Organizing your Python Code

In spite of yourself

Keno Leon
Jan 4 · 13 min read ★

Original photo by Tara Shankar at Pexels

The Problem:

I am not an organized coder, or rather I struggle with organizing my code, nothing new, this has been going on for a long time…

> 📝 Many years ago I made a php image CMS of a few thousand lines… in one single file, most of the functionality was inside a giant loop (*plenty of nested loops, though*), no functions, no comments, variables recklessly added here and there, it was so bad it took me longer to read the code than to add functionality, in desperation I printed the whole thing, took it to the bar and drunkenly tried to make sense of it, it didn't and at some point I gave up on it.

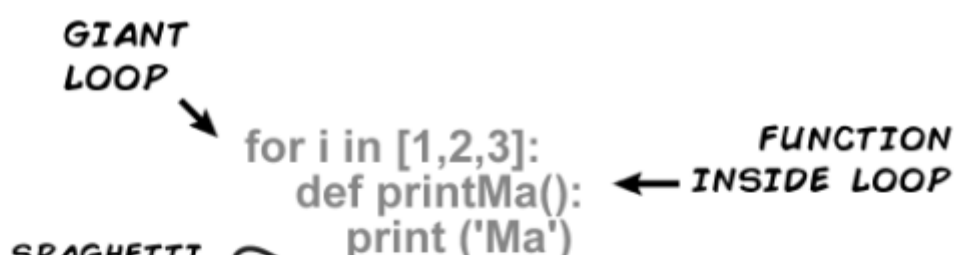
Ever since, organizing my code has been something I think one should strive for, mostly to avoid failure, and I try to, **it makes life easier for you and whoever ends up reading and maintaining your code**, let's explore some aspects and solutions here; this is meant as a simple, beginners friendly overview, not a definite resource, the subject as you will see can get quite complex.

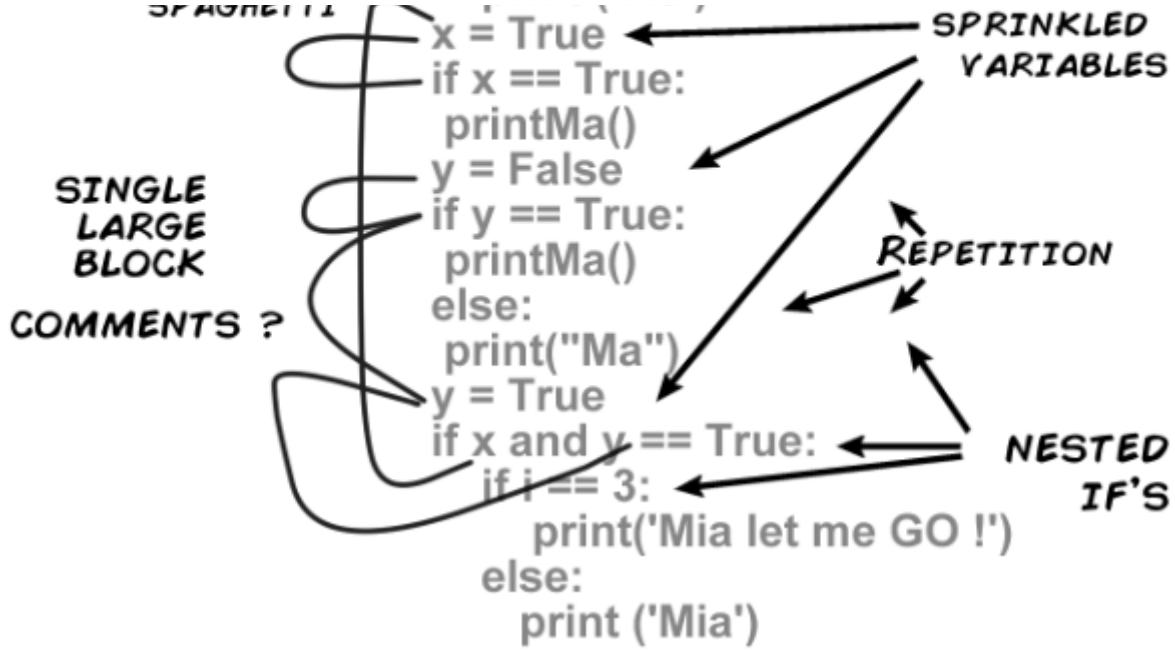
. . .

