Python

Mingming Li

First Created: Feb 26, 2023
Last Modified: March 10, 2023

Contents

List of Figures

List of Tables

Chapter 1

Module

Functions encapsulate pieces of code so that they can be reused throughout a program. Modules collect sets of functions together so that they can be used by any number of programs. Packages group sets of modules because their modules provide related functionality or because they depend on each other.

It is important to be aware of what the library can offer, since using predefined functionality makes programming much faster than creating everything from scratch.

Syntax for importing:

```
import importable
import importable1, importable2, ..., importableN
import importable as preferred_name

from importable import object as preferred_name
from importable import object1, object2, ..., objectN
from importable import *
```

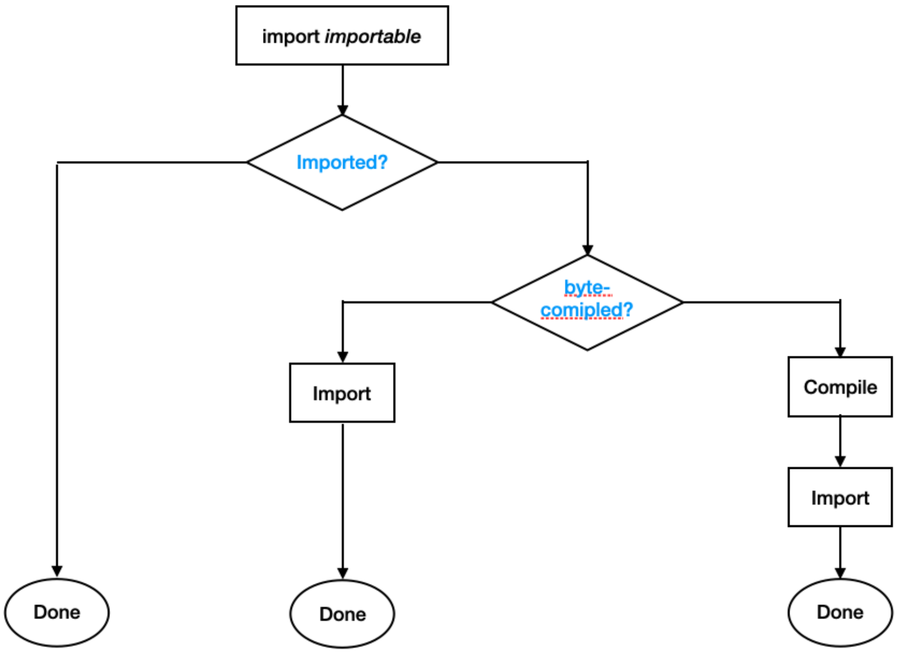
In the last syntex, the * means “import everything that is not private”, which in practical terms means either every object in the module is imported except for those whose names begin with a leading underscore, or, if the module has a global `__all__` variable that holds a list of names, that all the objects named in the `__all__` variable are imported.

How does Python know where to look for the modules and packages that are imported?

The built-in `sys` module has a list called `sys.path` that holds a list of the directories that consistutes the **Python path**. The first directory is the directory that contains the program itself, even if the program was invoked from another directory. If the `PYTHONPATH` environment variable is set, the paths specified in it are the next ones in the list. The final paths are those needed to access Python’s standard library — these are set when Python is installed.

When we first import a module, if it isn’t built-in, Python looks for the module in each path listed in `sys.path` in turn.

Using byte-code compiled files leads to faster start-up times since the interpreter only has to load and run the code, rather than load, compile, (save if possible),and run the code;runtimes are not affected, though. WhenPythonis installed, the standard library modules are usually byte-code compiled as part of the installation process.



Figures 1.1: Import process

1.1 Package

A package is simply a directory that contains a set of modules and a file called `__init__.py`.

In some situations it is convenient to load in all of a **package**'s modules using a single statement. To do this we must edit the **package**'s `__init__.py` file to contain a statement which specifies which modules we want loaded. This statement must assign a list of module names to the special variable `__all__`.

This syntax can also be applied to a module in which case all the functions, variables, and other object defined in the module (appart from those whose names begin with a leading underscore) will be imported. If we want to control exactly what is imported, we can define an `__all__` list in the module itself.

1.2 Custom module

You can define your own module. If you want your module to be available to all your program, there are 3 approaches:

1. Put the module in the Python distribution's `site-packages` subdirectory.
2. Create a directory for the custom modules and set the `PYTHONPATH` environment variable to this directory
3. Put the module in the local site-packages subdirectory (`/.local/lib/python3.9/site-packages`)

The second and third approaches have the advantage of keeping our own code separate from the official installation.

