Python is a versatile programming language that can be used for a wide range of technical tasks—computation, statistics, data analysis, game development, and more.

Python Unlocked walks you through the most effective techniques and best practices for high-performance Python programming—showing you how to make the most of the Python language. You'll get to know objects and functions inside and out and will learn how to use them to your advantage in your programming projects. You will also find out how to work with a range of design patterns, think out of box for algorithms, and find out data structures available for Python. You will develop a better understanding of testing with Python. Finally, you will learn tricks to optimize and scale your code for Python.

If you want the edge when it comes to Python, use this book to unlock the secrets of smarter Python programming.

What you will learn from this book
- Manipulate object creation processes for instances, classes, and functions
- Use the best possible language constructs to write data structures with super speed and maintainability
- Make efficient use of design patterns to decrease development time and make your code more maintainable
- Write better test cases with an improved understanding of the testing framework of Python, and discover how to develop new functionalities in it
- Write fully-optimized code with the Python language by profiling, compiling C modules, and more
- Learn how to scale the applications to build efficient, robust applications

Who this book is written for
If you are a Python developer and you think that you don't know everything about the language yet, then this is the book for you. It is assumed that you know the basics of the Python language.

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Objects in Depth'
- A synopsis of the book’s content
- More information on Python Unlocked
About the Author

Arun Tigeraniya has a BE in electronics and communication. After his graduation, he worked at various companies as a Python developer. His main professional interests are AI and Big Data. He enjoys writing efficient and testable code, and interesting technical articles. He has worked with open source technology since 2008. He currently works at Jaarvis Labs Limited, India.
Preface

Python is a versatile programming language that can be used for a wide range of technical tasks—computation, statistics, data analysis, game development, and more. Though Python is easy to learn, its range of features means there are many aspects of it that even experienced Python developers don’t know about. Even if you’re confident with the basics, its logic and syntax, by digging deeper you can work much more effectively with Python—and get more from the language.

*Python Unlocked* walks you through the most effective techniques and best practices for high performance Python programming—showing you how to make the most of the Python language. You’ll get to know objects and functions inside and out, and will learn how to use them to your advantage in your programming projects. You will also find out how to work with a range of design patterns, including abstract factory, singleton, and the strategy pattern, all of which will help make programming with Python much more efficient. As the process of writing a program is never complete without testing it, you will learn to test threaded applications and run parallel tests.

If you want the edge when it comes to Python, use this book to unlock the secrets of smarter Python programming.

What this book covers

*Chapter 1, Objects in Depth*, discusses object properties, attributes, creation and how calling objects work.

*Chapter 2, Namespaces and Classes*, discusses namespaces, how imports work, class multiple inheritance, MRO, Abstract classes, and protocols.

*Chapter 3, Functions and Utilities*, teaches function definitions, decorators, and some utilities.
Chapter 4, Data Structures and Algorithms, discusses in-built, library, third party data structures and algorithms.

Chapter 5, Elegance with Design Patterns, covers many important design patterns.

Chapter 6, Test-Driven Development, discusses mock objects, parameterization, creating custom test runners, testing threaded applications, and running testcases in parallel.

Chapter 7, Optimization Techniques, covers optimization techniques, profiling, using fast libraries, and compiling C modules.

Chapter 8, Scaling Python, covers multithreading, multiprocessing, asynchronization, and scaling horizontally.
In this chapter, we will dive into Python objects. Objects are the building blocks of the language. They may represent or abstract a real entity. We will be more interested in factors affecting such behavior. This will help us understand and appreciate the language in a better way. We will cover the following topics:

- Object characteristics
- Calling objects
- How objects are created
- Playing with attributes

Understanding objects

Key 1: Objects are language's abstraction for data. Identity, value, and type are characteristic of them.

