Table of Contents

* [Python Interface Overview](https://realpython.com/python-interface/#python-interface-overview)
* [Informal Interfaces](https://realpython.com/python-interface/#informal-interfaces)
  + [Using Metaclasses](https://realpython.com/python-interface/#using-metaclasses)
  + [Using Virtual Base Classes](https://realpython.com/python-interface/#using-virtual-base-classes)
* [Formal Interfaces](https://realpython.com/python-interface/#formal-interfaces)
  + [Using abc.ABCMeta](https://realpython.com/python-interface/#using-abcabcmeta)
  + [Using .\_\_subclasshook\_\_()](https://realpython.com/python-interface/#using-__subclasshook__)
  + [Using abc to Register a Virtual Subclass](https://realpython.com/python-interface/#using-abc-to-register-a-virtual-subclass)
  + [Using Subclass Detection With Registration](https://realpython.com/python-interface/#using-subclass-detection-with-registration)
  + [Using Abstract Method Declaration](https://realpython.com/python-interface/#using-abstract-method-declaration)
* [Interfaces in Other Languages](https://realpython.com/python-interface/#interfaces-in-other-languages)
  + [Java](https://realpython.com/python-interface/#java)
  + [C++](https://realpython.com/python-interface/#c)
  + [Go](https://realpython.com/python-interface/#go)
* [Conclusion](https://realpython.com/python-interface/#conclusion)

Interfaces play an important role in software engineering. As an application grows, updates and changes to the code base become more difficult to manage. More often than not, you wind up having 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 determine what class you should use to tackle the 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 have a better understanding of some aspects of 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**](https://realpython.com/bonus/python-mastery-course/), a free course for Python developers that shows you the roadmap and the mindset you’ll need to take your Python skills to the next level.

**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. An **abstract method** is one that the interface 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’s abstract methods.

Python’s approach to interface design is somewhat different when compared to languages like [Java](https://realpython.com/oop-in-python-vs-java/), Go, and [C++](https://realpython.com/python-vs-cpp/). These languages all have an interface keyword, while Python does not. Python further 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 methods.

**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 interface**. An informal Python interface is a class that 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](https://realpython.com/python-data-engineer/) who needs to extract text from various different unstructured file types, like PDFs and emails. You’ll create an informal interface that defines the methods that will be in both the PdfParser and EmlParser concrete classes:

class InformalParserInterface:

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Load in the file for extracting text."""

pass

def extract\_text(self, full\_file\_name: str) -> dict:

"""Extract text from the currently loaded file."""

pass

InformalParserInterface defines the two methods .load\_data\_source() and .extract\_text(). These methods are defined but not implemented. The implementation will occur once you create **concrete classes** that inherit from InformalParserInterface.

As you can see, InformalParserInterface looks identical to a standard [Python class](https://realpython.com/python-classes/). You rely on [duck typing](https://realpython.com/lessons/duck-typing/) to inform users that this is an interface and should be used accordingly.

**Note:** Haven’t heard of **duck typing**? This term says that if you have an object that looks like a duck, walks like a duck, and quacks like a duck, then it must be a duck! To learn more, check out [Duck Typing](https://realpython.com/lessons/duck-typing/).

With duck typing in mind, you define two classes that implement the InformalParserInterface. To use your interface, you must create a concrete class. A **concrete class** is a subclass of the interface that provides an implementation of the interface’s methods. You’ll create two concrete classes to implement your interface. The first is PdfParser, which you’ll use to parse the text from [PDF](https://realpython.com/creating-modifying-pdf/) files:

class PdfParser(InformalParserInterface):

"""Extract text from a PDF"""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides InformalParserInterface.load\_data\_source()"""

pass

def extract\_text(self, full\_file\_path: str) -> dict:

"""Overrides InformalParserInterface.extract\_text()"""

pass

The concrete implementation of InformalParserInterface now allows you to extract text from [PDF](https://realpython.com/pdf-python/) files.

