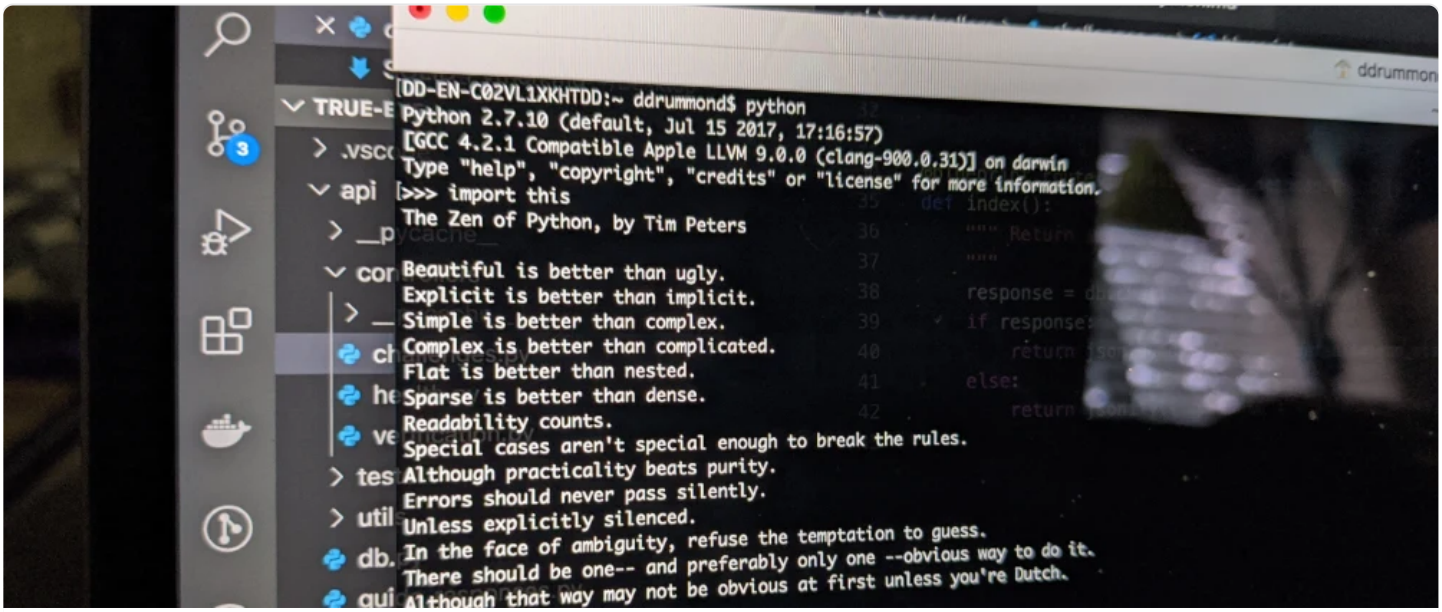

A Pythonic Guide to SOLID Design Principles

#python #codequality

Derek D. 20 de mar. de 2020 • *Updated on 10 de set. de 2020* • 11 min read

People that know me will tell you I am a big fan of the SOLID Design Principles championed by Robert C. Martin (Uncle Bob). Over the years I've used these principles in C#, PHP, Node.js, and Python. Everywhere I took them they were generally well-received...except when I started working in Python. I kept getting comments like "It's not a very Pythonic way to do things" during code review. I was new to Python at the time so I didn't really know how to respond. I didn't know what Pythonic code meant or looked like and none of the explanations offered were very satisfying. It honestly pissed me off. I felt like people were using the Pythonic way to cop-out of writing more disciplined code. Since then I've been on a mission to prove SOLID code is Pythonic. That's the wind up now here's the pitch.

What is SOLID Design

Michael Feathers can be credited for creating the mnemonic SOLID which is based on principles from Robert C. Martin's paper, "Design Principles and Design Patterns". The principles are

- Single Responsibility Principle
- Open Closed Princple
- Liskov's Substitutablilty Principle

- Interface Segregation Principle
- Dependency Inversion Principle

We will cover these in more detail shortly. The most important thing to note about the SOLID design principles is they are meant to be used holistically. Choosing one and just one is not going to do much for you. It's when used together you start to see the real value in these principles.

What is the Pythonic Way

Although there is no official definition for the Pythonic way a little Googling gives you several answers along this general vain.

> "Above correct syntax Pythonic code follows the normally accepted conventions of the Python community, and uses the language in a way that follows the founding philosophy." - Derek D.