All data and items that we work on in a program are objects, such as numbers, strings, classes, instances, and modules. They possess some qualities that are similar to real things as all of them are uniquely identifiable just like humans are identifiable by their DNA. They have a type that defines what kind of object it is, and the properties that it supports, just like humans of type cobbler support repairing shoes, and blacksmiths support making metal items. They possess some value, such as strength, money, knowledge, and beauty do for humans.

Name is just a means to identify an object in a namespace similar to how it is used to identify a person in a group.
Identity

In Python, every object has a unique identity. We can get this identity by passing an object to built-in ID function ID (object). This returns the memory address of the object in CPython.

Interpreter can reuse some objects so that the total number of objects remains low. For example, integers and strings can be reused in the following manner:

```python
g>>> i = "asdf"
g>>> j = "asdf"
g>>> id(i) == id(j)
True
g>>> i = 10000000000000000000000000000000
>>> j = 10000000000000000000000000000000
>>> id(j) == id(i) #cpython 3.5 reuses integers till 256
False
g>>> i = 4
g>>> j = 4
g>>> id(i) == id(j)
True
g>>> class Kls:
...     pass
...
g>>> k = Kls()
g>>> j = Kls()
g>>> id(k) == id(j) #always different as id gives memory address
False
```

This is also a reason that addition of two strings is a third new string, and, hence, it is best to use the StringIO module to work with a buffer, or use the join attribute of strings:

```python
g>>> # bad
g... print('a' + ' ' + 'simple' + ' ' + 'sentence' + ' ' + '')
a simple sentence
>>> #good
g... print(' '.join(['a','simple','sentence','.']))
a simple sentence .
```
Value

Key 2: Immutability is the inability to change an object’s value.

The value of the object is the data that is stored in it. Data in an object can be stored as numbers, strings, or references to other objects. Strings, and integers are objects themselves. Hence, for objects that are not implemented in C (or core objects), it is a reference to other objects, and we perceive value as the group value of the referenced object. Let’s take an example of an object iC instance of the C class with the str and lst attributes, as shown in the following diagram:

![Diagram of object iC with str and lst attributes]

The code snippet to create this object will be as follows:

```python
>>> class C:
...     def __init__(self, arg1, arg2):
...         self.str = arg1
...         self.lst = arg2
... >>> iC = C("arun",[1,2])
>>> iC.str
'arun'
>>> iC.lst
[1, 2]
>>> iC.lst.append(4)
>>> iC.lst
[1, 2, 4]
```
Then, when we modify \( iC \), we are either changing the objects references via attributes, or we are changing the references themselves and not the object \( iC \). This is important in understanding immutable objects because being immutable means not being able to change references. Hence, we can change mutable objects that are referenced by immutable objects. For example, lists inside tuple can be changed because the referenced objects are changing, not the references.

**Type**

**Key 3: Type is instance's class.**

An object's type tells us about the operations and functionality that the object supports, and it may also define the possible values for objects of that type. For example, your pet may be of type `dog` (an instance of the `dog` class) or `cat` (an instance of the `cat` class). If it is of type `dog`, it can bark; and if it is type `cat`, it can meow. Both are a type of animal (`cat` and `dog` inherit from the `animal` class).

An object's class provides a type to it. Interpreter gets the object's class by checking its `__class__` attribute. So, we can change an object's type by changing its `__class__` attribute:

```python
>>> k = []
>>> k.__class__
<class 'list'>
>>> type(k)
<class 'list'>
# type is instance's class
>>> class M:
...     def __init__(self,d):
...         self.d = d
...     def square(self):
...         return self.d * self.d
... 
>>> class N:
...     def __init__(self,d):
...         self.d = d
...     def cube(self):
...         return self.d * self.d * self.d
...
>>> m = M(4)
>>> type(m)  #type is its class
<class '__main__.M'>
```
>>> m.square()  # square defined in class M
16
>>> m.__class__ = N  # now type should change
>>> m.cube()  # cube in class N
64
>>> type(m)
<class '__main__.N'>  # yes type is changed

This will not work for built-in, compiled classes as it works only for class objects defined on runtime.

