Mastering Python Design Patterns

Create various design patterns to master the art of solving problems using Python

Sakis Kasampalis
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “The Factory Pattern”
- A synopsis of the book’s content
- More information on Mastering Python Design Patterns

About the Author

Sakis Kasampalis (@SKasampalis) is a software engineer living in the Netherlands. He is not dogmatic about particular programming languages and tools; his principle is that the right tool should be used for the right job. One of his favorite tools is Python because he finds it very productive.

Sakis was also the technical reviewer of Mastering Object-oriented Python and Learning Python Design Patterns, published by Packt Publishing.

I want to thank my sweetheart, Georgia, for supporting this effort. Many thanks to Owen Roberts who encouraged me to write this book. I also want to thank Sumeet Sawant for being a very kind and cooperative content development editor. Last but not least, I want to thank the reviewers of this book for their valuable feedback.
Mastering Python Design Patterns

Design patterns

In software engineering, a design pattern is a recommended solution to a software design problem. Design patterns generally describe how to structure our code to solve common design problems using best practices. It is important to note that a design pattern is a high-level solution; it doesn't focus on implementation details such as algorithms and data structures [GOF95, page 13], [j.mp/srcmdp]. It is up to us, as software engineers, to decide which algorithm and data structure is optimal to use for the problem we are trying to solve.

The most important part of a design pattern is probably its name. The benefit of naming all patterns is that we have, on our hands, a common vocabulary to communicate [GOF95, page 13]. Thus, if you send some code for review and your peer reviewer gives feedback mentioning "I think that you can use a Strategy here instead of...", even if you don't know or remember what a strategy is, you can immediately look it up.

As programming languages evolve, some design patterns such as Singleton become obsolete or even antipatterns [j.mp/jalfdp], others are built in the programming language (iterator), and new patterns are born (Borg/Monostate [j.mp/amdpp], [j.mp/wikidpc]).

Common misunderstandings about design patterns

There are a few misunderstandings about design patterns. One misunderstanding is that design patterns should be used right from the start when writing code. It is not unusual to see developers struggling with which pattern they should use in their code, even if they haven't first tried to solve the problem in their own way [j.mp/prsedp], [j.mp/stedp].

Not only is this wrong, but it is also against the nature of design patterns. Design patterns are discovered (in contrast to invented) as better solutions over existing solutions. If you have no existing solution, it doesn't make sense to look for a better one. Just go ahead and use your skills to solve your problem as best as you think. If your code reviewers have no objections and through time you see that your solution is smart and flexible enough, it means that you don't need to waste your time on struggling about which pattern to use. You might have even discovered a better design pattern than the existing one. Who knows? The point is do not limit your creativity in favor of forcing yourself to use existing design patterns.
A second misunderstanding is that design patterns should be used everywhere. This results in creating complex solutions with unnecessary interfaces and hierarchies, where a simpler and straightforward solution would be sufficient. Do not treat design patterns as a panacea because they are not. They must be used only if there is proof that your existing code "smells", and is hard to extend and maintain. Try thinking in terms of you aren't gonna need it (YAGNI [j.mp/c2yagni]) and Keep it simple stupid (KISS [j.mp/wikikis]). Using design patterns everywhere is as evil as premature optimization [j.mp/c2pro].

Design patterns and Python

This book focuses on design patterns in Python. Python is different than most common programming languages used in popular design patterns books (usually Java [FFBS04] or C++ [GOF95]). It supports duck-typing, functions are first-class citizens, and some patterns (for instance, iterator and decorator) are built-in features. The intent of this book is to demonstrate the most fundamental design patterns, not all patterns that have been documented so far [j.mp/wikidpc]. The code examples focus on using idiomatic Python when applicable [j.mp/idiompyt]. If you are not familiar with the Zen of Python, it is a good idea to open the Python REPL right now and execute import this. The Zen of Python is both amusing and meaningful.

What This Book Covers

*Chapter 1, The Factory Pattern*, will teach you how to use the Factory design pattern (Factory Method and Abstract Factory) to initialize objects, and cover the benefits of using the Factory design pattern instead of direct object instantiation.

*Chapter 2, The Builder Pattern*, will teach you how to simplify the creation of objects that are typically composed by more than one related objects.

*Chapter 3, The Prototype Pattern*, will teach you how to create a new object that is a full copy (hence, the name clone) of an existing object.

Part 2: Structural patterns presents design patterns that deal with relationships between the entities (classes, objects, and so on) of a system.

