2.6 Abstract classes

Python is considered to be a very flexible programming language, but that doesn't mean that there are no controls to impose a set of functionalities or an order in a class hierarchy. When you develop a system in a group of programmers, it would be useful to have some means of establishing requirements for classes in matters of interfaces (methods) exposed by each class.

What is an abstract class?

An **abstract class** should be considered a blueprint for other classes, a kind of contract between a class designer and a programmer:

- the class designer sets requirements regarding methods that must be implemented by just declaring them, but not defining them in detail. Such methods are called **abstract methods**.
- The programmer has to deliver all method definitions and the completeness would be validated by another, dedicated module. The programmer delivers the method definitions by overriding the method declarations received from the class designer.

This contract assures you that a child class, built upon your abstract class, will be equipped with a set of concrete methods imposed by the abstract class.

Why do we want to use abstract classes?

The very important reason is: we want our code to be polymorphic, so all subclasses have to deliver a set of their own method implementations in order to call them by using common method names.

Furthermore, a class which contains one or more abstract methods is called an abstract class. This means that abstract classes are not limited to containing only abstract methods – some of the methods can already be defined, but if any of the methods is an abstract one, then the class becomes abstract.

What is an abstract method?

An abstract method is a method that has a declaration, but does not have any implementation. We'll give some examples of such methods to emphasize their abstract nature.

Let's talk about an example:

Assume that you're designing a music player application, intended to support multiple file formats. Some of the formats are known now, but some are not yet known. The idea is to design an abstract class representing a base music format and corresponding methods for "open", "play", "get details", "rewind", etc., to maintain polymorphism.

Your team should implement concrete classes for each format you'd like to support. Whenever new format specifications become available, you won't have to rework your music player code, you'll just have to deliver a class supporting the new file format, fulfilling the contract imposed by the abstract class.

Remember that it isn't possible to instantiate an abstract class, and it needs subclasses to provide implementations for those abstract methods which are declared in the abstract classes. This behavior is a test performed by a dedicated Python module to validate if the developer has implemented a subclass that overrides all abstract methods.

When we're designing large functional units, in the form of classes, we should use an abstract class. When we want to provide common implemented functionality for all implementations of the class, we could also use an abstract class, because abstract classes partially allow us to implement classes by delivering concrete definitions for some of the methods, not only declarations.

We have just defined the means by which to provide a common Application Program Interface (API) for a set of subclasses. This capability is especially useful in situations where your team or third-party is going to provide implementations, such as with plugins in an application, even after the main application development is finished.

Let's start with a typical class that can be instantiated:

```
class BluePrint:
    def hello(self):
        print('Nothing is blue unless you need it')
bp = BluePrint()
bp.hello()
```

There's nothing new for you here – it's just an example of creating a class and instantiating it. No errors, no doubts, and it gives a clear output:

output

```
Nothing is blue unless you need it
```

Python has come up with a module which provides the helper class for defining Abstract Base Classes (ABC) and that module name is **abc**.

The ABC allows you to mark classes as abstract ones and distinguish which methods of the base abstract class are abstract. A method becomes abstract by being decorated with an `@abstractmethod` decorator.

To start with ABC you should:

- import the `abc` module;
- make your base class inherit the helper class ABC, which is delivered by the `abc` module;
- decorate abstract methods with `@abstractmethod`, which is delivered by the abc module.
- When you run the code, the output doesn't surprise anyone:

```
import abc

class BluePrint(abc.ABC):
    @abc.abstractmethod
    def hello(self):
        pass

class GreenField(BluePrint):
    def hello(self):
        print('Welcome to Green Field!')


gf = GreenField()
gf.hello()
```

output

```
Welcome to Green Field!
```

So far everything works fine, so let's play with the code a little.

Now we'll try to instantiate the BluePrint class:

```
import abc

class BluePrint(abc.ABC):
    @abc.abstractmethod
    def hello(self):
        pass

class GreenField(BluePrint):
    def hello(self):
        print('Welcome to Green Field!')
gf = GreenField()
gf.hello()

bp = BluePrint()
```

The output:

output

```
Welcome to Green Field!
Traceback (most recent call last):
(...)
    bp = BluePrint()
TypeError: Can't instantiate abstract class BluePrint with abstract methods
hello
```

indicates that:

- it's possible to instantiate the `GreenField` class and call the `hello` method, because the Python developer has provided a concrete definition of the `hello` method. In other words, the Python developer has overridden the abstract method `hello` with their own implementation. When the base class provides more abstract methods, all of them must be overridden in a subclass before the subclass can be instantiated.
- Python raises a `TypeError` exception when we try to instantiate the base `BluePrint` class, because it contains an abstract method.