Making calls to objects

Key 4: All objects can be made callable.

To reuse and group code for some task, we group it in the functions classes, and then call it with different inputs. The objects that have a __call__ attribute are callable and __call__ is the entry point. For the C class, tp_call is checked in its structure:

```python
>>> def func():  # a function
...     print("adf")
...     print("asdf")
...     print("asdfd")

>>> func()
adf
>>> func.__call__()
adf
>>> func.__class__.__call__(func)
adm
```

```python
>>> class C:  # a callable class
...     def __call__(self):
...         print("adf")
...         print("asdf")
...         print("asdfd")

>>> c = C()
>>> c()
```

```python
>>> callable(lambda x:x+1)  # testing whether object is callable or not
```

True
Objects in Depth

>>> isinstance(lambda x:x+1, collections.Callable) # testing whether object is callable or not
True

Methods in classes are similar to functions, except that they are called with an implicit instance as a first argument. The functions are exposed as methods when they are accessed from the instance. The function is wrapped in a method class and returned. The method class stores instances in __self__ and function in __func__, and its __call__ method calls __func__ with first argument as __self__:

```python
>>> class D:
...     pass
...
>>> class C:
...     def do(self):
...         print("do run",self)
...
>>> def doo(obj):
...     print("doo run",obj)
...
>>> c = C()
>>> d = D()
>>> doo(c)
doo run <__main__.C object at 0x7fcf543625c0>
>>> doo(d)
doo run <__main__.D object at 0x7fcf54362400>
>>> # we do not need to pass object in case of C class do method
...
>>> c.do() # implicit pass of c object to do method
do run <__main__.C object at 0x7fcf543625c0>
```
do run <__main__.C object at 0x7fcf543625c0>
>>> C.do(d)

do run <__main__.D object at 0x7fcf54362400>
>>> c.do.__func__(d) #we called real function this way

do run <__main__.D object at 0x7fcf54362400>

Using this logic, we can also collect methods that are needed from other classes in
the current class, like the following code, instead of multiple inheritances if data
attributes do not clash. This will result in two dictionary lookups for an attribute
search: one for instance, and one for class.

```python
>>> #in library
... class PrintVals:
...     def __init__(self, endl):
...         self.endl = endl
...
...     def print_D8(self, data):
...         print("{0} {1} {2}".format(data[0], data[1], self.endl))
...

>>> class PrintKVals: #from in2 library
...     def __init__(self, knm):
...         self.knm = knm
...
...     def print_D8(self, data):
...         print("{2}:{0} {1}".format(data[0], data[1], self.knm))
...

>>> class CollectPrint:
...
...     def __init__(self, endl):
...         self.endl = endl
...         self.knm = "[k]"
...
...     print_D8 = PrintVals.print_D8
...     print_D8K = PrintKVals.print_D8
...

>>> c = CollectPrint("}
>>> c.print_D8([1,2])
1 2 }
>>> c.print_D8K([1,2])
[k]:1 2
```
When we call classes, we are calling its type, that is metaclass, with class as a first argument to give us a new instance:

```python
>>> class Meta(type):
...     def __call__(*args):
...         print("meta call",args)
...     
>>> class C(metaclass=Meta):
...     pass
...
>>> c = C()
meta call (<class '__main__.C'>,)
>>> c = C.__class__.__call__(C)
meta call (<class '__main__.C'>,)
```

Similarly, when we call instances, we are calling their type, that is class, with instance as first argument:

```python
>>> class C:
...     def __call__(*args):
...         print("C call",args)
...     
>>> c = C()
>>> c()
C call (<__main__.C object at 0x7f5d70c2bb38>,)
>>> c.__class__.__call__(c)
C call (<__main__.C object at 0x7f5d70c2bb38>,)
```

### How objects are created

Objects other than built-in types or compiled module classes are created at runtime. Objects can be classes, instances, functions, and so on. We call an object's type to give us a new instance; or put in another way, we call a type class to give us an instance of that type.
Creation of function objects

Key 5: Create function on runtime.

