docs.python-guide.org

Structuring Your Project — The Hitchhiker's Guide to Python

25-32 minutes

By “structure” we mean the decisions you make concerning how your project best meets its objective. We need to consider how to best leverage Python’s features to create clean, effective code. In practical terms, “structure” means making clean code whose logic and dependencies are clear as well as how the files and folders are organized in the filesystem.

Which functions should go into which modules? How does data flow through the project? What features and functions can be grouped together and isolated? By answering questions like these you can begin to plan, in a broad sense, what your finished product will look like.

In this section we take a closer look at Python's module and import systems as they are the central elements to enforcing structure in your project. We then discuss various perspectives on how to build code which can be extended and tested reliably.

Structure of the Repository¶

It's Important.¶

Just as Code Style, API Design, and Automation are essential for a healthy development cycle, Repository structure is a crucial part of your project's architecture.

When a potential user or contributor lands on your repository's page, they see a few things:

- Project Name
- Project Description
- Bunch O' Files

Only when they scroll below the fold will the user see your project's README.

If your repo is a massive dump of files or a nested mess of directories, they might look elsewhere before even reading your beautiful documentation.

> Dress for the job you want, not the job you have.

Of course, first impressions aren't everything. You and your colleagues will spend countless hours working with this repository, eventually becoming intimately familiar with every nook and cranny. The layout of it is important.

Sample Repository¶

tl;dr: This is what *Kenneth Reitz recommended in 2013<https://www.kennethreitz.org/essays/repository-structure-and-*

python>.

This repository is available on GitHub.

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

Let’s get into some specifics.

The Actual Module¶

Location	`./sample/` or `./sample.py`
Purpose	The code of interest

Your module package is the core focus of the repository. It should not be tucked away:

If your module consists of only a single file, you can place it directly in the root of your repository:

Your library does not belong in an ambiguous src or python subdirectory.

License¶

Location	`./LICENSE`
Purpose	Lawyering up.

This is arguably the most important part of your repository, aside from the source code itself. The full license text and copyright claims should exist in this file.

If you aren’t sure which license you should use for your project, check out choosealicense.com.

Of course, you are also free to publish code without a license, but this would prevent many people from potentially using your code.

Setup.py¶

Location	`./setup.py`
Purpose	Package and distribution management.

If your module package is at the root of your repository, this should obviously be at the root as well.

Requirements File¶

Location	`./requirements.txt`
Purpose	Development dependencies.

A pip requirements file should be placed at the root of the repository. It should specify the dependencies required to contribute to the project: testing, building, and generating documentation.

If your project has no development dependencies, or you prefer development environment setup via `setup.py`, this file may be unnecessary.

Documentation¶

Location	`./docs/`
Purpose	Package reference documentation.

There is little reason for this to exist elsewhere.

Test Suite¶

For advice on writing your tests, see Testing Your Code.

Location	`./test_sample.py` or `./tests`
Purpose	Package integration and unit tests.

Starting out, a small test suite will often exist in a single file:

Once a test suite grows, you can move your tests to a directory, like so:

```
tests/test_basic.py
tests/test_advanced.py
```

Obviously, these test modules must import your packaged module to test it. You can do this a few ways:

- Expect the package to be installed in site-packages.
- Use a simple (but *explicit*) path modification to resolve the package properly.

I highly recommend the latter. Requiring a developer to run `setup.py develop` to test an actively changing codebase also requires them to have an isolated environment setup for each instance of the codebase.

To give the individual tests import context, create a `tests/context.py` file:

```
import os
import sys
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__),
'..')))

import sample
```

Then, within the individual test modules, import the module like so:

```
from .context import sample
```

This will always work as expected, regardless of installation method.

Some people will assert that you should distribute your tests within your module itself – I disagree. It often increases complexity for your users; many test suites often require additional dependencies and runtime contexts.

Makefile¶

Location	`./Makefile`
Purpose	Generic management tasks.

If you look at most of my projects or any Pocoo project, you'll notice a Makefile lying around. Why? These projects aren't written in C… In short, make is an incredibly useful tool for defining generic tasks for your project.

Sample Makefile:

```
init:
    pip install -r requirements.txt

test:
    py.test tests

.PHONY: init test
```

Other generic management scripts (e.g. `manage.py` or `fabfile.py`) belong at the root of the repository as well.

