# Implementing an Interface in Python

by William Murphy 41 Comments advanced python

Mark as Completed

#### **Table of Contents**

- Python Interface Overview
- Informal Interfaces
  - Using Metaclasses
  - Using Virtual Base Classes
- Formal Interfaces
  - Using abc.ABCMeta
  - Using .\_\_subclasshook\_\_()
  - Using abc to Register a Virtual Subclass
  - Using Subclass Detection With Registration
  - Using Abstract Method Declaration
- Interfaces in Other Languages
  - Java
  - o C++
  - o Go
- Conclusion

Interfaces play an important role in software engineering. As an application grows, updates and changes to the code base become more difficult to mana classes that look very similar but are unrelated, which can lead to some confusion. In this tutorial, you'll see how you can use a **Python interface** to help current problem.

#### In this tutorial, you'll be able to:

- Understand how interfaces work and the caveats of Python interface creation
- Comprehend how useful interfaces are in a dynamic language like Python
- Implement an informal Python interface
- Use abc. ABCMeta and @abc.abstractmethod to implement a formal Python interface

Interfaces in Python are handled differently than in most other languages, and they can vary in their design complexity. By the end of this tutorial, you'll he Python's data model, as well as how interfaces in Python compare to those in languages like Java, C++, and Go.

Free Bonus: 5 Thoughts On Python Mastery, a free course for Python developers that shows you the roadmap and the mindset you'll need to take y

## **Python Interface Overview**

At a high level, an interface acts as a **blueprint** for designing classes. Like classes, interfaces define methods. Unlike classes, these methods are abstract. A simply defines. It doesn't implement the methods. This is done by classes, which then **implement** the interface and give concrete meaning to the interface

Python's approach to interface design is somewhat different when compared to languages like Java, Go, and C++. These languages all have an interface deviates from other languages in one other aspect. It doesn't require the class that's implementing the interface to define all of the interface's abstract m

Remove ads

## **Informal Interfaces**

In certain circumstances, you may not need the strict rules of a formal Python interface. Python's dynamic nature allows you to implement an **informal ir** defines methods that can be overridden, but there's no strict enforcement.

In the following example, you'll take the perspective of a data engineer who needs to extract text from various different unstructured file types, like PDFs defines the methods that will be in both the PdfParser and EmlParser concrete classes:

The second concrete class is EmlParser, which you'll use to parse the text from emails:

c1

< ▶

The concrete implementation of InformalParserInterface now allows you to extract text from email files.

So far, you've defined two **concrete implementations** of the InformalPythonInterface. However, note that EmlParser fails to properly define .extract\_text whether EmlParser implements InformalParserInterface, then you'd get the following result:

>>

>>

11.

>> Tr

< ▶

This would return True, which poses a bit of a problem since it violates the definition of an interface!

Now check the **method resolution order (MRO)** of PdfParser and EmlParser. This tells you the superclasses of the class in question, as well as the order in You can view a class's MRO by using the dunder method cls.\_mro\_:

>>

(\_

>>

(\_

Such informal interfaces are fine for small projects where only a few developers are working on the source code. However, as projects get larger and tean countless hours looking for hard-to-find logic errors in the codebase!

## **Using Metaclasses**

Ideally, you would want issubclass(EmlParser, InformalParserInterface) to return False when the implementing class doesn't define all of the interface's a metaclass called ParserMeta. You'll be overriding two dunder methods:

1. .\_\_instancecheck\_\_()

2. .\_\_subclasscheck\_\_()

In the code block below, you create a class called UpdatedInformalParserInterface that builds from the ParserMeta metaclass:

c1

cl



Here, you have a metaclass that's used to create UpdatedInformalParserInterface. By using a metaclass, you don't need to explicitly define the subclasses. **methods**. If it doesn't, then issubclass(EmlParserNew, UpdatedInformalParserInterface) will return False.

Running issubclass() on your concrete classes will produce the following:

>> Tr

Tr

>> Fa

< →

 $As \ expected, \ {\tt EmlParserNew} \ is \ not \ a \ subclass \ of \ {\tt UpdatedInformalParserInterface} \ since \ . \\ {\tt extract\_text()} \ wasn't \ defined \ in \ {\tt EmlParserNew}.$ 

Now, let's have a look at the MRO:

>>

(<

< ▶

As you can see, UpdatedInformalParserInterface is a superclass of PdfParserNew, but it doesn't appear in the MRO. This unusual behavior is caused by the fabraser class of PdfParserNew.