Some spaghetti code for starters:

```
for i in [1,2,3]:
    def printMa():
        print ('Ma')
    x = True
    if x == True:
     printMa()
    y = False
    if y == True:
     printMa()
    else:
     print("Ma")
    y = True
    if x and y == True:
        if i == 3:
            print('Mia let me GO !')
        else:
           print ('Mia')

# OUTPUT:
# Ma Ma mia Ma Ma mia Ma Ma mia let me GO !
```

The telltale sign you are dealing with spaghetti code is that it is hard to read, you don't know what it does and how it does it, the name spaghetti comes from needing to reference back and forth variables and functions which end up resembling spaghetti, but there's much more messiness going on here:





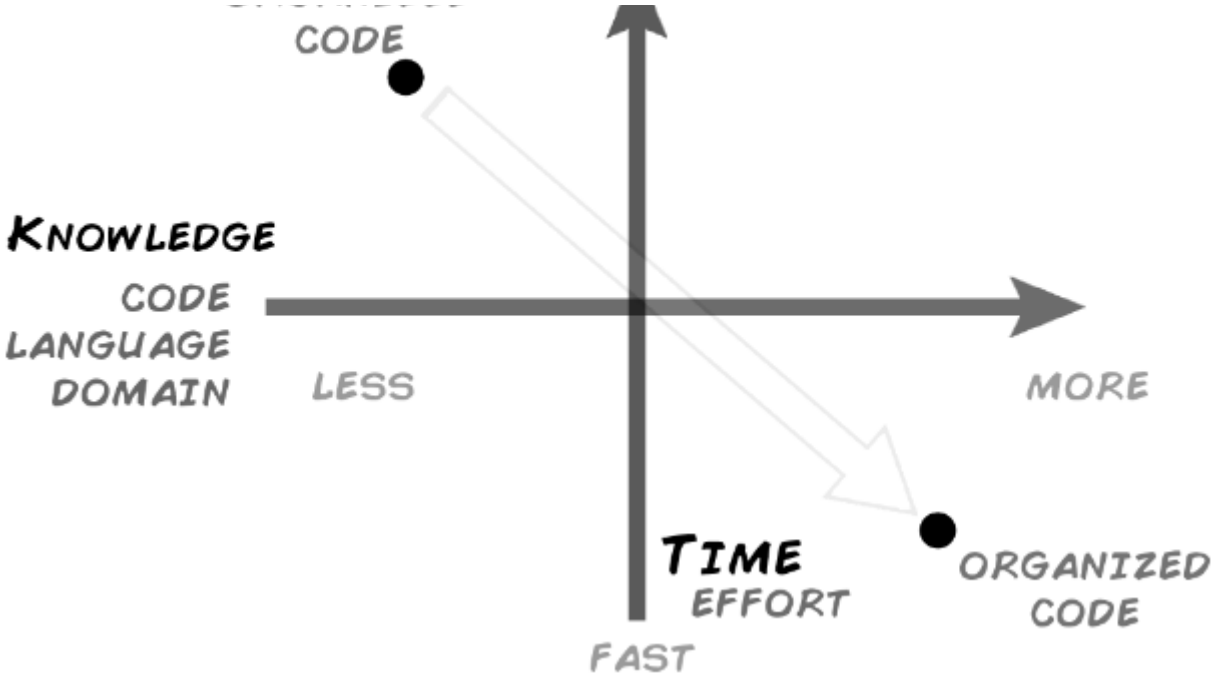
Hey it works though !

We can define a program or script as a set of instructions, usually along with some data we give a computer to do something we want, sing a song in this case, the computer will happily do whatever we tell it as long as it doesn't interfere with some internal rules (the language syntax), organization then is something we do to address our limits and shortcomings as humans, and since we are all a little different, what works for most might not work and/or make sense for you and vice versa, and that is mostly ok, as long as you understand that when writing in a team environment organization is something that needs to be agreed upon.

. . .

To organize or not ?