Regarding Django Applications¶

I've noticed a new trend in Django applications since the release of Django 1.4. Many developers are structuring their repositories poorly due to the new bundled application templates.

How? Well, they go to their bare and fresh repository and run the following, as they always have:

```
$ django-admin.py startproject samplesite
```

The resulting repository structure looks like this:

```
README.rst
samplesite/manage.py
samplesite/samplesite/settings.py
samplesite/samplesite/wsgi.py
samplesite/samplesite/sampleapp/models.py
```

Don’t do this.

Repetitive paths are confusing for both your tools and your developers. Unnecessary nesting doesn’t help anybody (unless they’re nostalgic for monolithic SVN repos).

Let’s do it properly:

```
$ django-admin.py startproject samplesite .
```

Note the “`.`”.

The resulting structure:

```
README.rst
manage.py
samplesite/settings.py
samplesite/wsgi.py
samplesite/sampleapp/models.py
```

Structure of Code is Key¶

Thanks to the way imports and modules are handled in Python, it is relatively easy to structure a Python project. Easy, here, means that you do not have many constraints and that the module importing model is easy to grasp. Therefore, you are left with the pure architectural task of crafting the different parts of your project and their interactions.

Easy structuring of a project means it is also easy to do it poorly. Some signs of a poorly structured project include:

- Multiple and messy circular dependencies: if your classes Table and Chair in `furn.py` need to import Carpenter from `workers.py` to answer a question such as `table.isdoneby()`, and if conversely the class Carpenter needs to import Table and Chair to answer the question `carpenter.whatdo()`, then you have a circular dependency. In this case you will have to resort to fragile hacks such as using import statements inside methods or functions.

- Hidden coupling: each and every change in Table's implementation breaks 20 tests in unrelated test cases because it breaks Carpenter's code, which requires very careful surgery to adapt the change. This means you have too many assumptions about Table in Carpenter's code or the reverse.
- Heavy usage of global state or context: instead of explicitly passing `(height, width, type, wood)` to each other, Table and Carpenter rely on global variables that can be modified and are modified on the fly by different agents. You need to scrutinize all access to these global variables to understand why a rectangular table became a square, and discover that remote template code is also modifying this context, messing with table dimensions.
- Spaghetti code: multiple pages of nested if clauses and for loops with a lot of copy-pasted procedural code and no proper segmentation are known as spaghetti code. Python's meaningful indentation (one of its most controversial features) make it very hard to maintain this kind of code. So the good news is that you might not see too much of it.
- Ravioli code is more likely in Python: it consists of hundreds of similar little pieces of logic, often classes or objects, without proper structure. If you never can remember if you have to use FurnitureTable, AssetTable or Table, or even TableNew for your task at hand, you might be swimming in ravioli code.

Modules¶

Python modules are one of the main abstraction layers available and probably the most natural one. Abstraction layers allow separating code into parts holding related data and functionality.

For example, a layer of a project can handle interfacing with user actions, while another would handle low-level manipulation of data. The most natural way to separate these two layers is to regroup all interfacing functionality in one file, and all low-level operations in another file. In this case, the interface file needs to import the low-level file. This is done with the `import` and `from ... import` statements.

As soon as you use *import* statements you use modules. These can be either built-in modules such as *os* and *sys*, third-party modules you have installed in your environment, or your project's internal modules.

To keep in line with the style guide, keep module names short, lowercase, and be sure to avoid using special symbols like the dot (.) or question mark (?). So a file name like `my.spam.py` is one you should avoid! Naming this way will interfere with the way Python looks for modules.

In the case of *my.spam.py* Python expects to find a `spam.py` file in a folder named `my` which is not the case. There is an [example](#) of how the dot notation should be used in the Python docs.

If you'd like you could name your module `my_spam.py`, but even our friend the underscore should not be seen often in module names. However, using other characters (spaces or hyphens) in module names will prevent importing (- is the subtract operator), so try to keep module names short so there is no need to separate words. And, most of all, don't namespace with underscores; use submodules instead.

```
# OK
import library.plugin.foo
```

```
# not OK
import library.foo_plugin
```

Aside from some naming restrictions, nothing special is required for a Python file to be a module, but you need to understand the import mechanism in order to use this concept properly and avoid some issues.

Concretely, the `import modu` statement will look for the proper file, which is `modu.py` in the same directory as the caller if it exists. If it is not found, the Python interpreter will search for `modu.py` in the "path" recursively and raise an ImportError exception if it is not found.