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

class EmlParser(InformalParserInterface):

"""Extract text from an email"""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides InformalParserInterface.load\_data\_source()"""

pass

def extract\_text\_from\_email(self, full\_file\_path: str) -> dict:

"""A method defined only in EmlParser.

Does not override InformalParserInterface.extract\_text()

"""

pass

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(). If you were to check whether EmlParser implements InformalParserInterface, then you’d get the following result:

>>>

>>> # Check if both PdfParser and EmlParser implement InformalParserInterface

>>> issubclass(PdfParser, InformalParserInterface)

True

>>> issubclass(EmlParser, InformalParserInterface)

True

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 which they’re searched for executing a method. You can view a class’s MRO by using the dunder method cls.\_\_mro\_\_:

>>>

>>> PdfParser.\_\_mro\_\_

(\_\_main\_\_.PdfParser, \_\_main\_\_.InformalParserInterface, object)

>>> EmlParser.\_\_mro\_\_

(\_\_main\_\_.EmlParser, \_\_main\_\_.InformalParserInterface, object)

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 teams grow, this could lead to developers spending 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 abstract methods. To do this, you’ll create a [metaclass](https://realpython.com/python-metaclasses/) called ParserMeta. You’ll be overriding two [dunder](https://dbader.org/blog/python-dunder-methods) methods:

1. .\_\_instancecheck\_\_()
2. .\_\_subclasscheck\_\_()

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

class ParserMeta(type):

"""A Parser metaclass that will be used for parser class creation.

"""

def \_\_instancecheck\_\_(cls, instance):

return cls.\_\_subclasscheck\_\_(type(instance))

def \_\_subclasscheck\_\_(cls, subclass):

return (hasattr(subclass, 'load\_data\_source') and

callable(subclass.load\_data\_source) and

hasattr(subclass, 'extract\_text') and

callable(subclass.extract\_text))

class UpdatedInformalParserInterface(metaclass=ParserMeta):

"""This interface is used for concrete classes to inherit from.

There is no need to define the ParserMeta methods as any class

as they are implicitly made available via .\_\_subclasscheck\_\_().

"""

pass

Now that ParserMeta and UpdatedInformalParserInterface have been created, you can create your concrete implementations.

First, create a new class for parsing PDFs called PdfParserNew:

class PdfParserNew:

"""Extract text from a PDF."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides UpdatedInformalParserInterface.load\_data\_source()"""

pass

def extract\_text(self, full\_file\_path: str) -> dict:

"""Overrides UpdatedInformalParserInterface.extract\_text()"""

pass

Here, PdfParserNew overrides .load\_data\_source() and .extract\_text(), so issubclass(PdfParserNew, UpdatedInformalParserInterface) should return True.

In this next code block, you have a new implementation of the email parser called EmlParserNew:

class EmlParserNew:

"""Extract text from an email."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides UpdatedInformalParserInterface.load\_data\_source()"""

pass

def extract\_text\_from\_email(self, full\_file\_path: str) -> dict:

"""A method defined only in EmlParser.

Does not override UpdatedInformalParserInterface.extract\_text()

"""

pass

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

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

>>>

>>> issubclass(PdfParserNew, UpdatedInformalParserInterface)

True

>>> issubclass(EmlParserNew, UpdatedInformalParserInterface)

False

As expected, EmlParserNew is not a subclass of UpdatedInformalParserInterface since .extract\_text() wasn’t defined in EmlParserNew.

Now, let’s have a look at the MRO:

>>>

>>> PdfParserNew.\_\_mro\_\_

(<class '\_\_main\_\_.PdfParserNew'>, <class 'object'>)

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 fact that UpdatedInformalParserInterface is a **virtual base class** of PdfParserNew.

[](https://srv.realpython.net/click/27944979224/?c=48874902176&p=58946116052&r=98164)

[Remove ads](https://realpython.com/account/join/)

Using Virtual Base Classes

In the previous example, issubclass(EmlParserNew, UpdatedInformalParserInterface) returned True, even though UpdatedInformalParserInterface did not appear in the EmlParserNew MRO. That’s 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 class is a virtual subclass of the superclass. Additionally, virtual base classes don’t appear in the subclass MRO.