*Chapter 4, The Adapter Pattern*, will teach you how to make your existing code compatible with a foreign interface (for example, an external library) with minimal changes.

*Chapter 5, The Decorator Pattern*, will teach you how to enhance the functionality of an object without using inheritance.

*Chapter 6, The Facade Pattern*, will teach you how to create a single entry point to hide the complexity of a system.
Chapter 7, *The Flyweight Pattern*, will teach you how to reuse objects from an object pool to improve the memory usage and possibly the performance of your applications.

Chapter 8, *The Model-View-Controller Pattern*, will teach you how to improve the maintainability of your applications by avoiding mixing the business logic with the user interface.

Chapter 9, *The Proxy Pattern*, will teach you how to improve the security of your application by adding an extra layer of protection.

Part 3: Behavioral patterns presents design patterns that deal with the communication of the system's entities.

Chapter 10, *The Chain of Responsibility Pattern*, will teach you how to send a request to multiple receivers.

Chapter 11, *The Command Pattern*, will teach you how to make your application capable of reverting already applied operations.

Chapter 12, *The Interpreter Pattern*, will teach you how to create a simple language on top of Python, which can be used by domain experts without forcing them to learn how to program in Python.

Chapter 13, *The Observer Pattern*, will teach you how to send notifications to the registered stakeholders of an object whenever its state changes.

Chapter 14, *The State Pattern*, will teach you how to create a state machine to model a problem and the benefits of this technique.

Chapter 15, *The Strategy Pattern*, will teach you how to pick (during runtime) an algorithm between many available algorithms, based on some input criteria (for example, the element size).

Chapter 16, *The Template Pattern*, will teach you how to make a clear separation between the common and different parts of an algorithm to avoid unnecessary code duplication.
The Factory Pattern

Creational design patterns deal with an object creation [j.mp.wikicrea]. The aim of a creational design pattern is to provide better alternatives for situations where a direct object creation (which in Python happens by the __init__() function [j.mp/divefunc], [Lott14, page 26]) is not convenient.

In the Factory design pattern, a client asks for an object without knowing where the object is coming from (that is, which class is used to generate it). The idea behind a factory is to simplify an object creation. It is easier to track which objects are created if this is done through a central function, in contrast to letting a client create objects using a direct class instantiation [Eckel08, page 187]. A factory reduces the complexity of maintaining an application by decoupling the code that creates an object from the code that uses it [Zlobin13, page 30].

Factories typically come in two forms: the Factory Method, which is a method (or in Pythonic terms, a function) that returns a different object per input parameter [j.mp/factorympat]; the Abstract Factory, which is a group of Factory Methods used to create a family of related products [GOF95, page 100], [j.mp/absfpat].

Factory Method

In the Factory Method, we execute a single function, passing a parameter that provides information about what we want. We are not required to know any details about how the object is implemented and where it is coming from.
The Factory Pattern

A real-life example
An example of the Factory Method pattern used in reality is in plastic toy construction. The molding powder used to construct plastic toys is the same, but different figures can be produced using different plastic molds. This is like having a Factory Method in which the input is the name of the figure that we want (duck and car) and the output is the plastic figure that we requested. The toy construction case is shown in the following figure, which is provided by www.sourcemaking.com [j.mp/factorympat].

A software example
The Django framework uses the Factory Method pattern for creating the fields of a form. The forms module of Django supports the creation of different kinds of fields (CharField, EmailField) and customizations (max_length, required) [j.mp/djangofacm].

Use cases
If you realize that you cannot track the objects created by your application because the code that creates them is in many different places instead of a single function/method, you should consider using the Factory Method pattern [Eckel08, page 187]. The Factory Method centralizes an object creation and tracking your objects becomes much easier. Note that it is absolutely fine to create more than one Factory Method, and this is how it is typically done in practice. Each Factory Method logically groups the creation of objects that have similarities. For example, one Factory Method might be responsible for connecting you to different databases (MySQL, SQLite), another Factory Method might be responsible for creating the geometrical object that you request (circle, triangle), and so on.
The Factory Method is also useful when you want to decouple an object creation from an object usage. We are not coupled/bound to a specific class when creating an object, we just provide partial information about what we want by calling a function. This means that introducing changes to the function is easy without requiring any changes to the code that uses it [Zlobin13, page 30].