I think the most accurate description comes from the Zen of Python.

> The Zen of Python, by Tim Peters
>
> Beautiful is better than ugly.
> Explicit is better than implicit.
> Simple is better than complex.
> Complex is better than complicated.
> Flat is better than nested.
> Sparse is better than dense.
> Readability counts.
> Special cases aren't special enough to break the rules.
> Although practicality beats purity.
> Errors should never pass silently.
> Unless explicitly silenced.
> In the face of ambiguity, refuse the temptation to guess.
> There should be one-- and preferably only one --obvious way to do it.
> Although that way may not be obvious at first unless you're Dutch.
> Now is better than never.
> Although never is often better than *right* now.
> If the implementation is hard to explain, it's a bad idea.

> If the implementation is easy to explain, it may be a good idea.
> Namespaces are one honking great idea -- let's do more of those!

One Last Thing

Before I jump right into the principles and how they relate to the Zen of Python, there's one thing I want to do that no other SOLID tutorial does. Instead of using a different code snippet for each principle, We are going to work with a single code base and make it more SOLID as we cover each principle. Here is the code we are going to start with.

```
class FTPClient:
  def __init__(self, **kwargs):
    self._ftp_client = FTPDriver(kwargs['host'], kwargs['port'])
    self._sftp_client = SFTPDriver(kwargs['sftp_host'], kwargs['user'], kwargs[

  def upload(self, file:bytes, **kwargs):
    is_sftp = kwargs['sftp']
    if is_sftp:
      with self._sftp_client.Connection() as sftp:
        sftp.put(file)
    else:
      self._ftp_client.upload(file)

  def download(self, target:str, **kwargs) -> bytes:
    is_sftp = kwargs['sftp']
    if is_sftp:
      with self._sftp_client.Connection() as sftp:
        return sftp.get(target)
    else:
      return self._ftp_client.download(target)
```

Single Responsibility Principle (SRP)

Definition: Every module/class should only have one responsibility and therefore only one reason to change.

Relevant Zen: There should be one-- and preferably only one --obvious way to do things

The Single Responsibility Principle (SRP) is all about increasing cohesion and decreasing coupling by organizing code around responsibilities. It's not a big leap to see why that happens. If all the code for any given responsibility is in a single place that's cohesive and while responsibilities may be similar they don't often overlap. Consider this non-code example. If it is your responsibility to sweep and my

responsibility to mop there's no reason for me to keep track of whether or not the floor has been swept. I can just ask you, "has the floor been swept"? and base my action according to your response.

I find it useful to think of responsibilities as use cases, which is how our Zen comes into play. Each use case should only be handled in one place, in turn, creating one obvious way to do things. This also satisfies the, "one reason to change" portion of the SRP's definition. The only reason this class should change is if the use case has changed.

Examining our original code we can see the class does not have a single responsibility because it has to manage connection details for an FTP, and SFTP server. Furthermore, the methods don't even have a single responsibility because both have to choose which protocol they will be using. This can be fixed by splitting the `FTPClient` class into 2 classes each with one of the responsibilities.

```
class FTPClient:
  def __init__(self, host, port):
    self._client = FTPDriver(host, port)

  def upload(self, file:bytes):
    self._client.upload(file)

  def download(self, target:str) -> bytes:
    return self._client.download(target)


class SFTPClient(FTPClient):
  def __init__(self, host, user, password):
    self._client = SFTPDriver(host, username=user, password=password)

  def upload(self, file:bytes):
    with self._client.Connection() as sftp:
      sftp.put(file)

  def download(self, target:str) -> bytes:
    with self._client.Connection() as sftp:
      return sftp.get(target)
```

One quick change and our code is already feeling much more Pythonic. The code is sparse, and not dense, simple not complex, flat and not nested. If you're not on board yet think about how the original code would look with error handling compared to the code following SRP.

Open Closed Principle (OCP)

Definition: Software Entities (classes, functions, modules) should be open for extension but closed to change.

Relevant Zen: Simple is better than complex. Complex is better than complicated.

Since the definition of change and extension are so similar it is easy to get overwhelmed by the Open Closed Principle. I've found the most intuitive way to decide if I'm making a change or extension is to think about function signatures. A change is anything that forces calling code to be updated. This could be changing the function name, swapping the order of parameters, or adding a non-default parameter. Any code that calls the function would be forced to change in accordance with the new signature. An extension, on the other hand, allows for new functionality, without having to change the calling code. This could be renaming a parameter, adding a new parameter with a default value, or adding the `*arg`, or `**kwargs` parameters. Any code that calls the function would still work as originally written. The same rules apply to classes as well.