Take a look at this code block:

class PersonMeta(type):

"""A person metaclass"""

def \_\_instancecheck\_\_(cls, instance):

return cls.\_\_subclasscheck\_\_(type(instance))

def \_\_subclasscheck\_\_(cls, subclass):

return (hasattr(subclass, 'name') and

callable(subclass.name) and

hasattr(subclass, 'age') and

callable(subclass.age))

class PersonSuper:

"""A person superclass"""

def name(self) -> str:

pass

def age(self) -> int:

pass

class Person(metaclass=PersonMeta):

"""Person interface built from PersonMeta metaclass."""

pass

Here, you have the setup for creating your virtual base classes:

1. The metaclass PersonMeta
2. The base class PersonSuper
3. The Python interface Person

Now that the setup for creating **virtual base classes** is done you’ll define two concrete classes, Employee and Friend. The Employee class inherits from PersonSuper, while Friend implicitly inherits from Person:

# Inheriting subclasses

class Employee(PersonSuper):

"""Inherits from PersonSuper

PersonSuper will appear in Employee.\_\_mro\_\_

"""

pass

class Friend:

"""Built implicitly from Person

Friend is a virtual subclass of Person since

both required methods exist.

Person not in Friend.\_\_mro\_\_

"""

def name(self):

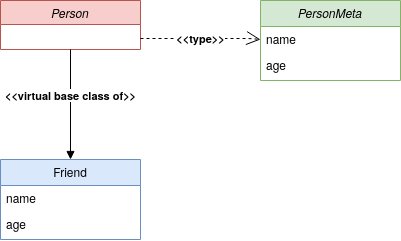
pass

def age(self):

pass

Although Friend does not explicitly inherit from Person, it implements .name() and .age(), so Person becomes a **virtual base class** of Friend. When you run issubclass(Friend, Person) it should return True, meaning that Friend is a subclass of Person.

The following [**UML**](https://realpython.com/inheritance-composition-python/#whats-inheritance) diagram shows what happens when you call issubclass() on the Friend class:

[](https://files.realpython.com/media/virtual-base-class.b545144aafef.png)

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 Friend are created from the Person interface. Your code 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 wrong approach for larger applications. In order to 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](https://docs.python.org/3/library/abc.html) module. Going back to your UpdatedInformalParserInterface interface, you created your own 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\_\_() and .\_\_subclasscheck\_\_(), as it creates a more reliable implementation of these dunder methods.

Using .\_\_subclasshook\_\_()

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

import abc

class FormalParserInterface(metaclass=abc.ABCMeta):

@classmethod

def \_\_subclasshook\_\_(cls, subclass):

return (hasattr(subclass, 'load\_data\_source') and

callable(subclass.load\_data\_source) and

hasattr(subclass, 'extract\_text') and

callable(subclass.extract\_text))

class PdfParserNew:

"""Extract text from a PDF."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text(self, full\_file\_path: str) -> dict:

"""Overrides FormalParserInterface.extract\_text()"""

pass

class EmlParserNew:

"""Extract text from an email."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text\_from\_email(self, full\_file\_path: str) -> dict:

"""A method defined only in EmlParser.

Does not override FormalParserInterface.extract\_text()

"""

pass

If you run issubclass() on PdfParserNew and EmlParserNew, then issubclass() will return True and False, respectively.

[](https://srv.realpython.net/click/16642766600/?c=48874902176&p=58946116052&r=85004)

[Remove ads](https://realpython.com/account/join/)

Using abc to Register a Virtual Subclass

Once you’ve imported the abc module, you can directly **register a virtual subclass** by using the .register() metamethod. In the next example, you register the interface Double as a virtual base class of the built-in \_\_float\_\_ class:

class Double(metaclass=abc.ABCMeta):

"""Double precision floating point number."""

pass

Double.register(float)

You can check out the effect of using .register():

>>>

>>> issubclass(float, Double)

True

>>> isinstance(1.2345, Double)

True

By using the .register() meta method, you’ve successfully registered Double as a virtual subclass of float.