Now we'll try to inherit the abstract class and forget about overriding the abstract method by creating a `RedField` class that does not override the `hello` method.

```
import abc

class BluePrint(abc.ABC):
    @abc.abstractmethod
    def hello(self):
        pass

class GreenField(BluePrint):
    def hello(self):
```

```
        print('Welcome to Green Field!')

class RedField(BluePrint):
    def yellow(self):
        pass

gf = GreenField()
gf.hello()

rf = RedField()
```

The output:

output

```
Welcome to Green Field!
Traceback (most recent call last):
(...)
    rf = RedField()
TypeError: Can't instantiate abstract class RedField with abstract methods
hello
```

indicates that:

- it's possible to instantiate the `GreenField` class and call the `hello` method;
- the `RedField` class is still recognized as an abstract one, because it inherits all elements of its super class, which is abstract, and the RedField class does not override the abstract `hello` method.

Multiple inheritance When you plan to implement a multiple inheritance from abstract classes, remember that an effective subclass should override all abstract methods inherited from its super classes.

Summary:

- Abstract Base Class (ABC) is a class that cannot be instantiated. Such a class is a base class for concrete classes;
- ABC can only be inherited from;
- we are forced to override all abstract methods by delivering concrete method implementations.

A note:

It's tempting to call a module "abc" and then try to import it, but by doing so Python imports the module containing the ABC class instead of your local file. This could cause some confusion – why does such a common name as "abc" conflict with my simple module "abc"?

Run your own experiment to become familiar with the error messages you would encounter in such a situation.

LAB

Objectives

- Creation of abstract classes and abstract methods;
- multiple inheritance of abstract classes;
- overriding abstract methods;
- delivering multiple child classes.

Scenario

- You are about to create a multifunction device (MFD) that can scan and print documents;
- the system consists of a scanner and a printer;
- your task is to create blueprints for it and deliver the implementations;
- create an abstract class representing a scanner that enforces the following methods:
    - **scan_document** – returns a string indicating that the document has been scanned;
    - **get_scanner_status** – returns information about the scanner (max. resolution, serial number)
- Create an abstract class representing a printer that enforces the following methods:
    - **print_document** – returns a string indicating that the document has been printed;
    - **get_printer_status** – returns information about the printer (max. resolution, serial number)
- Create MFD1, MFD2 and MFD3 classes that inherit the abstract classes responsible for scanning and printing:
    - MFD1 – should be a cheap device, made of a cheap printer and a cheap scanner, so device capabilities (resolution) should be low;
    - MFD2 – should be a medium-priced device allowing additional operations like printing operation history, and the resolution is better than the lower-priced device;
    - MFD3 – should be a premium device allowing additional operations like printing operation history and fax machine.
- Instantiate MFD1, MFD2 and MFD3 to demonstrate their abilities. All devices should be capable of serving generic feature sets.

resposta

; resposta1.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*

import abc

class Scanner(abc.ABC):
    def __init__(self):
        self.scanner_resolution = 100

    @abc.abstractmethod
    def scan_document(self) -> str:
        pass

    @abc.abstractmethod
    def get_scanner_status(self) -> str:
        pass

class Printer(abc.ABC):
    def __init__(self):
        self.printer_resolution = 100
        self.printer_operation_history = False
        self.fax_machine = False

    @abc.abstractmethod
    def print_document(self) -> str:
        pass

    @abc.abstractmethod
```

```
    def get_printer_status(self) -> str:
        pass

class MFD():
    def __init__(self,serial_number=""):
        self.serial_number = serial_number

class MFD1(MFD,Scanner,Printer):
    def __init__(self,serial_number=""):
        MFD.__init__(self,serial_number)
        Scanner.__init__(self)
        Printer.__init__(self)

    def low(self):
        self.scanner_resolution = 100
        self.printer_resolution = 100
        self.printer_operation_history = False
        self.fax_machine = False