Here is an example of adding support for bulk operations.

Your gut reaction is probably to add a `upload_bulk` and `download_bulk` functions to the `FTPClient` class. Fortunately, that is also a SOLID way to handle this use case.

```
class FTPClient:
  def __init__(self, host, port):
      ... # For this example the __init__ implementation is not significant

  def upload(self, file:bytes):
      ... # For this example the upload implementation is not significant

  def download(self, target:str) -> bytes:
      ... # For this example the download implementation is not significant

  def upload_bulk(self, files:List[str]):
    for file in files:
      self.upload(file)

  def download_bulk(self, targets:List[str]) -> List[bytes]:
    files = []
    for target in targets:
      files.append(self.download(target))

    return files
```

In this case, it's better to extend the class with functions than extend through inheritance, because a BulkFTPClient child class would have to change the function signature for download reflecting it returns a list of bytes rather than just bytes, violating the Open Closed Principle as well as Liskov's Substituitability Principle.

Liskov's Substituitability Principle (LSP)

Definition: If S is a subtype of T, then objects of type T may be replaced with objects of Type S.

Relevant Zen: Special cases aren't special enough to break the rules.

Liskov's Substituitablity Principle was the first of the SOLID design principles I learned of and the only one I learned at University. Maybe that's why this one is so intuitive to me. A plain English way of saying this is, "Any child class can replace its parent class without breaking functionality."

You may have noticed all of the FTP client classes so far have the same function signatures. That was done purposefully so they would follow Liskov's Substituitability Principle. An SFTPClient object can replace an FTPClient object and whatever code is calling `upload`, or `download`, is blissfully unaware.

Another specialized case of FTP file transfers is supporting FTPS (yes FTPS and SFTP are different). Solving this can be tricky because we have choices. They are:
1. Add `upload_secure`, and `download_secure` functions.
2. Add a secure flag through `**kwargs`.
3. Create a new class, `FTPSClient`, that extends `FTPClient`.

For reasons that we will get into during the Interface Segregation, and Dependency Inversion principles the new FTPSClient class is the way to go.

```
class FTPClient:
  def __init__(self, host, port):
    ...

  def upload(self, file:bytes):
    ...

  def download(self, target:str) -> bytes:
    ...

class FTPSClient(FTPClient):
    def __init__(self, host, port, username, password):
        self._client = FTPSDriver(host, port, user=username, password=password)
```

This is exactly the kind of edge case inheritance is meant for and following Liskov's makes for effective polymorphism. You'll note than now `FTPClient`'s can be replaced by an `FTPSClient` or `SFTPClient`. In fact, all 3 are interchangeable which brings us to interface segregation.

Interface Segregation Principle (ISP)

Definition: A client should not depend on methods it does not use.

Relevant Zen: Readability Counts && complicated is better than complex.

Unlike Liskov's, The Interface Segregation Principle was the last and most difficult principle for me to understand. I always equated it to the interface keyword, and most explanations for SOLID design don't do much to dispel that confusion. additionally, most guides I've found try to break everything up into tiny interfaces most often with a single function per-interface because "too many interfaces are better than too few".

There are 2 issues here, first Python doesn't have interfaces, and second languages like C# and Java that do have interfaces, breaking them up too much always ends up with interfaces implementing interfaces which can get complex and complex is not Pythonic.

First I want to explore the too small of interfaces problem by looking at some C# code, then we'll cover a Pythonic approach to ISP. If you agree or are just choosing to trust me that super small interfaces are not the best way to segregate your interfaces feel free to skip to the Pythonic Solution below.

```
# Warning here be C# code
public interface ICanUpload {
  void upload(Byte[] file);
}

public interface ICanDownload {
  Byte[] download();
}

class FTPClient : ICanUpload, ICanDownload {
  public void upload(Byte[] file) {
    ...
  }

  public Byte[] download(string target) {
    ...
```

```
  }
}
```

The trouble starts when you need to specify the type of a parameter that implements both the ICanDownload and ICanUpload interfaces. The code snippet below demonstrates the problem.

```
class ReportGenerator {
  public Byte[] doStuff(Byte[] raw) {
    ...
  }

  public void generateReport(/*What type should go here?*/ client) {
    raw_data = client.download('client_rundown.csv');
    report = this.doStuff(raw_data);
    client.upload(report);
  }
}
```