Remove ads

## **Using Virtual Base Classes**

In the previous example, issubclass(EmlParserNew, UpdatedInformalParserInterface) returned True, even though UpdatedInformalParserInterface did not ap because UpdatedInformalParserInterface is a **virtual base class** of EmlParserNew.

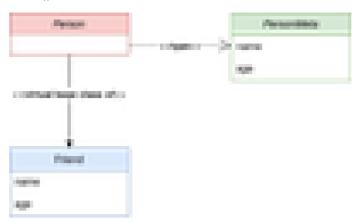
The key difference between these and standard subclasses is that virtual base classes use the .\_\_subclasscheck\_\_() dunder method to implicitly check if a Additionally, virtual base classes don't appear in the subclass MRO.

Take a look at this code block:

c1

c1

The following UML diagram shows what happens when you call issubclass() on the Friend class:



Taking a look at PersonMeta, you'll notice that there's another dunder method called .\_\_instancecheck\_\_(). This method is used to check if instances of Fri will call .\_\_instancecheck\_\_() when you use isinstance(Friend, Person).

### **Formal Interfaces**

Informal interfaces can be useful for projects with a small code base and a limited number of programmers. However, informal interfaces would be the w create a **formal Python interface**, you'll need a few more tools from Python's abc module.

### Using abc.ABCMeta

To enforce the subclass instantiation of abstract methods, you'll utilize Python's builtin ABCMeta from the abc module. Going back to your UpdatedInformalP metaclass, ParserMeta, with the overridden dunder methods .\_\_instancecheck\_\_() and .\_\_subclasscheck\_\_().

Rather than create your own metaclass, you'll use abc.ABCMeta as the metaclass. Then, you'll overwrite .\_\_subclasshook\_\_() in place of .\_\_instancecheck\_\_() reliable implementation of these dunder methods.

### Using .\_\_subclasshook\_\_()

Here's the implementation of FormalParserInterface using abc.ABCMeta as your metaclass:

im

c1

cl

cl



The decorator register method helps you to create a hierarchy of custom virtual class inheritance.

### **Using Subclass Detection With Registration**

You must be careful when you're combining .\_\_subclasshook\_\_() with .register(), as .\_\_subclasshook\_\_() takes precedence over virtual subclass registrati are taken into consideration, you must add NotImplemented to the .\_\_subclasshook\_\_() dunder method. The FormalParserInterface would be updated to the

c1

cl

@⊦ cl

pr pr

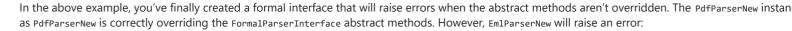
Since you've used registration, you can see that EmlParserNew is considered a virtual subclass of your FormalParserInterface interface. This is not what you override .extract\_text(). Please use caution with virtual subclass registration!

## **Using Abstract Method Declaration**

An **abstract method** is a method that's declared by the Python interface, but it may not have a useful implementation. The abstract method must be over interface in question.

To create abstract methods in Python, you add the @abc.abstractmethod decorator to the interface's methods. In the next example, you update the Formal methods .load\_data\_source() and .extract\_text():

c1



>> >> Tr

As you can see, the traceback message tells you that you haven't overridden all the abstract methods. This is the behavior you expect when building a for

# **Interfaces in Other Languages**

Interfaces appear in many programming languages, and their implementation varies greatly from language to language. In the next few sections, you'll co

#### Java

Unlike Python, Java contains an interface keyword. Keeping with the file parser example, you declare an interface in Java like so:

pu }

Now you'll create two concrete classes, PdfParser and EmlParser, to implement the FileParserInterface. To do so, you must use the implements keyword in

pu

}

Continuing with your file parsing example, a fully-functional Java interface would look something like this:

im im

pu

cl
};
cl

A Python interface and a C++ interface have some similarities in that they both make use of abstract base classes to simulate interfaces.

### Go

pa

Although Go's syntax is reminiscent of Python, the Go programming language contains an interface keyword, like Java. Let's create the fileParserInterface

ty } **♦** 

A big difference between Python and Go is that Go doesn't have classes. Rather, Go is similar to C in that it uses the struct keyword to create structures. Contains data and methods. However, unlike a class, all of the data and methods are publicly accessed. The concrete structs in Go will be used to implement the contains data and methods.

Here's an example of how Go uses interfaces:

Unlike a Python interface, a Go interface is created using structs and the explicit keyword interface.

Remove ads

}