    def medium(self):
        self.scanner_resolution = 300
        self.printer_resolution = 300
        self.printer_operation_history = True
        self.fax_machine = False

    def high(self):
        self.scanner_resolution = 600
        self.printer_resolution = 600
        self.printer_operation_history = True
        self.fax_machine = True

    def scan_document(self):
        print("El document ha estat escanejat")
    def get_scanner_status(self):
        fax = "" if self.fax_machine == False else ", Fax"
        print( "màxima resolució scanner: {}{}  número de
sèrie:{}".format(self.scanner_resolution,fax,self.serial_number) )
    def print_document(self):
        print("El document ha estat imprés")
    def get_printer_status(self):
        poh = "" if self.printer_operation_history == False else ", Històric
d'impressora"
        print( "màxima resolució impressora: {}{}  número de
sèrie:{}".format(self.scanner_resolution,poh,self.serial_number) )

d1 = MFD1("d1-sn")
d1.low()
d1.get_scanner_status()
d1.get_printer_status()

d2 = MFD1("d2-sn")
d2.medium()
d2.get_scanner_status()
d2.get_printer_status()

d3 = MFD1("d3-sn")
```

```
d3.high()
d3.get_scanner_status()
d3.get_printer_status()
```

; resposta2.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*

import abc

class Scanner(abc.ABC):
    def __init__(self):
        self.scanner_resolution = 100

    @abc.abstractmethod
    def scan_document(self) -> str:
        pass

    @abc.abstractmethod
    def get_scanner_status(self) -> str:
        pass

class Printer(abc.ABC):
    def __init__(self):
        self.printer_resolution = 100
        self.printer_operation_history = False
        self.fax_machine = False

    @abc.abstractmethod
    def print_document(self) -> str:
        pass

    @abc.abstractmethod
    def get_printer_status(self) -> str:
        pass

class MFD():
    def __init__(self,serial_number=""):
        self.serial_number = serial_number

    def scan_document(self):
        print("El document ha estat escanejat")
    def get_scanner_status(self):
        fax = "" if self.fax_machine == False else ", Fax"
        print( "màxima resolució scanner: {}{}  número de
sèrie:{}".format(self.scanner_resolution,fax,self.serial_number) )
    def print_document(self):
        print("El document ha estat imprés")
    def get_printer_status(self):
        poh = "" if self.printer_operation_history == False else ", Històric
d'impressora"
        print( "màxima resolució impressora: {}{}  número de
sèrie:{}".format(self.scanner_resolution,poh,self.serial_number) )
```

```
class MFD1(MFD,Scanner,Printer):
    def __init__(self,serial_number=""):
        MFD.__init__(self,serial_number)
        Scanner.__init__(self)
        Printer.__init__(self)
        self.low()

    def low(self):
        self.scanner_resolution = 100
        self.printer_resolution = 100
        self.printer_operation_history = False
        self.fax_machine = False

class MFD2(MFD,Scanner,Printer):
    def __init__(self,serial_number=""):
        MFD.__init__(self,serial_number)
        Scanner.__init__(self)
        Printer.__init__(self)
        self.medium()

    def medium(self):
        self.scanner_resolution = 300
        self.printer_resolution = 300
        self.printer_operation_history = True
        self.fax_machine = False

class MFD3(MFD,Scanner,Printer):
    def __init__(self,serial_number=""):
        MFD.__init__(self,serial_number)
        Scanner.__init__(self)
        Printer.__init__(self)
        self.high()

    def high(self):
        self.scanner_resolution = 600
        self.printer_resolution = 600
        self.printer_operation_history = True
        self.fax_machine = True


d1 = MFD1("d1-sn")
d1.get_scanner_status()
d1.get_printer_status()

d2 = MFD2("d2-sn")
d2.get_scanner_status()
d2.get_printer_status()

d3 = MFD3("d3-sn")
d3.get_scanner_status()
d3.get_printer_status()
```

output

```
màxima resolució scanner: 100  número de sèrie:d1-sn
```

```
màxima resolució impressora: 100  número de sèrie:d1-sn
màxima resolució scanner: 300  número de sèrie:d2-sn
màxima resolució impressora: 300, Històric d'impressora  número de sèrie:d2-sn
màxima resolució scanner: 600, Fax  número de sèrie:d3-sn
màxima resolució impressora: 600, Històric d'impressora  número de sèrie:d3-sn
```