In the `generateReport` function signature you either have to specify the concrete `FTPClient` class as the parameter type, which violates the Dependency Inversion Principle or create an interface that implements both ICanUpload, and ICanDownload interfaces. Otherwise, an object that just implements `ICanUpload` could be passed in but would fail the download call and vice-versa with an object only implementing the `ICanDownload` interface. The normal answer is to create an `IFTPClient` interface and let the `generateReport` function depend on that.

```
public interface IFTPClient: ICanUpload, ICanDownload {
    void upload(Byte[] file);
    Byte[] download(string target);
}
```

That works, except we are still depending on FTP clients. What if we want to start storing our reports in S3?

The Pythonic Solution

To me, ISP is about making reasonable choices for how other developers will interface with your code. That's right it's more related to the I in API and CLI than it is the interface keyword. This is also where the "Readability Counts" from the Zen of Python is a driving force. A good interface will follow the semantics of the abstraction and match the terminology making the code more readable.

Let's look at how we can add an S3Client since it has the same upload/download semantics as the `FTPClients`. We want to keep the `S3Clients` signature for `upload` and `download` consistent, but it would be nonsense for the new `S3Client` to inherit from `FTPClient`. After all, S3 is not a special case of FTP. What FTP and S3 do have in common is that they are file transfer protocols and these protocols often share a similar interface as seen in this example. So instead of inheriting from `FTPClient` it would be better to tie these classes together with an abstract base class, the closest thing Python has to an interface.

We create a `FileTransferClient` which becomes our interface and all of our existing clients now inherit from that rather than inheriting from `FTPClient`. This forces a common interface and allows us to move bulk operations into their own interface since not every file transfer protocol will support them.

```
from abc import ABC
class FileTransferClient(ABC):
  def upload(self, file:bytes):
    pass

  def download(self, target:str) -> bytes:
    pass

  def cd(self, target_dir):
    pass


class BulkFileTransferClient(ABC):
  def upload_bulk(self, files:List[bytes]):
    pass

  def download_bulk(self, targets:List[str]):
    pass
```

What does this afford us though...well this.

```
class FTPClient(FileTransferClient, BulkFileTransferClient):
  ...

class FTPSClient(FileTransferClient, BulkFileTransferClient):
  ...

class SFTPClient(FileTransferClient, BulkFileTransferClient):
  ...

class S3Client(FileTransferClient):
```

```
  ...

class SCPClient(FileTransferClient):
  ...
```

Oh Man! is that good code or what. We even managed to squeeze in a `SCPClient` and kept bulk actions as their own mixin. All this ties together nicely with Dependency Injection, a technique used for the Dependency Inversion Principle.

Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. They should depend on abstractions and abstractions should not depend on details, rather details should depend on abstractions.

Relevant Zen: Explicit is Better than Implicit

This is what ties it all together. Everything we've done with the other SOLID principles was to get to a place where we are no longer dependent on a detail, the underlying file transfer protocol, being used to move files around. We can now write code around our business rules without tying them to a specific implementation. Our code satisfies both requirements of dependency inversion.

Our high-level modules no longer need to depend on a low-level module like FTPClient, SFTPClient, or S3Client, instead, they depend on an abstraction FileTransferClient. We are depending on the abstraction of moving files not the detail of how those files are moved.

Our abstraction FileTransferClient is not dependent on protocol specific details and instead, those details depend on how they will be used through the abstraction (i.e. that files can be uploaded or downloaded).

Here is a example of Dependency Inversion at work.

```
def exchange(client:FileTransferClient, to_upload:bytes, to_download:str) -> by
    client.upload(to_upload)
    return client.download(to_download)

if __name__ == '__main__':
    ftp = FTPClient('ftp.host.com')
    sftp = FTPSClient('sftp.host.com', 22)
    ftps = SFTPClient('ftps.host.com', 990, 'ftps_user', 'P@ssw0rd1!')
    s3 = S3Client('ftp.host.com')
    scp = SCPClient('ftp.host.com')
```

```
for client in [ftp, sftp, ftps, s3, scp]:
    exchange(client, b'Hello', 'greeting.txt')
```

Conclusion

There you have it a SOLID implementation that is also very Pythonic. I'm hoping you've at least warmed up to SOLID if you hadn't before, and for those of you that are learning Python and not sure how to continue writing SOLID code this has been helpful. This, of course, was a curated example that I knew would lend itself to my argument, but in writing this I was still surprised how much changed along the way. Not every problem will fit this exact breakdown, but I've tried to include enough reasoning behind my decisions that you can choose the most SOLID & Pythonic implementation in the future.