Another use case worth mentioning is related to improving the performance and memory usage of an application. A Factory Method can improve the performance and memory usage by creating new objects only if it is absolutely necessary [Zlobin13, page 28]. When we create objects using a direct class instantiation, extra memory is allocated every time a new object is created (unless the class uses caching internally, which is usually not the case). We can see that in practice in the following code (file id.py), it creates two instances of the same class A and uses the id() function to compare their memory addresses. The addresses are also printed in the output so that we can inspect them. The fact that the memory addresses are different means that two distinct objects are created as follows:

```python
class A(object):
    pass

if __name__ == '__main__':
    a = A()
    b = A()

    print(id(a) == id(b))
    print(a, b)
```

Executing id.py on my computer gives the following output:

```
>> python3 id.py
False
<_main__.A object at 0x7f5771de8f60> <_main__.A object at 0x7f5771df2208>
```

Note that the addresses that you see if you execute the file are not the same as I see because they depend on the current memory layout and allocation. But the result must be the same: the two addresses should be different. There's one exception that happens if you write and execute the code in the Python **Read-Eval-Print Loop (REPL)** (interactive prompt), but that's a REPL-specific optimization which is not happening normally.
Implementation

Data comes in many forms. There are two main file categories for storing/retrieving data: human-readable files and binary files. Examples of human-readable files are XML, Atom, YAML, and JSON. Examples of binary files are the .sq3 file format used by SQLite and the .mp3 file format used to listen to music.

In this example, we will focus on two popular human-readable formats: XML and JSON. Although human-readable files are generally slower to parse than binary files, they make data exchange, inspection, and modification much easier. For this reason, it is advised to prefer working with human-readable files, unless there are other restrictions that do not allow it (mainly unacceptable performance and proprietary binary formats).

In this problem, we have some input data stored in an XML and a JSON file, and we want to parse them and retrieve some information. At the same time, we want to centralize the client’s connection to those (and all future) external services. We will use the Factory Method to solve this problem. The example focuses only on XML and JSON, but adding support for more services should be straightforward.

First, let’s take a look at the data files. The XML file, person.xml, is based on the Wikipedia example [j.mp/wikijson] and contains information about individuals (firstName, lastName, gender, and so on) as follows:

```xml
<persons>
  <person>
    <firstName>John</firstName>
    <lastName>Smith</lastName>
    <age>25</age>
    <address>
      <streetAddress>21 2nd Street</streetAddress>
      <city>New York</city>
      <state>NY</state>
      <postalCode>10021</postalCode>
    </address>
    <phoneNumbers>
      <phoneNumber type="home">212 555-1234</phoneNumber>
      <phoneNumber type="fax">646 555-4567</phoneNumber>
    </phoneNumbers>
    <gender><type>male</type></gender>
  </person>
  <person>
    <firstName>Jimy</firstName>
  </person>
</persons>
```
The JSON file, `donut.json`, comes from the GitHub account of Adobe [j.mp/adobejson] and contains donut information (type, price/unit that is, ppu, topping, and so on) as follows:

```
[
  {
    "id": "0001",
    "type": "donut",
    "name": "Cake",
    "ppu": 0.55,
```
The Factory Pattern

"batters": {
  "batter": [
    { "id": "1001", "type": "Regular" },
    { "id": "1002", "type": "Chocolate" },
    { "id": "1003", "type": "Blueberry" },
    { "id": "1004", "type": "Devil's Food" }
  ]
},
"topping": [
  { "id": "5001", "type": "None" },
  { "id": "5002", "type": "Glazed" },
  { "id": "5005", "type": "Sugar" },
  { "id": "5007", "type": "Powdered Sugar" },
  { "id": "5006", "type": "Chocolate with Sprinkles" },
  { "id": "5003", "type": "Chocolate" },
  { "id": "5004", "type": "Maple" }
],
{
  "id": "0002",
  "type": "donut",
  "name": "Raised",
  "ppu": 0.55,
  "batters": {
    "batter": [
      { "id": "1001", "type": "Regular" }
    ]
  },
  "topping": [
    { "id": "5001", "type": "None" },
    { "id": "5002", "type": "Glazed" },
    { "id": "5005", "type": "Sugar" },
    { "id": "5003", "type": "Chocolate" },
    { "id": "5004", "type": "Maple" }
  ]
},
{
  "id": "0003",
  "type": "donut",
  "name": "Old Fashioned",
  "ppu": 0.55,
  "batters": {
    "batter": [

We will use two libraries that are part of the Python distribution for working with XML and JSON: `xml.etree.ElementTree` and `json` as follows:

```python
import xml.etree.ElementTree as etree
import json
```