Let's first take a look at how function objects can be created. This will broaden our view. This process is done by interpreter behind the scenes when it sees a `def` keyword. It compiles the code, which is shown as follows, and passes the code name arguments to the function class that returns an object:

```python
>>> function_class = (lambda x:x).__class__
>>> function_class
<class 'function'>
>>> def foo():
...     print("hello world")
...     ...
>>> def myprint(*args,**kwargs):
...     print("this is my print")
...     print(*args,**kwargs)
...
>>> newfunc1 = function_class(foo.__code__, {'print':myprint})
>>> newfunc1()
this is my print
hello world
>>> newfunc2 = function_class(compile("print('asdf')","filename","single"),{'print':print})
>>> newfunc2()
asdf
```
**Creation of instances**

*Key 6: Process flow for instance creation.*

We call class to get a new instance. We saw from the making calls to objects section that when we call class, it calls its metaclass `__call__` method to get a new instance. It is the responsibility of `__call__` to return a new object that is properly initialized. It is able to call class's `__new__` and `__init__` because class is passed as first argument, and instance is created by this function itself:

```python
>>> class Meta(type):
...     def __call__(*args):
...         print("meta call ",args)
...         return None
...     ...

>>> class C(metaclass=Meta):
...     def __init__(*args):
...         print("C init not called",args)
...     ...

>>> c = C()  #init will not get called
meta call  (<class '__main__.C'>,)
>>> print(c)
None
```

To enable developer access to both functionalities, creating new object, and initializing new object, in class itself; `__call__` calls the `__new__` class to return a new object and `__init__` to initialize it. The full flow can be visualized as shown in the following code:

```python
>>> class Meta(type):
...     def __call__(*args):
...         print("meta call ,class object :",args)
...         class_object = args[0]
...         if '__new__' in class_object.__dict__:
...             new_method = getattr(class_object,'__new__',None)
...             instance = new_method(class_object)
...         else:
...             instance = object.__new__(class_object)
...         if '__init__' in class_object.__dict__:
...             init_method =  getattr(class_object,'__init__',None)
...             init_method(instance,*args[1:])
...         return instance
...     ...
```
>>> class C(metaclass=Meta):
...     def __init__(instance_object, *args):
...         print("class init", args)
...     def __new__(*args):
...         print("class new", args)
...         return object.__new__(*args)
...     
>>> class D(metaclass=Meta):
...     pass
...
>>> c=C(1,2)
meta call ,class object : (<class '__main__.C'>, 1, 2)
class new (<class '__main__.C'>,)
class init (1, 2)
>>> d = D(1,2)
meta call ,class object : (<class '__main__.D'>, 1, 2)

Take a look at the following diagram:
Creation of class objects

Key 7: Process flow for class creation.

There are three ways in which we can create classes. One is to simply define the class. The second one is to use the built-in \_\_build\_class\_\_ function, and the third is to use the \new\_class\_ method of type module. Method one uses two, method two uses method three internally. When interpreter sees a class keyword, it collects the name, bases, and metaclass that is defined for the class. It will call the \_\_build\_class\_\_ built-in function with function (with the code object of the class), name of the class, base classes, metaclass that is defined, and so on:

\_\_build\_class\_\_(func, name, *bases, metaclass=None, **kwds) -> class

This function returns the class. This will call the \_\_prepare\_\_ class method of metaclass to get a mapping data structure to use as a namespace. The class body will be evaluated, and local variables will be stored in this mapping data structure. Metaclass's type will be called with this namespace dictionary, bases, and class name. It will in turn call the \_\_new\_\_ and \_\_init\_\_ methods of metaclass. Metaclass can change attributes passed to its method:

```python
>>> function_class = (lambda x:x).__class__
>>> M = __build_class__(function_class(
        ...     compile("def __init__(self,):
    print('adf')","
        ...     '<stdin>',
        ...     'exec'),
        ...     {'print':print}
        ... ),
        ...     'MyCls')
>>> m = M()
adf
>>> print(M,m)
<class '__main__.MyCls'> __main__.MyCls object at 0x0088B430
```
Take a look at the following diagram:
Playing with attributes

Key 8: Which attribute will be used.