The first question one might ask is why we need to organize our code, after all, spaghetti code can work, even if it becomes an unreadable, hard to maintain mess, further, while you are starting out with a new language or programming as a craft or hobby, realistically you might not know enough to properly organize your code, if we add internal requirements like style guides, testing, deadlines, documentation and source control, it is easy to see why not organizing your code has a certain appeal, it's a balancing act I think we can summarize as follows:



Ideally you want to follow the arrow so you can code fast, effortless and organized.

There is a real tradeoff (*a close relative to technical debt*) where the more you know the easier it is to organize your code, note that knowledge is not limited to your language of choice (*in this case python*), but also to your domain (*that thing you are trying to make*) and the codebase itself (*especially true for a new developer dropped into an existing codebase*).

If time and effort are involved, then it follows that I need to sell you the benefits of organizing your code and that is what I will try, here are a few layers of organizational upgrades if you will, some basic examples and what you get from them.

. . .

Level 1: Functions and classes.

Both functions and classes are natural aggregators: Functions typically deal with statements (*think actions or verbs and sentences*) and classes with objects (*think well, classes of things, or nouns and adjectives*), programmatically they are deep subjects on their own and the cornerstones of the language, yet you could have a complex script or program run without them, so why use them ?

A well designed function will save you space and can be used as a sentence, building block or logic unit, a well designed class will dramatically expand your vocabulary , and

together they will allow you to speak in paragraphs rather than yell commands willy nilly, let's for instance rewrite the previous example with **Functions**:

```
def sing(line):
    print ('Ma Ma')
    if line == 1:
        print('Mia')
    else:
        print ('Mia let me GO !')

sing(1)
sing(1)
sing(2)

# OUTPUT:
# Ma Ma Mia Ma Ma Mia Ma Ma Mia let me GO !
```

Classes (*in the words of the docs*) **`bundle data and functionality together`** , this allows you to start thinking in terms of more complex things, here for instance we can create a class that stands for a **`chorusSinger`** , sure there is more code to contend with, but we can now create unlimited chorus singers and ask them to sing the appropriate lines:

```
class chorusSinger:
    """This class creates a chorus singer that
        can sing one of two lines."""

    line1 = "Ma Ma Mia"
    line2 = "let me GO !"

    def sing(self, line):
        """This function sings one of 2 lines"""
        if line == 1:
            print(self.line1)
        else:
            print(self.line2)

# Create chorus singers:
chorusSinger1 = chorusSinger()
chorusSinger2 = chorusSinger()

# Let them sing !:
chorusSinger1.sing(1)
chorusSinger2.sing(1)
chorusSinger1.sing(1)
chorusSinger2.sing(2)

Output:
```

```
Singer 1: Ma Ma Mia
Singer 2: Ma Ma Mia
Singer 1: Ma Ma Mia
Singer 2: let me GO !
```

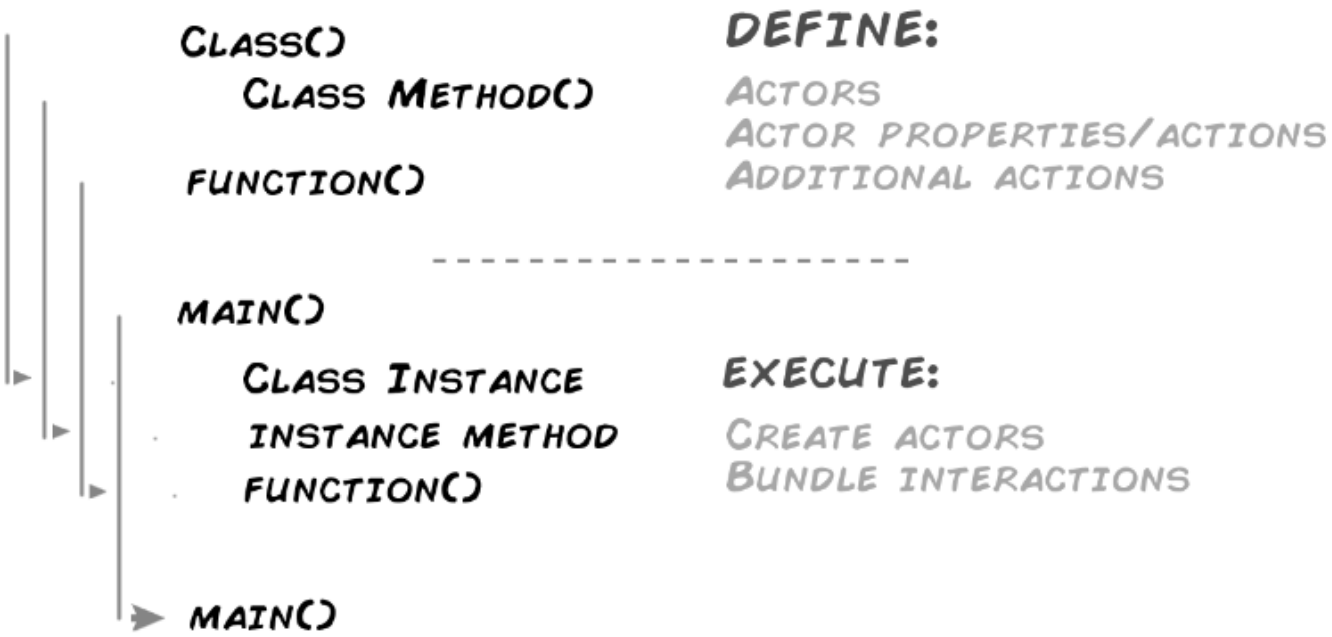
Note that we also added comments in the form of **“”” docstrings”””** which can later be used to document your code and **#inline comments** , which help you re-read your own code.