The `JSONConnector` class parses the JSON file and has a `parsed_data()` method that returns all data as a dictionary (`dict`). The `property` decorator is used to make `parsed_data()` appear as a normal variable instead of a method as follows:

```python
class JSONConnector:
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)

    @property
def parsed_data(self):
        return self.data
```

The `XMLConnector` class parses the XML file and has a `parsed_data()` method that returns all data as a list of `xml.etree.Element` as follows:

```python
class XMLConnector:
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
def parsed_data(self):
        return self.tree
```
The Factory Pattern

The `connection_factory()` function is a Factory Method. It returns an instance of `JSONConnector` or `XMLConnector` depending on the extension of the input file path as follows:

```python
def connection_factory(filepath):
    if filepath.endswith('json'):
        connector = JSONConnector
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)
```

The `connect_to()` function is a wrapper of `connection_factory()`. It adds exception handling as follows:

```python
def connect_to(filepath):
    factory = None
    try:
        factory = connection_factory(filepath)
    except ValueError as ve:
        print(ve)
    return factory
```

The `main()` function demonstrates how the Factory Method design pattern can be used. The first part makes sure that exception handling is effective as follows:

```python
def main():
    sqlite_factory = connect_to('data/person.sq3')
```

The next part shows how to work with the XML files using the Factory Method. **XPath** is used to find all `person` elements that have the last name **Liar**. For each matched person, the basic name and phone number information are shown as follows:

```python
xml_factory = connect_to('data/person.xml')
xml_data = xml_factory.parsed_data()
liars = xml_data.findall('.//{person}[{lastName}="{}"]'.format('Liar'))
print('found: {} persons'.format(len(liars)))
for liar in liars:
    print('first name: {}'.format(liar.find('firstName').text))
    print('last name: {}'.format(liar.find('lastName').text))
    print('phone number ({}):'.format(p.attrib['type']), p.text) for p in liar.find('phoneNumbers')
```
The final part shows how to work with the JSON files using the Factory Method. Here, there's no pattern matching, and therefore the name, price, and topping of all donuts are shown as follows:

```python
json_factory = connect_to('data/donut.json')
json_data = json_factory.parsed_data
print('found: {} donuts'.format(len(json_data)))
for donut in json_data:
    print('name: {}'.format(donut['name']))
    print('price: ${}'.format(donut['ppu']))
    [print('topping: {} {}'.format(t['id'], t['type'])) for t in donut['topping']]  
```

For completeness, here is the complete code of the Factory Method implementation (`factory_method.py`) as follows:

```python
import xml.etree.ElementTree as etree
import json

class JSONConnector:
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)

    @property
    def parsed_data(self):
        return self.data

class XMLConnector:
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def parsed_data(self):
        return self.tree

def connection_factory(filepath):
    if filepath.endswith('json'):
        connector = JSONConnector
    elif filepath.endswith('xml'):
        connector = XMLConnector
    else:
        raise ValueError('Cannot connect to {}'.format(filepath))
    return connector(filepath)
```
The Factory Pattern

def connect_to(filepath):
    factory = None
    try:
        factory = connection_factory(filepath)
    except ValueError as ve:
        print(ve)
    return factory

def main():
    sqlite_factory = connect_to('data/person.sq3')
    print()

    xml_factory = connect_to('data/person.xml')
    xml_data = xml_factory.parsed_data
    liars = xml_data.findall(".//{person}[lastName='Liar']")
    print('found: {} persons'.format(len(liars)))
    for liar in liars:
        print('first name: ' + liar.find('firstName').text)
        print('last name: ' + liar.find('lastName').text)
        [print('phone number ({}):'.format(p.attrib['type']), p.text) for p in liar.find('phoneNumbers')]
    print()

    json_factory = connect_to('data/donut.json')
    json_data = json_factory.parsed_data
    print('found: {} donuts'.format(len(json_data)))
    for donut in json_data:
        print('name: ' + donut['name'])
        print('price: ${}'.format(donut['ppu']))
        [print('topping: {} {}'.format(t['id'], t['type'])) for t in donut['topping']]

if __name__ == '__main__':
    main()

Here is the output of this program as follows:

>>> python3 factory_method.py
Cannot connect to data/person.sq3

found: 2 persons
first name: Jimy
last name: Liar
phone number (home): 212 555-1234
first name: Patty
last name: Liar
phone number (home): 212 555-1234
phone number (mobile): 001 452-8819

found: 3 donuts
name: Cake
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5005 Sugar
topping: 5007 Powdered Sugar
topping: 5006 Chocolate with Sprinkles
topping: 5003 Chocolate
topping: 5004 Maple
name: Raised
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5005 Sugar
topping: 5003 Chocolate
topping: 5004 Maple
name: Old Fashioned
price: $0.55
topping: 5001 None
topping: 5002 Glazed
topping: 5003 Chocolate
topping: 5004 Maple