Attributes are values that are associated with an object that can be referenced by name using dotted expressions. It is important to understand how attributes of an object are found. The following is the sequence that is used to search an attribute:

1. If an attribute is a special method, and it exists in the object's type (or bases), return it, for example: __call__, __str__, and __init__. When these methods are searched, their behavior is only in the instance's type:

   ```python
   >>> class C:
   ...     def __str__(self,):
   ...         return 'Class String'
   ...     def do(self):
   ...         return 'Class method'
   ...
   >>> c = C()
   >>> print(c)
   Class String
   >>> print(c.do())
   Class method
   >>> def strf(*args):
   ...     return 'Instance String',args
   ...
   >>> def doo(*args):
   ...     return 'Instance Method'
   ...
   >>> c.do = doo
   >>> c.__str__ = strf
   >>> print(c)
   Class String
   >>> print(c.do())
   Instance Method
   ```

2. If an object's type has a __getattr__ attribute, then this method is invoked to get the attribute whether this attribute is present or not. It is the total responsibility of __getattr__ to get the attribute. As shown in the following code snippet, even if the do method is present, it is not found as getattr didn't return any attribute:

   ```python
   >>> class C:
   ...     def do(self):
   ...         print("asdf")
   ...     def __getattr__(self,attr):
   ...         return 'Instance attribute not found'
   ...
   >>> c = C()
   >>> print(c.do())
   asdf
   >>> print(c.name)
   Instance attribute not found
   ```
... raise AttributeError('object has no attribute
"%s"' % attr)
...
>>> c = C()
>>> c.do()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 5, in __getattribute__
AttributeError: object has no attribute "do"

3. Search in object's type __dict__ to find the attribute. If it is present, and it is
data descriptor, return it:

>>> class Desc:
...     def __init__(self, i):
...         self.i = i
...     def __get__(self, obj, objtype):
...         return self.i
...     def __set__(self, obj, value):
...         self.i = value
...
>>> class C:
...     attx = Desc(23)
...
>>> c = C()
>>> c.attx
23
>>> c.__dict__['attx'] = 1234
>>> c.attx
23
>>> C.attx = 12
>>> c.attx
1234

4. Search in object's __dict__ type (and if this object is class, search bases
__dict__ as well) to find the attribute. If the attribute is descriptor, return the result.

5. Search in object's type __dict__ to find the attribute. If the attribute is found, return it. If it is non-data descriptor, return its result, and check in other bases using the same logic:

>>> class Desc:
...     def __init__(self, i):
...         self.i = i
...     def __get__(self, obj, objtype):
...         return self.i
...
>>> class C:
...     attx = Desc(23)
...
>>> c = C()
>>> c.attx
23
>>> c.__dict__['attx'] = 34
>>> c.attx
34

6. If object type's __getattr__ is defined, check whether it can give us the attribute:

```python
>>> class C:
...     def __getattr__(self, key):
...         return key+'_#'
...
>>> c = C()
>>> c.asdf
'asdf_#'
```

7. Raise AttributeError.

Descriptors
Key 9: Making custom behavior attributes.

Any attribute of a class, which is an object defining any of these methods, acts as a descriptor:

- `__get__(self, obj, type=None) --> value`
- `__set__(self, obj, value) --> None`
- `__delete__(self, obj) --> None`

When an attribute is searched in an object first, it is searched in its dictionary then its type's (base class's) dictionary. If found, object has one of these methods defined and that method is invoked instead. Let's assume that b is an instance of the B class, then the following will happen:

- Invocation through class is type.__getattr__() transforming to B.__dict__['x'].__get__(None, B)
- Invocation through instance is object.__getattr__() --> type(b).__dict__['x'].__get__(b, type(b))
Objects with only `__get__` are non-data descriptors, and objects that include `__set__` / `__del__` are data descriptors. Data descriptors take precedence over instance attributes, whereas non-data descriptors do not.