We are not quite done with functions and classes, the execution if you notice this previous example is just left hanging there at the end, if you wanted to add sections or other singers it could get messy spaghetti, so we can further organize by adding a function that does that, once more what we earn is the power of multiplicity along with readability:

```
class chorusSinger:
    """This class creates a chorus singer that
        can sing one of two lines."""

    line1 = "Ma Ma Mia"
    line2 = "let me GO !"

    def sing(self, line):
        """This function sings one of 2 lines"""
        if line == 1:
            print(self.line1)
        else:
            print(self.line2)

def singPart():
    """Creates Singers and tells them to sing"""
    chorusSinger1 = chorusSinger()
    chorusSinger2 = chorusSinger()
    chorusSinger1.sing(1)
    chorusSinger2.sing(1)
    chorusSinger1.sing(1)
    chorusSinger2.sing(2)

def main():
    """Plays parts"""
    singPart()
    singPart()

# Runs the program
main()

# OUTPUT:
  Singer 1: Ma Ma Mia
  Singer 2: Ma Ma Mia
```

```
Singer 1: Ma Ma Mia
Singer 2: let me GO !

Singer 1: Ma Ma Mia
Singer 2: Ma Ma Mia
Singer 1: Ma Ma Mia
Singer 2: let me GO !
```

This simple structure is fairly common, can serve as the basis of many short programs or scripts and open the door to more complex ones; here's an overview of this pattern:



ℹ️ : A problem you will soon encounter with this scheme, especially if you are starting out with OOP, is how to have a global variable or state you can access from within functions, and a second but closely related one is how to wrangle multiple objects (instantiate them, name them, call them etc, etc) check out my other post on alternatives to globals for a solution to both problems if you get stuck, but this next section should also help clarify these issues.

. . .

Level 2: Single file structure.

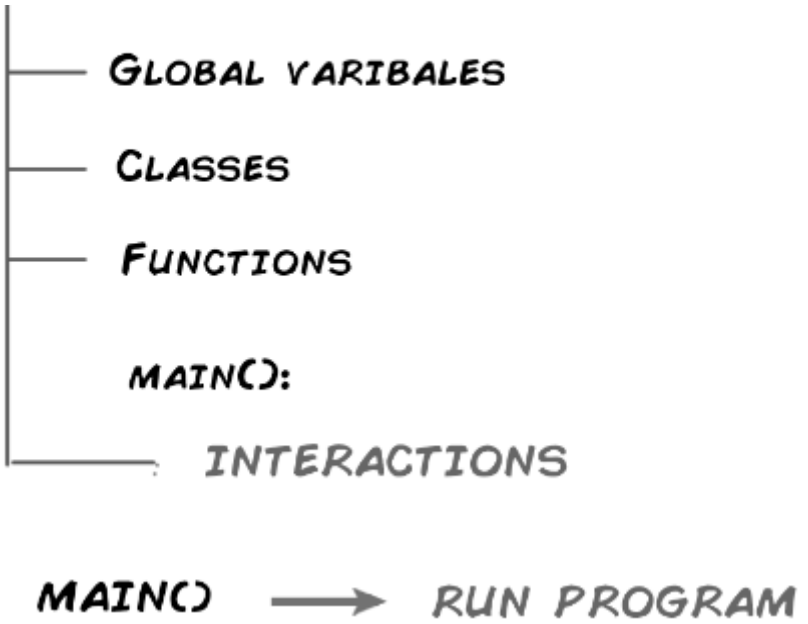
Most if not all the code one writes at first is usually a single file, and even advanced scripts with a lot of functionality can fit neatly into one, there is no set rule as to how you organize your single file script, but certain conventions do seem to exist, here, we expand on the previous concepts and include things like imports and variables:

⚠ For the next example I'll be using a Text to Speech library called **pyttsx3 | pip install pyttsx3**

Onwards :

```
"""
Sings 2 lines in different voices
"""

import pyttsx3

TTS = pyttsx3.init()
VOICES = TTS.getProperty('voices')

LINE1 = 'Ma Ma Mia'
LINE2 = 'Let me GO!'

def sing(gender, PART):
    """Sings a given line in a given voice"""
    if gender == 'male':
        TTS.setProperty('voice', VOICES[0].id)
    else:
        TTS.setProperty('voice', VOICES[1].id)
    TTS.say(PART)

def main():
    sing('male', LINE1)
    sing('female', LINE1)
    sing('male', LINE1)
    sing('female', LINE1 + LINE2)
    TTS.runAndWait()

main()

OUTPUT:

# You should hear 2 different voices singing 2 different parts
```

What's new here besides the unrelated upgrade to a robotic voice, is that we have imports and data (Constants, globals) up top followed by functions (*I omitted classes for brevity, but you can add them before your functions*), and finally the execution, which is a very similar structure to the previous level :



Couple of things worth noting in this pattern, if you use classes as objects, you will need to hold a global reference to interact with them in your functions and main function; lists `[]` are a popular way of referencing them (*this should solve the problems previously mentioned*), second; most game and graphic engines have a single event loop which usually goes in the main function, for more complex structures like GUI's with multithreading you'll have to rely on patterns which we'll cover in a minute.

. . .

Level 3: Modules and packages

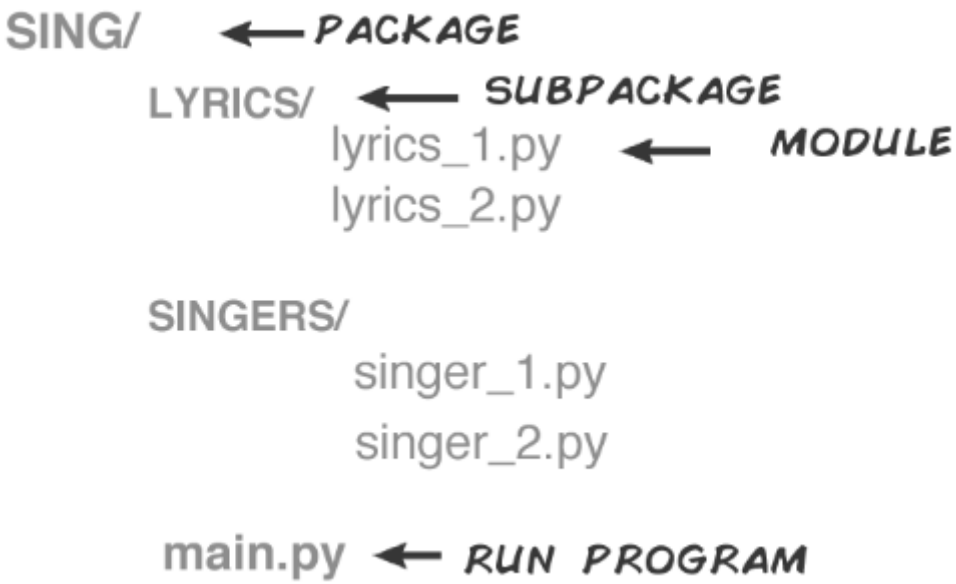
You are most likely to encounter packages and modules while writing single file scripts by importing a module or package like we just did.

```
First definitions: a module in python is simply a file you import, a package is a collection of files or modules that have a certain hierarchy.

For a thorough description consult the documentation:

https://docs.python.org/3/tutorial/modules.html#packages
```

