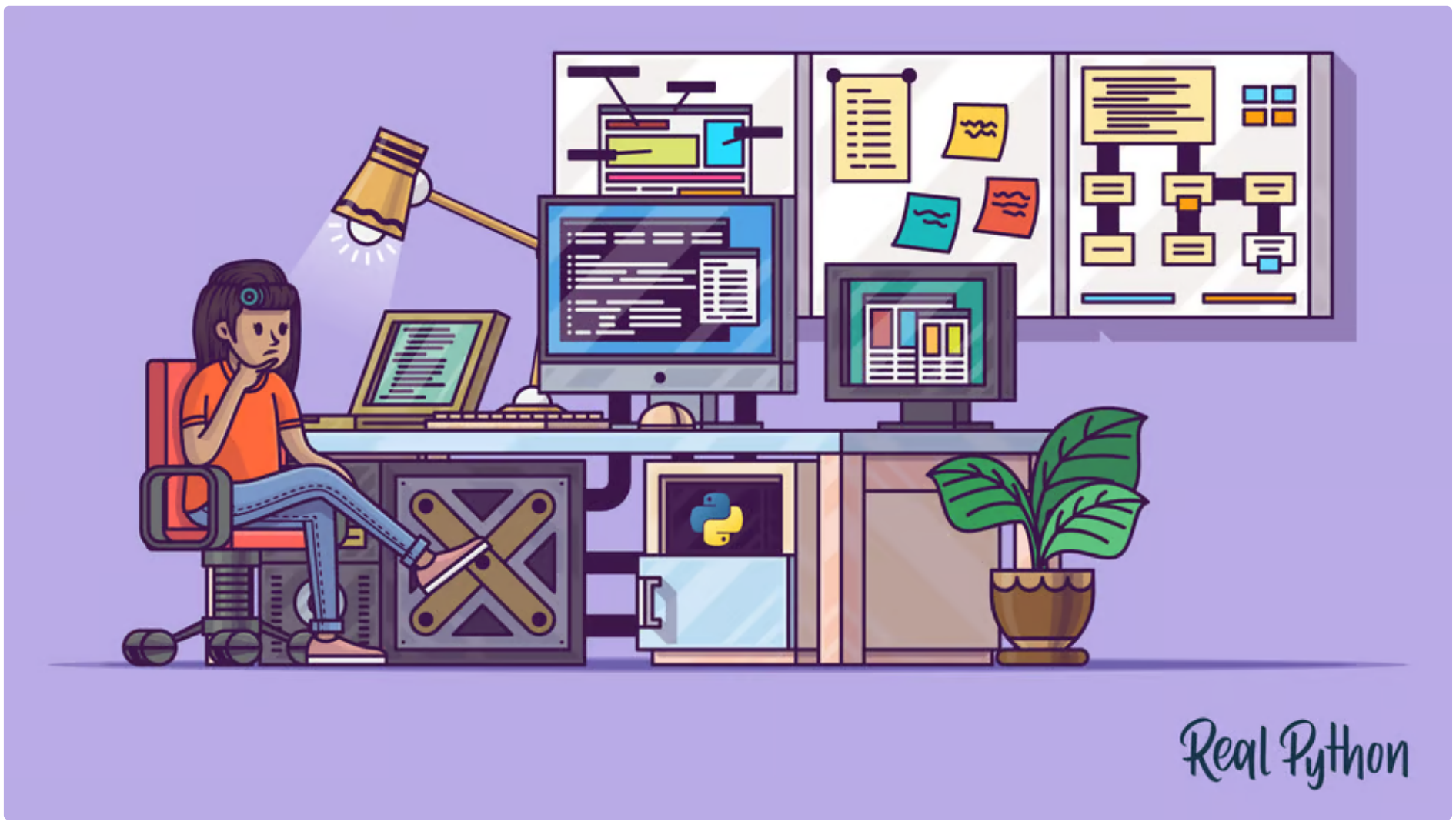


Implementing an Interface in Python

by William Murphy Feb 10, 2020 51 Comments advanced python

Mark as Completed

Share Share Email

Table of Contents

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.

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, Go, and C++. 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 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:

Python

```
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. You rely on 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.

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 files:

Python

```
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 files.

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

Python

```
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:

Python

```
>>> # 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__`:

Python

```
>>> 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 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:

Python

```
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`:

Python

```
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`:

Python

```
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:

Python

```
>>> 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:

Python

```
>>> 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`.

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:

Python

```
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`:

Python

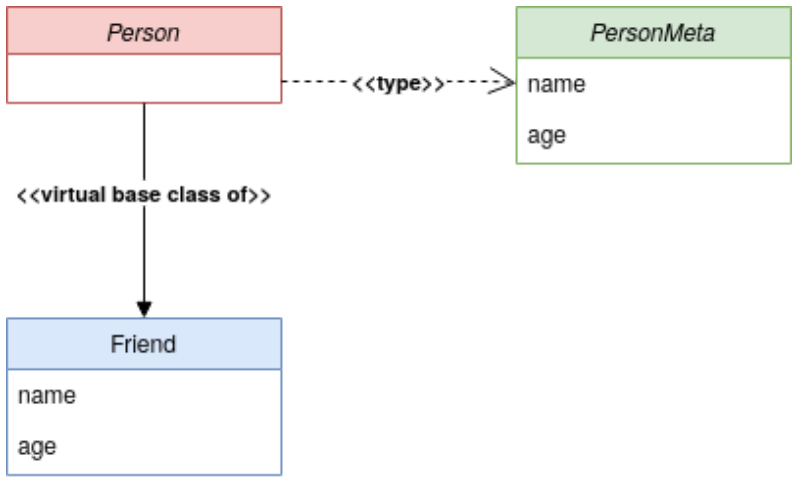
```
# Inheriting subclasses
class Employee(PersonSuper):
    """Inherits from PersonSuper
    PersonSuper will appear in Employee.__mro__
    """
    pass

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** 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 `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` 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:

Python

```
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.

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:

Python

```
class Double(metaclass=abc.ABCMeta):
    """Double precision floating point number."""
    pass

Double.register(float)
```

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

Python

```
>>> 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 to set the decorated class as a virtual subclass:

Python

```
@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:

Python

```
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()`:

Python

```python
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:

Python

```
>>> 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 message tells you that you haven't overridden all the abstract methods. This is the behavior you expect when building a formal Python interface.

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

Java

```
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:

Java

```
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:

Java

```
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:

C++

```
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:

C++

```
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:

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 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:

Go

```
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`.

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!