Once `modu.py` is found, the Python interpreter will execute the module in an isolated scope. Any top-level statement in `modu.py` will be executed, including other imports if any. Function and class definitions are stored in the module's dictionary.

Then, the module's variables, functions, and classes will be available to the caller through the module's namespace, a central concept in programming that is particularly helpful and powerful in Python.

In many languages, an `include file` directive is used by the preprocessor to take all code found in the file and 'copy' it into the caller's code. It is different in Python: the included code is isolated in a module namespace, which means that you generally don't have to worry that the included code could have unwanted effects, e.g. override an existing function with the same name.

It is possible to simulate the more standard behavior by using a special syntax of the import statement: `from modu import *`. This is generally considered bad practice. **Using `import *` makes code harder to read and makes dependencies less compartmentalized**.

Using `from modu import func` is a way to pinpoint the function you want to import and put it in the local namespace. While much less harmful than `import` * because it shows explicitly what is imported in the local namespace, its only advantage over a simpler `import modu` is that it will save a little typing.

Very bad

```
[...]
from modu import *
[...]
x = sqrt(4)  # Is sqrt part of modu? A builtin? Defined above?
```

Better

```
from modu import sqrt
[...]
x = sqrt(4)  # sqrt may be part of modu, if not redefined in between
```

Best

```
import modu
[...]
x = modu.sqrt(4)  # sqrt is visibly part of modu's namespace
```

As mentioned in the Code Style section, readability is one of the main features of Python. Readability means to avoid useless boilerplate text and clutter; therefore some efforts are spent trying to achieve a certain level of brevity. But terseness and obscurity are the limits where brevity should stop. Being able to tell immediately where a class or function comes from, as in the `modu.func` idiom, greatly improves code readability and understandability in all but the simplest single file projects.

Packages¶

Python provides a very straightforward packaging system, which is simply an extension of the module mechanism to a directory.

Any directory with an `__init__.py` file is considered a Python package. The different modules in the package are imported in a similar manner as plain modules, but with a special behavior for the `__init__.py` file, which is used to gather all package-wide definitions.

A file `modu.py` in the directory `pack/` is imported with the statement `import pack.modu`. This statement will look for an `__init__.py` file in `pack` and execute all of its top-level statements. Then it will look for a file named `pack/modu.py` and execute all of its top-level statements. After these operations, any variable, function, or class defined in `modu.py` is available in the pack.modu namespace.

A commonly seen issue is to add too much code to `__init__.py` files. When the project complexity grows, there may be sub-packages and sub-sub-packages in a deep directory structure. In this case, importing a single item from a sub-sub-package will require executing all `__init__.py` files met while traversing the tree.

Leaving an `__init__.py` file empty is considered normal and even a good practice, if the package’s modules and sub-packages do not need to share any code.

Lastly, a convenient syntax is available for importing deeply nested packages: `import very.deep.module as mod`. This allows you to use *mod* in place of the verbose repetition of `very.deep.module`.

Object-oriented programming¶

Python is sometimes described as an object-oriented programming language. This can be somewhat misleading and needs to be clarified.

In Python, everything is an object, and can be handled as such. This is what is meant when we say, for example, that functions are first-class objects. Functions, classes, strings, and even types are objects in Python: like any object, they have a type, they can be passed as function arguments, and they may have methods and properties. In this understanding, Python is an object-oriented language.

However, unlike Java, Python does not impose object-oriented programming as the main programming paradigm. It is perfectly viable for a Python project to not be object-oriented, i.e. to use no or very few class definitions, class inheritance, or any other mechanisms that are specific to object-oriented programming.

Moreover, as seen in the modules section, the way Python handles modules and namespaces gives the developer a natural way to ensure the encapsulation and separation of abstraction layers, both being the most common reasons to use object-orientation. Therefore, Python programmers have more latitude to not use object-orientation, when it is not required by the business model.

There are some reasons to avoid unnecessary object-orientation. Defining custom classes is useful when we want to glue together some state and some functionality. The problem, as pointed out by the discussions about functional programming, comes from the “state” part of the equation.