Once you’ve registered Double, you can use it as class [decorator](https://realpython.com/courses/python-decorators-101/) to set the decorated class as a virtual subclass:

@Double.register

class Double64:

"""A 64-bit double-precision floating-point number."""

pass

print(issubclass(Double64, Double)) # True

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 registration. To ensure that the registered virtual subclasses are taken into consideration, you must add NotImplemented to the .\_\_subclasshook\_\_() dunder method. The FormalParserInterface would be updated to the following:

class FormalParserInterface(metaclass=abc.ABCMeta):

@classmethod

def \_\_subclasshook\_\_(cls, subclass):

return (hasattr(subclass, 'load\_data\_source') and

callable(subclass.load\_data\_source) and

hasattr(subclass, 'extract\_text') and

callable(subclass.extract\_text) or

NotImplemented)

class PdfParserNew:

"""Extract text from a PDF."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text(self, full\_file\_path: str) -> dict:

"""Overrides FormalParserInterface.extract\_text()"""

pass

@FormalParserInterface.register

class EmlParserNew:

"""Extract text from an email."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text\_from\_email(self, full\_file\_path: str) -> dict:

"""A method defined only in EmlParser.

Does not override FormalParserInterface.extract\_text()

"""

pass

print(issubclass(PdfParserNew, FormalParserInterface)) # True

print(issubclass(EmlParserNew, FormalParserInterface)) # True

Since you’ve used registration, you can see that EmlParserNew is considered a virtual subclass of your FormalParserInterface interface. This is not what you wanted since EmlParserNew doesn’t 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 overridden by the concrete class that implements the 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 FormalParserInterface to include the abstract methods .load\_data\_source() and .extract\_text():

class FormalParserInterface(metaclass=abc.ABCMeta):

@classmethod

def \_\_subclasshook\_\_(cls, subclass):

return (hasattr(subclass, 'load\_data\_source') and

callable(subclass.load\_data\_source) and

hasattr(subclass, 'extract\_text') and

callable(subclass.extract\_text) or

NotImplemented)

@abc.abstractmethod

def load\_data\_source(self, path: str, file\_name: str):

"""Load in the data set"""

raise NotImplementedError

@abc.abstractmethod

def extract\_text(self, full\_file\_path: str):

"""Extract text from the data set"""

raise NotImplementedError

class PdfParserNew(FormalParserInterface):

"""Extract text from a PDF."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text(self, full\_file\_path: str) -> dict:

"""Overrides FormalParserInterface.extract\_text()"""

pass

class EmlParserNew(FormalParserInterface):

"""Extract text from an email."""

def load\_data\_source(self, path: str, file\_name: str) -> str:

"""Overrides FormalParserInterface.load\_data\_source()"""

pass

def extract\_text\_from\_email(self, full\_file\_path: str) -> dict:

"""A method defined only in EmlParser.

Does not override FormalParserInterface.extract\_text()

"""

pass

In the above example, you’ve finally created a formal interface that will raise errors when the abstract methods aren’t overridden. The PdfParserNew instance, pdf\_parser, won’t raise any errors, as PdfParserNew is correctly overriding the FormalParserInterface abstract methods. However, EmlParserNew will raise an error:

>>>

>>> pdf\_parser = PdfParserNew()

>>> eml\_parser = EmlParserNew()

Traceback (most recent call last):

File "real\_python\_interfaces.py", line 53, in <module>

eml\_interface = EmlParserNew()

TypeError: Can't instantiate abstract class EmlParserNew with abstract methods extract\_text

As you can see, the [traceback](https://realpython.com/python-traceback/) message tells you that you haven’t overridden all the abstract methods. This is the behavior you expect when building a formal Python interface.