If you disagree or need any clarification please leave a comment or @d3r3kdrumm0nd on Twitter.

Discussion (28)

Subscribe

Add to the discussion

Julius • Aug 29 '20

I am not sure if I understand the example used for LSP.

`You may have noticed all of the FTP client classes so far have the same function signatures. That was done purposefully so they would follow Liskov's Substituitability Principle. An SFTPClient object can replace an FTPClient object and whatever code is calling upload, or download, is blissfully unaware.`

SFTPClient requires username and password, whereas FTPClient does not. In the calling code I cannot replace FTPClient(host=host, port=port) with SFTPClient(host=host, port=port), so I do not see how SFTPClient can be a substitute for FTPClient. Could you please explain?

2

Derek D. • Sep 1 '20

Great question! I had to look up Liskov's again to make sure this example was still valid. Fortunately for me, and quite by accident, I think it still works. Let me try to explain.

Liskov's as it originally appeared states, "Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T." So I took this as the principle applies to objects, not classes. In general, I think most people agree. So a class has a constructor which returns an object and as long as the object returned by the constructor can replace an object of the parent type you are following Liskov's.

All that being said with higher-level languages like Python it's probably a better practice to follow Liskov's even with the constructor since we can pass a Class name to a function and then construct an object of that class inside the function. In that scenario what I've written in this article would absolutely blow up.

1

Julius • Sep 1 '20

Thank you for the explanation!

1

Comment deleted

Derek D. • Sep 1 '20

Thanks for pointing it out. I've fixed it.

3

Pavel Morava • Mar 23 '20 • Edited

You did not need to step out from Python for interfaces.

You could use Protocol from typing module. :)

Great article, anyway. Thank you.

1

Derek D. • Mar 23 '20

I didn't know about that module. Thanks for the tip. I look forward to playing around with it.

2

Pavel Morava • Mar 24 '20

You'll need mypy or an internal static typing analyzer (as in PyCharm) to appreciate it. Otherwise, it is ignored in runtime.

1

Bryce Guinta • Sep 6 '20 • Edited

What do you think about dependency injecting the FTPDriver itself so that the IO (establishing connection) isn't in the constructor? Example:

```
class FTPClient:
    def __init__(self, driver):
        self._driver = driver

    @classmethod
    def from_connection(cls, host, port):
        driver = FTPDriver(host, port)
        return cls(driver)
    ...
```

This allows one to install a `TestDriver` for testing `FTPClient`. and bubbles up IO to an outer layer which uncle bob is big on (blog.cleancoder.com/uncle-bob/2012...), but we get to do it in one class because Python supports classmethods.

1

Derek D. • Sep 9 '20

It's an excellent practice. I left it out so the examples were more concrete and demonstrated the principles as simply as possible.

2

Paddy3118 • Jun 3 '20

On single responsibility:
If one class/function/method does X on L or X on M or X on N then it seems the single responsibility could be "do X" rather than splitting it to 3 separate places of L. M. and N handling.
If a lead uses the principle to insist others change code then it could cause friction if it is not convincingly argued *why* **a** view on responsibility is to be preferred.

1

Derek D. • Jun 4 '20

You've hit the nail on the head. I tried to enforce SOLID design in a code base where I was not a team lead. It caused a lot of friction because I didn't explain the principle or benefits well. I hope this article helps with insufficient explanations, but I don't think friction is inherently bad. As a team lead you should be pushing your developers to be even better, but sometimes that means letting them make a mistake and learn from it. The really good team leads will be able to find that balance.

I'm curious about the X on L or X or M, or X on N explanations you offered though. I don't quite follow what you are getting at, but if you have a more concrete example I'd love to discuss it further.

4

Jordan Cahill • Jun 10 '20

I believe that what Paddy3118 meant with the X on L or X or M, or X on N explanations is that sometimes developers may have different opinions on what a single responsibility constitutes. In the end ...

There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

I typically find that after discussing the various possibilities with my team and figuring out the pros and cons of different implementations, the obvious best way will arise.