In some architectures, typically web applications, multiple instances of Python processes are spawned to respond to external requests that can happen at the same time. In this case, holding some state in instantiated objects, which means keeping some static information about the world, is prone to concurrency problems or race conditions. Sometimes, between the initialization of the state of an object (usually done with the `__init__()` method) and the actual use of the object state through one of its methods, the world may have changed, and the retained state may be outdated. For example, a request may load an item in memory and mark it as read by a user. If another request requires the deletion of this item at the same time, it may happen that the deletion actually occurs after the first process loaded the item, and then we have to mark as read a deleted object.

This and other issues led to the idea that using stateless functions is a better programming paradigm.

Another way to say the same thing is to suggest using functions and procedures with as few implicit contexts and side-effects as possible. A function's implicit context is made up of any of the global variables or items in the persistence layer that are accessed from within the function. Side-effects are the changes that a function makes to its implicit context. If a function saves or deletes data in a global variable or in the persistence layer, it is said to have a side-effect.

Carefully isolating functions with context and side-effects from functions with logic (called pure functions) allow the following benefits:

- Pure functions are deterministic: given a fixed input, the output will always be the same.
- Pure functions are much easier to change or replace if they need to be refactored or optimized.
- Pure functions are easier to test with unit tests: There is less need for complex context setup and data cleaning afterwards.
- Pure functions are easier to manipulate, decorate, and pass around.

In summary, pure functions are more efficient building blocks than classes and objects for some architectures because they have no context or side-effects.

Obviously, object-orientation is useful and even necessary in many cases, for example when developing graphical desktop applications or games, where the things that are manipulated (windows, buttons, avatars, vehicles) have a relatively long life of their own in the computer's memory.

Decorators¶

The Python language provides a simple yet powerful syntax called 'decorators'. A decorator is a function or a class that wraps (or decorates) a function or a method. The 'decorated' function or method will replace the original 'undecorated' function or method. Because functions are first-class objects in Python, this can be done 'manually', but using the @decorator syntax is clearer and thus preferred.

```
def foo():
    # do something

def decorator(func):
    # manipulate func
    return func

foo = decorator(foo)  # Manually decorate

@decorator
def bar():
    # Do something
# bar() is decorated
```

This mechanism is useful for separating concerns and avoiding external unrelated logic 'polluting' the core logic of the function or method. A good example of a piece of functionality that is better handled with decoration is memoization or caching: you want to store the results of an expensive function in a table and use them directly instead of recomputing them when they have already been computed. This is clearly not part of the function logic.

Context Managers¶

A context manager is a Python object that provides extra contextual information to an action. This extra information takes the form of running a callable upon initiating the context using the `with` statement, as well as running a callable upon completing all the code inside the `with` block. The most well known example of using a context manager is shown here, opening on a file:

```
with open('file.txt') as f:
    contents = f.read()
```

Anyone familiar with this pattern knows that invoking `open` in this fashion ensures that `f`’s `close` method will be called at some point. This reduces a developer’s cognitive load and makes the code easier to read.

There are two easy ways to implement this functionality yourself: using a class or using a generator. Let’s implement the above functionality ourselves, starting with the class approach:

```
class CustomOpen(object):
    def __init__(self, filename):
        self.file = open(filename)

    def __enter__(self):
        return self.file

    def __exit__(self, ctx_type, ctx_value,
ctx_traceback):
        self.file.close()

with CustomOpen('file') as f:
    contents = f.read()
```

This is just a regular Python object with two extra methods that are used by the `with` statement. CustomOpen is first instantiated and then its `__enter__` method is called and whatever `__enter__` returns is assigned to `f` in the `as f` part of the statement. When the contents of the `with` block is finished executing, the `__exit__` method is then called.

And now the generator approach using Python's own contextlib:

```
from contextlib import contextmanager

@contextmanager
def custom_open(filename):
    f = open(filename)
    try:
        yield f
    finally:
        f.close()

with custom_open('file') as f:
    contents = f.read()
```

This works in exactly the same way as the class example above, albeit it's more terse. The `custom_open` function executes until it reaches the `yield` statement. It then gives control back to the `with` statement, which assigns whatever was `yield`'ed to *f* in the `as f` portion. The `finally` clause ensures that `close()` is called whether or not there was an exception inside the `with`.

Since the two approaches appear the same, we should follow the Zen of Python to decide when to use which. The class approach might be better if there's a considerable amount of logic to encapsulate. The function approach might be better for situations where we're dealing with a simple action.

