README

The most important file

- Often the first file people see
- Make a good impression!
- Details **what** the code is *for*
- Details **how** to get started
  - How to get access
  - Where to download from
  - How to install (including dependencies!)
  - How to run tests/examples
- Where to get more information
  - FAQ, papers, forums, etc.

The README

Example

```
# PlanetDetector

PlanetDetector detects planets in images.

Installation:
    pip install --user planetdetector

Run it like:
    planetdetector --mars image03.jpg

## Requirements
PlanetDetector needs `libplanet > 2.3`
```

The directories

- Separate out "project admin" from source
  - True for both Python and C++/Fortran
- Might be divided into subcomponents
- Tests might be alongside source or separate
- `include` directory for public API
- Documentation might be separate files or inline

Handling dependencies

Python

- `requirements.txt`
- List required/optional dependencies and versions

Compiled languages

- Some “standard” dependencies, e.g. FFTW, LAPACK
  - Provide instructions for installing
- For non-standard dependencies, put under `externals/`
- Can commit files directly
- Or include as git submodules
  - Easier updating

Source code

- Data structures
- Functions
- Files/Modules
- Sub-components

Data types

"Bad programmers worry about the code. Good programmers worry about data structures and their relationships." - Linus Torvalds

- Choosing the right data structure can make all the difference
- Can be easier to reason about data structure than logic

Data types

Not great

```
DiffLookup lookupFunc(DiffLookup *table, string label) {
  for (int i = 0; DiffNameTable[i].method != DIFF_DEFAULT; ++i) {
    if (strcasecmp(label.c_str(), DiffNameTable[i].label) == 0) {
      auto method = DiffNameTable[i].method;
      if (isImplemented(table, method)) {
        for (int j=0;;++j){
          if (table[j].method == method){
            return table[j];
          }
        }
      }
    }
  }
  ...
```

Data types

Better

```
DiffLookup lookupFunc(map<string, DiffLookup> table, string label) {
  return table[label];
}
```

Data types

Object oriented programming

- Wrap up several concepts into a higher-level abstraction
- Bundle together related nouns (data) and verbs (functions)
- Abstract a `Particle`, wrapping up mass, charge, position, etc., and how to calculate energy, force, etc.
- Reduces cognitive load, freeing up mental energy to think about more important things

Object oriented programming

Before

```
energy = calculate_kinetic_energy(mass1, charge1, position1,
                                  velocity1, E_field)
force = coulomb_force(charge1, charge2, position1, position2)
update_position(position1, mass1, charge1, velocity1, force)
```

Object oriented programming

Before

```
energy = particle1.kinetic_energy(E_field)
particle1.set_coulomb_force(particle2)
particle1.push()
```

Data types

Object oriented programming

- Object-oriented programming is a way to wrap up data and functions that operate on that data
- Can be a good mental fit for lots of problems in physics
- Four “pillars”:
  - Abstraction
  - Encapsulation
  - Inheritance
  - Polymorphism

Functions

- Reusable tasks
- Gives names to things
  - Names are amazing!
- Single responsibility principle
  - Each “thing” should have one task
- If it's hard to name, is it really two functions?

Functions

Example

```
for i in range(len(array)-1, 0, -1):
    for j in range(i):
        if list[j] > list[j+1]:
            temp = array[j]
            array[j] = array[j+1]
            array[j+1] = temp
```

Functions

. . . becomes

```
result = sort(array)
```

Naming things

- Names are hard!
- Trade off between short and descriptive
- Variables are nouns, functions are (usually) verbs
- Dnt ndlssly abbrev
  - Absolutely no single-letter variables!
- `bool very_long_variable_names_can_be_difficult_to_read = true`
- Naming conventions help distinguish between types of names, e.g.:
  - `PascalCase` for classes/types
  - `camelCase`/`snake_case` for functions/variables

Runtime checks and multiple conditions

Arrow code

```
def calculate_thing(x, y, limit, dry_run):
    if x < limit:
        if y >= 0:
            if not dry_run:
                # do thing
            else:
                return True
        else:
            raise ValueError("negative y")
    else:
        raise ValueError("x over limit")
```

Runtime checks and multiple conditions

Prefer preconditions

```
def calculate_thing(x, y, limit, dry_run):
    if x > limit:
        raise ValueError("x over limit")
    if y < 0:
        raise ValueError("negative y")
    if dry_run:
        return True

    # do thing
```

Splitting up files

- Single files get unmanageable above 10k lines
- Group logically related things together
- Sub-components might even go in separate directories
- Namespace: set of symbols to organise objects
  - filesystem! `/home/peter/documents` and `/home/nicky/documents`
  - Python modules
  - C++ `std::`

Fortran Modules

- Always use modules!
- Compiler generates interfaces for procedures in modules
  - Doesn't do this for bare `subroutines` in files
- Interfaces catch bugs!
- Can control access to "internals"

Fortran Modules

integrator.f90

```
module integrator
  private               ! Everything private by default
  public :: integrate   ! Make integrate public
contains
  real function integrate(array, spacing)
    real, dimension(:), intent(in) :: array
    real, intent(in) :: spacing
    ...
  end function

  subroutine helper(...)
  ...
  end subroutine
end module
```

Fortran Modules

example.f90

```
real function total_mass(volume, density)
  use integrator
  ...
  total_mass = integrate(density, dv)
end function
```

No namespaces

- Fortran modules are not namespaced

Fortran Modules

even_better_example.f90

```
real function total_mass(volume, density)
  use integrator, only : integrate
  ...
  total_mass = integrate(density, dv)
end function
```

No namespaces

- Fortran modules are not namespaced

Python Modules

- Python modules can be single files or whole directories
- Make a directory into a module with a `__init__.py` file
- `__init__.py` defines what is imported by default
- Modules are namespaces:
  - `mycode.min` and `numpy.min` can co-exist

Example.py

mycode/

```
mycode/
 |-- __init__.py
 \-- integrator.py
```

mycode/integrator.py

```
def integrate():
```

mycode/__init__.py

```
from integrator import integrate
```

Using mycode

```
import mycode
mycode.integrate(data)
```

C++

Namespaces

- No modules in C++ (yet!)
- `#include`-ing files literally inserts the text of the file
- Namespaces allow reusing the same name for different things

```
#include <vector>
namespace mycode {
  class vector { ... };
}
std::vector<double> standard;
mycode::vector mine;
```

Anonymous namespaces

- Reduces the scope of members to the translation unit
  - Translation unit: a source file and all its included headers

Documentation

- Writing documentation is bad
- Undocumented code is worse
- Try to make code self-explanatory
  - Always in-sync!
- Then write documentation directly in source
  - => Reference manual
- Then write stand alone documentation
  - => How-to guide or tutorial

Inline documentation

Documentation builders

- Built in to some languages like Python:

```
def foo(a, b):
    """
    Foos a and b together, returning a list of the results
    """
```

- Tools exist for other languages, e.g. Doxygen, Ford

```
//> Foos a and b together, returning a list of the results
std::list<result> foo(int a, int b) {
```

```
!! Foos a and b together, returning a list of the results
function foo(a, b) result(list)
```

Inline documentation

Documentation builders

- Compiles inline documentation into e.g. LaTeX, PDF, HTML
  - HTML could go on project website
  - Most allow LaTeX
- Web services exist for doing this automatically
  - e.g. Readthedocs https://readthedocs.org/

Conclusion

- Abstract over related things
- Give things names
- Group things in namespaces
- Keep it simple