Notice that although \texttt{JSONConnector} and \texttt{XMLConnector} have the same interfaces, what is returned by \texttt{parsed\_data()} is not handled in a uniform way. Different python code must be used to work with each connector. Although it would be nice to be able to use the same code for all connectors, this is at most times not realistic unless we use some kind of common mapping for the data which is very often provided by external data providers. Assuming that you can use exactly the same code for handling the XML and JSON files, what changes are required to support a third format, for example, SQLite? Find an SQLite file or create your own and try it.
As it is now, the code does not forbid a direct instantiation of a connector. Is it possible to do this? Try doing it.

Hint: Functions in Python can have nested classes.

Abstract Factory

The Abstract Factory design pattern is a generalization of Factory Method. Basically, an Abstract Factory is a (logical) group of Factory Methods, where each Factory Method is responsible for generating a different kind of object [Eckel08, page 193].

A real-life example

Abstract Factory is used in car manufacturing. The same machinery is used for stamping the parts (doors, panels, hoods, fenders, and mirrors) of different car models. The model that is assembled by the machinery is configurable and easy to change at any time. We can see an example of the car manufacturing Abstract Factory in the following figure, which is provided by [www.sourcemaking.com][j.mp/absfpat].
A software example

The **django_factory** package is an Abstract Factory implementation for creating Django models in tests. It is used for creating instances of models that support test-specific attributes. This is important because the tests become readable and avoid sharing unnecessary code [j.mp/djangoabs].

Use cases

Since the Abstract Factory pattern is a generalization of the Factory Method pattern, it offers the same benefits: it makes tracking an object creation easier, it decouples an object creation from an object usage, and it gives us the potential to improve the memory usage and performance of our application.

But a question is raised: how do we know when to use the Factory Method versus using an Abstract Factory? The answer is that we usually start with the Factory Method which is simpler. If we find out that our application requires many Factory Methods which it makes sense to combine for creating a family of objects, we end up with an Abstract Factory.

A benefit of the Abstract Factory that is usually not very visible from a user's point of view when using the Factory Method is that it gives us the ability to modify the behavior of our application dynamically (in runtime) by changing the active Factory Method. The classic example is giving the ability to change the look and feel of an application (for example, Apple-like, Windows-like, and so on) for the user while the application is in use, without the need to terminate it and start it again [GOF95, page 99].

Implementation

To demonstrate the Abstract Factory pattern, I will reuse one of my favorite examples, included in *Python 3 Patterns & Idioms*, Bruce Eckel, [Eckel08, page 193]. Imagine that we are creating a game or we want to include a mini-game as part of our application to entertain our users. We want to include at least two games, one for children and one for adults. We will decide which game to create and launch in runtime, based on user input. An Abstract Factory takes care of the game creation part.
Let's start with the kid's game. It is called FrogWorld. The main hero is a frog who enjoys eating bugs. Every hero needs a good name, and in our case the name is given by the user in runtime. The `interact_with()` method is used to describe the interaction of the frog with an obstacle (for example, bug, puzzle, and other frog) as follows:

```python
class Frog:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        print('{} the Frog encounters {} and {}!'.format(self, obstacle, obstacle.action()))
```

There can be many different kinds of obstacles but for our example an obstacle can only be a Bug. When the frog encounters a bug, only one action is supported: it eats it!

```python
class Bug:
    def __str__(self):
        return 'a bug'

    def action(self):
        return 'eats it'
```

The FrogWorld class is an Abstract Factory. Its main responsibilities are creating the main character and the obstacle(s) of the game. Keeping the creation methods separate and their names generic (for example, `make_character()`, `make_obstacle()`) allows us to dynamically change the active factory (and therefore the active game) without any code changes. In a statically typed language, the Abstract Factory would be an abstract class/interface with empty methods, but in Python this is not required because the types are checked in runtime [Eckel08, page 195], [j.mp/ginstromdp] as follows:

```python
class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Frog World ------'
```
def make_character(self):
    return Frog(self.player_name)

def make_obstacle(self):
    return Bug()

The WizardWorld game is similar. The only differences are that the wizard battles against monsters like orks instead of eating bugs!