Dynamic typing¶

Python is dynamically typed, which means that variables do not have a fixed type. In fact, in Python, variables are very different from what they are in many other languages, specifically statically-typed languages. Variables are not a segment of the computer's memory where some value is written, they are 'tags' or 'names' pointing to objects. It is therefore possible for the variable 'a' to be set to the value 1, then to the value 'a string', then to a function.

The dynamic typing of Python is often considered to be a weakness, and indeed it can lead to complexities and hard-to-debug code. Something named 'a' can be set to many different things, and the developer or the maintainer needs to track this name in the code to make sure it has not been set to a completely unrelated object.

Some guidelines help to avoid this issue:

- Avoid using the same variable name for different things.

Bad

```
a = 1
a = 'a string'
def a():
    pass  # Do something
```

Good

```
count = 1
msg = 'a string'
def func():
    pass  # Do something
```

Using short functions or methods helps reduce the risk of using the same name for two unrelated things.

It is better to use different names even for things that are related, when they have a different type:

Bad

```
items = 'a b c d'  # This is a string...
items = items.split(' ')  # ...becoming a list
items = set(items)  # ...and then a set
```

There is no efficiency gain when reusing names: the assignments will have to create new objects anyway. However, when the complexity grows and each assignment is separated by other lines of code, including 'if' branches and loops, it becomes harder to ascertain what a given variable's type is.

Some coding practices, like functional programming, recommend never reassigning a variable. In Java this is done with the *final* keyword. Python does not have a *final* keyword and it would be against its philosophy anyway. However, it may be a good discipline to avoid assigning to a variable more than once, and it helps in grasping the concept of mutable and immutable types.

Mutable and immutable types¶

Python has two kinds of built-in or user-defined types.

Mutable types are those that allow in-place modification of the content. Typical mutables are lists and dictionaries: All lists have mutating methods, like `list.append()` or `list.pop()`, and can be modified in place. The same goes for dictionaries.

Immutable types provide no method for changing their content. For instance, the variable x set to the integer 6 has no "increment" method. If you want to compute x + 1, you have to create another integer and give it a name.

```
my_list = [1, 2, 3]
my_list[0] = 4
print my_list  # [4, 2, 3] <- The same list has
changed

x = 6
```

```
x = x + 1  # The new x is another object
```

One consequence of this difference in behavior is that mutable types are not “stable”, and therefore cannot be used as dictionary keys.

Using properly mutable types for things that are mutable in nature and immutable types for things that are fixed in nature helps to clarify the intent of the code.

For example, the immutable equivalent of a list is the tuple, created with `(1, 2)`. This tuple is a pair that cannot be changed in-place, and can be used as a key for a dictionary.

One peculiarity of Python that can surprise beginners is that strings are immutable. This means that when constructing a string from its parts, appending each part to the string is inefficient because the entirety of the string is copied on each append. Instead, it is much more efficient to accumulate the parts in a list, which is mutable, and then glue (`join`) the parts together when the full string is needed. List comprehensions are usually the fastest and most idiomatic way to do this.

Bad

```
# create a concatenated string from 0 to 19 (e.g.
"012..1819")
nums = ""
for n in range(20):
    nums += str(n)   # slow and inefficient
print nums
```

Better

```
# create a concatenated string from 0 to 19 (e.g.
"012..1819")
nums = []
for n in range(20):
    nums.append(str(n))
```

```
print "".join(nums)  # much more efficient
```

Best

```
# create a concatenated string from 0 to 19 (e.g.
"012..1819")
nums = [str(n) for n in range(20)]
print "".join(nums)
```

One final thing to mention about strings is that using `join()` is not always best. In the instances where you are creating a new string from a pre-determined number of strings, using the addition operator is actually faster, but in cases like above or in cases where you are adding to an existing string, using `join()` should be your preferred method.

```
foo = 'foo'
bar = 'bar'

foobar = foo + bar  # This is good
foo += 'ooo'  # This is bad, instead you should
do:
foo = ''.join([foo, 'ooo'])
```

Note

You can also use the % formatting operator to concatenate a pre-determined number of strings besides `str.join()` and +. However, **PEP 3101** discourages the usage of the % operator in favor of the `str.format()` method.

```
foo = 'foo'
bar = 'bar'

foobar = '%s%s' % (foo, bar) # It is OK
foobar = '{0}{1}'.format(foo, bar) # It is better
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) #
It is best
```

Vendorizing Dependencies¶