Class, static, and instance methods

Key 10: Implementing class method and static method.

Class, static, and instance methods are all implementable using descriptors. We can understand descriptors and these methods in one go:

- Class methods are methods that always get class as their first argument and they can be executed without any instance of class.
- Static methods are methods that do not get any implicit objects as first argument when executed via class or instance.
- Instance methods get instances when called via instance but no implicit argument when called via class.

A sample code usage of these methods is as follows:

```python
>>> class C:
...     @staticmethod
...     def sdo(*args):
...         print(args)
...     @classmethod
...     def cdo(*args):
...         print(args)
...     def do(*args):
...         print(args)
...
>>> ic = C()
# staticmethod called through class: no implicit argument is passed
>>> C.sdo(1,2)
(1, 2)
# staticmethod called through instance: no implicit argument is passed
>>> ic.sdo(1,2)
(1, 2)
# classmethod called through instance: first argument implicitly class
>>> ic.cdo(1,2)
<class '__main__.C'>, 1, 2)
# classmethod called through class: first argument implicitly class
>>> C.cdo(1,2)
```
Objects in Depth

(<class '__main__.C'>, 1, 2)
# instancemethod called through instance: first argument implicitly
instance
>>> ic.do(1,2)
(<__main__.C object at 0x00DC9E30>, 1, 2)
# instancemethod called through class: no implicit argument, acts like
static method.
>>> C.do(1,2)
(1, 2)

They can be understood and implemented using descriptors easily as follows:

from functools import partial
>>> class my_instancemethod:
...     def __init__(self, f):
...         # we store reference to function in instance
...         # for future reference
...         self.f = f
...     def __get__(self, obj, objtype):
...         # obj is None when called from class
...         # objtype is always present
...         if obj is not None:
...             return partial(self.f,obj)
...         else: # called from class
...             return self.f
...

>>> class my_classmethod:
...     def __init__(self, f):
...         self.f = f
...     def __get__(self, obj, objtype):
...         # we pass objtype i.e class object
...         # when called from instance or class
...         return partial(self.f,objtype)
...

>>> class my_staticmethod:
...     def __init__(self, f):
...         self.f = f
...     def __get__(self, obj, objtype):
...         # we do not pass anything
...         # for both conditions
...         return self.f
...

>>> class C:
...     @my_instancemethod
...     def ido(*args):
...
...    print("imethod", args)
...    @my_classmethod
def cdo(*args):
...    print("cmethod", args)
...    @my_staticmethod
def sdo(*args):
...    print("smethod", args)
...
>>> c = C()
>>> c.ido(1,2)
imethod (<__main__.C object at 0x00D7CBD0>, 1, 2)
>>> C.ido(1,2)
imethod (1, 2)
>>> c.cdo(1,2)
cmethod (<class '__main__.C'>, 1, 2)
>>> C.cdo(1,2)
cmethod (<class '__main__.C'>, 1, 2)
>>> c.sdo(1,2)
smethod (1, 2)
>>> C.sdo(1,2)
smethod (1, 2)

Summary

In this chapter, we dived into how objects work in the Python language, how are they connected, and how are they called. Descriptors and instance creation are very important topics as they give us a picture of how system works. We also dived into how attributes are looked up for objects.

Now, we are all prepared to learn how to use language constructs to their maximum potential. In the next chapter, we will also discover utilities that are extremely helpful in elegantly finishing a project.

[19]
**Where to buy this book**

You can buy Python Unlocked from the [Packt Publishing website](http://www.PacktPub.com).

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

[Click here](http://www.PacktPub.com) for ordering and shipping details.