class Wizard:
    def __init__(self, name):
        self.name = name

def __str__(self):
    return self.name

    def interact_with(self, obstacle):
        print('{} the Wizard battles against {} and {}!'.format(self, obstacle, obstacle.action()))

class Ork:
    def __str__(self):
        return 'an evil ork'

    def action(self):
        return 'kills it'

class WizardWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

def __str__(self):
    return '\n\n\t------ Wizard World -------'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()}
The Factory Pattern

The GameEnvironment is the main entry point of our game. It accepts factory as an input, and uses it to create the world of the game. The play() method initiates the interaction between the created hero and the obstacle as follows:

```python
class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)
```

The validate_age() function prompts the user to give a valid age. If the age is not valid, it returns a tuple with the first element set to False. If the age is fine, the first element of the tuple is set to True and that's the case where we actually care about the second element of the tuple, which is the age given by the user as follows:

```python
def validate_age(name):
    try:
        age = input('Welcome {}. How old are you? '.format(name))
        age = int(age)
    except ValueError as err:
        print("Age {} is invalid, please try again...".format(age))
        return (False, age)
    return (True, age)
```

Last but not least comes the main() function. It asks for the user's name and age, and decides which game should be played by the age of the user as follows:

```python
def main():
    name = input("Hello. What's your name? ")
    valid_input = False
    while not valid_input:
        valid_input, age = validate_age(name)
        game = FrogWorld if age < 18 else WizardWorld
        environment = GameEnvironment(game(name))
        environment.play()
```

And the complete code of the Abstract Factory implementation (abstract_factory.py) is given as follows:

```python
class Frog:
    def __init__(self, name):
        self.name = name
```
def __str__(self):
    return self.name

def interact_with(self, obstacle):
    print('{} the Frog encounters {} and {}!'.format(self, obstacle, obstacle.action()))

class Bug:
    def __str__(self):
        return 'a bug'
    def action(self):
        return 'eats it'

class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name
    def __str__(self):
        return '

	------ Frog World -------'
    def make_character(self):
        return Frog(self.player_name)
    def make_obstacle(self):
        return Bug()

class Wizard:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return self.name
    def interact_with(self, obstacle):
        print('{} the Wizard battles against {} and {}!'.format(self, obstacle, obstacle.action()))

class Ork:
    def __str__(self):
        return 'an evil ork'
The Factory Pattern

def action(self):
    return 'kll it'

class WizardWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------ Wizard World -------'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()

class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)

def validate_age(name):
    try:
        age = input('Welcome {}. How old are you? '.format(name))
        age = int(age)
    except ValueError as err:
        print("Age {} is invalid, please try again...".format(age))
    return (False, age)

def main():
    name = input("Hello. What's your name? ")
    valid_input = False
    while not valid_input:
        valid_input, age = validate_age(name)
        game = FrogWorld if age < 18 else WizardWorld
        environment = GameEnvironment(game(name))
        environment.play()

    if __name__ == '__main__':
        main()
A sample output of this program is as follows:

```python
>>> python3 abstract_factory.py
Hello. What's your name? Nick
Welcome Nick. How old are you? 17
------ Frog World -------
Nick the Frog encounters a bug and eats it!
```

Try extending the game to make it more complete. You can go as far as you want: many obstacles, many enemies, and whatever else you like.

**Summary**

In this chapter, we have seen how to use the Factory Method and the Abstract Factory design patterns. Both patterns are used when we want to (a) track an object creation, (b) decouple an object creation from an object usage, or even (c) improve the performance and resource usage of an application. Case (c) was not demonstrated in the chapter. You might consider it as a good exercise.

The Factory Method design pattern is implemented as a single function that doesn't belong to any class, and is responsible for the creation of a single kind of object (a shape, a connection point, and so on). We saw how the Factory Method relates to toy construction, mentioned how it is used by Django for creating different form fields, and discussed other possible use cases for it. As an example, we implemented a Factory Method that provides access to the XML and JSON files.

The Abstract Factory design pattern is implemented as a number of Factory Methods that belong to a single class and are used to create a family of related objects (the parts of a car, the environment of a game, and so forth). We mentioned how the Abstract Factory is related with car manufacturing, how the django_factory Django package makes use of it to create clean tests, and covered the use cases of it. The implementation of the Abstract Factory is a mini-game that shows how we can use many related factories in a single class.

In the next chapter, we will talk about the Builder pattern, which is another creational pattern that can be used for fine-controlling the creation of complex objects.
Where to buy this book

You can buy Mastering Python Design Patterns from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.