Fundamentally, you gain the power of an explicit hierarchy when organizing your code into modules and packages (*along with readability*); according to people that study systems, a hierarchy serves the underlying activity (*whatever your code is trying to*

accomplish), by allowing the individual parts to focus on one or a few tasks, in other words, all your modules except one or a few should be specialized.

Let's say we want to make a lyrics singing program other people can use, we can either cram it all into a single file, or if we are using modules and packages come up with a certain hierarchy :



Once we have this or other type of hierarchy, we can use import statements from our main file to access subpackages:

FILE: lyrics_1.py

```
LYRICS = 'MA MA MIA'
```

FILE: singer_1.py

```
def SING(LYRICS):
    print (LYRICS)
```

FILE: main.py

```
import LYRICS.lyrics_1
import SINGERS.singer_1

SINGERS.singer_1.SING(LYRICS.lyrics_1.LYRICS)
```

OUTPUT: "MA MA MIA"

This is once more a simple example made for demonstration purposes, folder and file structures are usually more complex, files have Initialization scripts and imports can have extra attributes to make life easier, `import LYRICS.lyrics_1 as LYRICS_1` can save you some typing for instance, but separating data and functions across folders is a good place to start with this type of organization.

The advantage here is that you can easily tell others or your future self to add lyrics or singers by just looking at the folder structure, this time saved rereading your code is unfortunately spent organizing your code the first time around, but it is essential if you want to involve more people in your code…

. . .

> **Note:** These last 2 levels build upon a lot of information, and as such could be consider advanced, these are also meant for professional and open source work, not to say you shouldn't try them, just that they do require some time to understand, so don't get discouraged if at first you don't understand or feel comfortable with them. 🙂

Level 4: Patterns and recipes.

The problem that patterns and recipes try to solve is the following, you are trying to code something and rather than reinvent the wheel, you are willing to use someone else's solution, maybe this solution is so popular that a lot of people use it, in this way, you will be solving both your problem and the organization problem in one pass.

> **Patterns** Usually deal with Data Structures and the flow of information, **Recipes** on the other hand deal with more granular problems.

It is tempting to think this is a perfect solution, but there are a few things against it:

- You will spend some time learning the pattern or recipe.
- You are missing the learning that comes with trying and failing.

- Using something because it's the standard might not always be the best design choice.
- Not everyone uses or knows about the pattern you chose.

So what does a pattern look like ?

The following is called a **Singleton** pattern, and it is used when you need one and only one instance of a certain object:

```
class Singleton:
   __instance = None
   @staticmethod
   def getInstance():
      """ Static access method. """
      if Singleton.__instance == None:
         Singleton()
      return Singleton.__instance
   def __init__(self):
      """ Virtually private constructor. """
      if Singleton.__instance != None:
         raise Exception("This class is a singleton!")
      else:
         Singleton.__instance = self

X = Singleton()
print (X)

Y = Singleton.getInstance()
print (Y)

Z = Singleton.getInstance()
print (Z)

OUTPUT:
<__main__.Singleton object at 0x1030db748>
<__main__.Singleton object at 0x1030db748>
<__main__.Singleton object at 0x1030db748>
```

A singleton is handy in organizing since it can usually be at the top of the hierarchy like a King or CEO (*there can only be one*). You might not immediately (*or ever*) have the need for this pattern, but if you encounter it, you will know a great deal about the program you are working on.

Where to find patterns and Recipes ?

Usually scattered around the web in tutorials much like this one and certain books, here's a few resources to get you started:

Patterns:

Python Design Patterns - Singleton

This pattern restricts the instantiation of a class to one object. It is a type of creational pattern and involves only...

www.tutorialspoint.com

The Pattern Concept - Python 3 Patterns, Recipes and Idioms

"Design patterns help you learn from others' successes instead of your own failures [1]." Probably the most important...

python-3-patterns-idioms-test.readthedocs.io

Recipes:

Python Cookbook

If you need help writing programs in Python 3, or want to update older Python 2 code, this book is just the ticket...

shop.oreilly.com

Stack Overflow - Where Developers Learn, Share, & Build Careers

We build products that empower developers and connect them to solutions that enable productivity, growth, and...

stackoverflow.com

. . .