[](https://srv.realpython.net/click/32532819624/?c=48874902176&p=58946116052&r=09627)

[Remove ads](https://realpython.com/account/join/)

**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 compare interfaces in Python to Java, C++, and Go.

Java

Unlike Python, [Java](https://realpython.com/oop-in-python-vs-java/) contains an interface keyword. Keeping with the file parser example, you declare an interface in Java like so:

public interface FileParserInterface {

// Static fields, and abstract methods go here ...

public void loadDataSource();

public void extractText();

}

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

public class EmlParser implements FileParserInterface {

public void loadDataSource() {

// Code to load the data set

}

public void extractText() {

// Code to extract the text

}

}

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

import java.util.\*;

import java.io.\*;

public class FileParser {

public static void main(String[] args) throws IOException {

// The main entry point

}

public interface FileParserInterface {

HashMap<String, ArrayList<String>> file\_contents = null;

public void loadDataSource();

public void extractText();

}

public class PdfParser implements FileParserInterface {

public void loadDataSource() {

// Code to load the data set

}

public void extractText() {

// Code to extract the text

}

}

public class EmlParser implements FileParserInterface {

public void loadDataSource() {

// Code to load the data set

}

public void extractText() {

// Code to extract the text

}

}

}

As you can see, a Python interface gives you much more flexibility during creation than a Java interface does.

C++

Like Python, C++ uses abstract base classes to create interfaces. When defining an interface in C++, you use the keyword virtual to describe a method that should be overwritten in the concrete class:

class FileParserInterface {

public:

virtual void loadDataSource(std::string path, std::string file\_name);

virtual void extractText(std::string full\_file\_name);

};

When you want to implement the interface, you’ll give the concrete class name, followed by a colon (:), and then the name of the interface. The following example demonstrates C++ interface implementation:

class PdfParser : FileParserInterface {

public:

void loadDataSource(std::string path, std::string file\_name);

void extractText(std::string full\_file\_name);

};

class EmlParser : FileParserInterface {

public:

void loadDataSource(std::string path, std::string file\_name);

void extractText(std::string full\_file\_name);

};

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

Go

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

type fileParserInterface interface {

loadDataSet(path string, filename string)

extractText(full\_file\_path string)

}

A big difference between Python and Go is that Go doesn’t have classes. Rather, Go is similar to [C](https://realpython.com/build-python-c-extension-module/) in that it uses the struct keyword to create structures. A **structure** is similar to a class in that a structure 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 fileParserInterface.

Here’s an example of how Go uses interfaces:

package main

type fileParserInterface interface {

loadDataSet(path string, filename string)

extractText(full\_file\_path string)

}

type pdfParser struct {

// Data goes here ...

}

type emlParser struct {

// Data goes here ...

}

func (p pdfParser) loadDataSet() {

// Method definition ...

}

func (p pdfParser) extractText() {

// Method definition ...

}

func (e emlParser) loadDataSet() {

// Method definition ...

}

func (e emlParser) extractText() {

// Method definition ...

}

func main() {

// Main entrypoint

}

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

[](https://srv.realpython.net/click/33911031100/?c=48874902176&p=58946116052&r=59082)

[Remove ads](https://realpython.com/account/join/)

**Conclusion**

Python offers great flexibility when you’re creating interfaces. An informal Python interface is useful for small projects where you’re less likely to get confused as to what the return types of the methods are. As a project grows, the need for a **formal Python interface** becomes more important as it becomes more difficult to infer return types. This ensures that the concrete class, which implements the interface, overwrites the abstract methods.

**Now you can:**

* Understand **how interfaces work** and the caveats of creating a Python interface
* Understand the **usefulness** of interfaces in a dynamic language like Python
* Implement **formal and informal** interfaces in Python
* **Compare Python interfaces** to those in languages like Java, C++, and Go

Now that you’ve become familiar with how to create a Python interface, add a Python interface to your next project to see its usefulness in action!

https://www.pythontutorial.net/python-oop/python-abstract-class/