1 *Thread*

Paddy3118 • Jun 12 '20

Ay, someone else may have thought through wider implications, or not 🤔

1

Gino Mempin • May 22 '20 • Edited

In the last code snippet, in `exchange`, the param and the variable used inside the method don't match. Either `exchanger` should be `client`, or vice-versa.

Still, a great article :)

3

Victor • Jun 4 '20 • Edited

> In the `generateReport` function signature you either have to specify the concrete `FTPClient` class as the parameter type

```
void generateReport<TTransferClient>(TTransferClient client)
  where TTransferClient: ICanUpload, ICanDownload
{
  ...
}
```

Also, I think `IFTPClient` is a violation of SRP in this case. You should simply take in both separately `(ICanUpload uploader, ICanDownload downloader)`.

1

Ram • Sep 12 '20

Appreciate your time, and efforts to teach what you have learnt!
I just have a quick question, how would you approach SOLID design principle for this following problem..

I will send a directory path to your class, your class should perform following steps:
1, connect to the remote path (can be sftp, ftp, local path, or anything else in the future)
2, Reach the directory & read directory contents
3, for each file, detect the format and apply appropriate extraction method for texts.
4, update the DB
5, index in the ES
Finally, send me back the bool signal (success/ fail)

If you could, please share your abstract-level idea, that will be greatly helpful for me easily pick up this SOLID principle stuff. Thanks!

1

Paddy3118 • Jun 3 '20

On the open closed principle
It seems useful on an established large project. But they are only a small subset of Python projects. There is also **"plan to throw one away; you will, anyhow"**, which I use as Python is very good at exploring the solution space, in which OCP may be suppressed.

1

Derek D. • Jun 4 '20

I love that, "plan to throw one away; you will, anyhow". When prototyping you should absolutely be ready to throw code away. I've been caught in the sunk cost fallacy plenty of times when I would have been better throwing things away.

Writing SOLID code is a discipline and the more often you practice it the easier it becomes. The entire FTP example I used for this article was born from just practicing writing SOLID code. I didn't intend for every example to build of the previous but going through the act of writing the code to adhere to each principle revealed an elegant solution I didn't even know was there.

4

YonYonita • Apr 7

Hi Derek and thanks for this article!
You mentioned changing parameter names in the function signature adheres to the OCP. I was wondering how it works when the users of the API which we want to keep 'Closed' for change, explicitly state the parameter names when calling the APIs (for example: calling ftpCilent.upload(file=file_to_be_uploaded). I would assume that in that kind of coding style (which IMHO adds a lot to code readability) OCP may also require that parameter names are kept intact. What is your take on this?

Derek D. • Apr 10

This is a good point. It should state changing positional parameter names is still valid within OCP. Keyword args (kwargs) require the caller knowing the internal variable names of a function so changing those is not valid within OCP. I don't like my calling code knowing what's going on in my functions so I don't use them often which is probably why I missed this scenario.

1

Alex Garel • Jun 3 '20

Hi, thanks, this is a good article. It is interesting to see how SOLID can integrate into the python philosophy.

2

José Coelho • Mar 22 '20

Great article! 👏 It was super easy to follow with the FTP example.
I recently moved from working with Java to Python and it's great to know that you can still apply SOLID principles with Python.

2

Larry Gray • Jun 3 '20

Great article!

2

bshorrosh • Jun 3 '20

How about adding @abc.abstractmethod to the abstract methods, which will enforce the implementation in the concrete class.

1

Derek D. • Jun 4 '20

I debated using `abc` to enforce the implementations and went as far as to explain the Dreaded Diamond problem in C# and Java and how interfaces solve the problem in my first iteration of this article. In the end, I chose not to use `abc` so it wouldn't confuse readers users into thinking `@abc.abstractmethod` were required for ISP.

I personally do use them and would recommend them, especially while teaching the principle to others.

4

Jamie Collins • May 23 '20

Typo in intro to ISP: "complicated is better than complex" - the Zen of Python has it the other way around. Doesn't detract from a great article :)

1

Derek D. • Jun 4 '20

Definitely, something I want to fix though. Thanks for pointing it out.

1

Derek D.

One half of the industry transforming Namespace Podcast. Passionate about Code Craftmanship, Python, and React.js,

Follow

WORK
Software Engineer

LOCATION
Austin, TX

JOINED
11 de mar. de 2020

More from Derek D.

Nobody Likes a DRY PASTRY

#codequality

A Test Driven Approach to Python Packaging

#python #testing #codequality #tutorial

Go v Python A Technical Deep Dive

#go #python