Level 5: Templates and boilerplates.

Most if not all the code one encounters in the wild and sometimes work has certain level of organization, this organization much like patterns is someone else's idea, how it was arrived at is sometimes a mystery, the code and file structure, along with the projects style are all you have to make sense of it.

Templates and boilerplates are community or industry driven solutions for when you want to start a project and need some basic scaffolding, let's say for instance you wanted to make a Flask (*a web framework for python*) project, you could do it yourself, or search for a flask boilerplate, here's one :

realpython/flask-boilerplate

Hello. Want to get started with Flask quickly? Good. You came to the right place. This Flask application framework is...

github.com

After running the repo you will have the following file structure, along with a fully functional skeleton app, hard to beat for a few minutes of work.

```
├── Procfile
├── Procfile.dev
├── README.md
├── app.py
├── config.py
├── error.log
├── forms.py
├── models.py
├── requirements.txt
├── static
│   ├── css
│   │   ├── bootstrap-3.0.0.min.css
│   │   ├── bootstrap-theme-3.0.0.css
│   │   ├── bootstrap-theme-3.0.0.min.css
│   │   ├── font-awesome-3.2.1.min.css
│   │   ├── layout.forms.css
│   │   ├── layout.main.css
│   │   ├── main.css
│   │   ├── main.quickfix.css
│   │   └── main.responsive.css
│   ├── font
│   │   ├── FontAwesome.otf
│   │   ├── fontawesome-webfont.eot
│   │   ├── fontawesome-webfont.svg
│   │   ├── fontawesome-webfont.ttf
│   │   └── fontawesome-webfont.woff
│   ├── ico
```

```
│   │   ├── apple-touch-icon-114-precomposed.png
│   │   ├── apple-touch-icon-144-precomposed.png
│   │   ├── apple-touch-icon-57-precomposed.png
│   │   ├── apple-touch-icon-72-precomposed.png
│   │   └── favicon.png
│   ├── img
│   └── js
│       ├── libs
│       │   ├── bootstrap-3.0.0.min.js
│       │   ├── jquery-1.10.2.min.js
│       │   ├── modernizr-2.6.2.min.js
│       │   └── respond-1.3.0.min.js
│       ├── plugins.js
│       └── script.js
└── templates
    ├── errors
    │   ├── 404.html
    │   └── 500.html
    ├── forms
    │   ├── forgot.html
    │   ├── login.html
    │   └── register.html
    ├── layouts
    │   ├── form.html
    │   └── main.html
    └── pages
        ├── placeholder.about.html
        └── placeholder.home.html
```

There is even a family of templates you can use for different python projects (*even a template to make templates*) :

cookiecutter/cookiecutter

A command-line utility that creates projects from cookiecutters (project templates), e.g. Python package projects...

github.com

There are of course caveats:

- You have to learn your template of choice along with it's quirks and design logic.
- Your template might include more features than you need and you end with a bloated app.
- Your template might not include the feature that you need and require considerable effort to include it.

- Even worse, the template might not be suited for your feature, in which case, you might spend a lot of time and still not get a working project.

> **So what do I use ?** Depends, at work I usually go with templates and boilerplates, for my own projects I usually start with classes and functions on a single file, as the project progresses I usually start making a single file structure and if the project grows it ends up in a file hierarchy, I use common patterns all the time but quickly forget that I am using them, especially for things like MVC.

. . .

Conclusion

Organizing your code in python can be hard, but not organizing it is even worse, it is hard because it requires you to learn about python and your project, but if you are willing to start small and put in the time you will eventually become faster and organizing your code will be almost effortless.

Here I've tried to give you a basic roadmap of what progressive organizational skills look like on python, (*but it can also work for other languages*) and hope you find the process a little less intimidating.

tl;dr: With great organizational power comes great learning power bill. 🙂

Thanks for reading.