Doctesting is usually done by the following code:

```python
if __name__ == '__main__':
    import doctest

    doctest.testmod()
```

Whenever a module is imported Python creates a variable for the module called `__name__` and store the modules name in this variable. A modules name is simply the name of its `.py` file but without the extension.

Whenever a `.py` file is run Python creates a variable for the program called `__name__` and sets it to the string `__main__`.

Chapter 2

Programming techniques

2.1 Generator

Generator provide a means of performing lazy evaluation, which means that they compute only the values that are actually needed. This can be more efficient than computing a very large list in one go.

There are two ways to create a generator:

- generator expression
- the keyword **yield**

```
(expression for item in iterable)
(expression for item in iterable if condition)
```

If we need all the items in one go we can pass the generator returned to **list()** or **tuple()**. We can use **next()** to regrieve the next item from the generator.

2.2 Dynamic code execution

Dynamic code execution means treating a string as code to evaluate. There are two built-in functions for dynamic code execution:

- **eval()** for expression
- **exec()** for code

```
x = eval('2 ** 10')
print(x)   # 1024
```

```
import math

code = '''
def area_of_sphere(r):
    return 4 * math.pi * r ** 2
'''

context = {}
```

```python
context['math'] = math
exec(code, context)  # define the function area_of_sphere
```

After the exec() call the context dictionary contains a key called `area_of_sphere` whose value is the area_of_sphere() function.

```python
area_of_sphere = context['area_of_sphere']
area = area_of_sphere(5)
print(area)  # 314.1592653589793
```

2.3 Decorator

A decorator is a function that takes a function or method as its sole argument and returns a new function or method that incorporates the docorated function or method with some additional functionality added.

```python
@positive_result
def discriminant(a, b, c):
    return b ** 2 - 4 * a * c


def positive_result(function):
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >= 0"
        return result

    wrapper.__name__ = function.__name__
    wrapper.__doc__ = function.__doc__
    return wrapper
```

Here positive_result is a decorator. It define a new local function (here wrapper()) tha calls the original function. The wrapper finishes by returning the result computed by the wrapped function. After creating the wrapper, we set its name and docstring to those of the original function. This helps with introspection, since we want error messages to mention the name of the original function, not the wrapper. Finally, we return the wrapper function.

We can also use the `functools` module's @functools.wraps decorator to set the function name and docstring to its original ones. Here's the code:

```python
import functools


def positive_result(function):
    @functools.wraps(function)
    def wrapper(*args, **kwargs):
        result = function(*args, **kwargs)
        assert result >= 0, function.__name__ + "() result isn't >=0"
        return result

    return wrapper
```

Here's an example code to create a decorator with parameters.

```python
@bounded(0, 100)
def percent(amount, total):
    return (amount / total) * 100

def bounded(minimum, maximum):
    def decorator(function):
        @functools.wraps(function)
        def wrapper(*args, **kwargs):
            result = function(*args, **kwargs)
            if result < minimum:
                return minimum
            elif result > maximum:
                return maximum
            return result

        return wrapper

    return decorator
```

Decorators can also be used for logging. This is a very neat and efficient way for logging.

2.4 Function annotation

```python
def function_name(par1: exp1, par2: exp2, ..., parN: expN) -> rexp:
    suite
```

Every colon expression part (: expN) and the return expression part (-> rexp) are optional annotations.

If annotations are present they are added to the function's `__annotations__` dictionary. If they are not present this dictionary is empty. The dictionary's keys are the parameter names, and the value are the corresponding expressions. Annotations have no special significance to Python. What we can do depends on how we use the `__annotations__` dictionary. For example to do type checking.

2.5 Partial function

Partial function application is the creation of a function from an existing function and some arguments to produce a new function that does what the original function did, but with some arguments fixed so that callers don't have to pass them. Here's a very simple example:

```python
enumerate1 = functools.partial(enumerate, start=1)
for lino, line in enumerate1(lines):
    process_line(lino, line)
```

Using partial function application can simplify our code, especially when we want to call the same functions with the same arguments again and again. For example:

```python
reader = functools.partial(open, mode='rt', encoding='utf8')
writer = functools.partial(open, mode='wt', encoding='utf8')
```

```python
Conv2D_common = functools.partial(Conv2D, kernel_size=(3, 3), activation='relu',
    padding='same')
```

```

h = Conv2D_common(256)(input_layer)
h = Conv2D_common(64)(h)

# The same full code is:
# h = Conv2D(256, (3, 3), activation='relu', padding='same')(input_layer)
# h = Conv2D(64, (3, 3), activation='relu', padding='same')(h)
```

2.6 Coroutines

Coroutines are functions whose processing can be suspended and resumed at specific points. So, typically, a coroutine will execute up to a certain statement, then suspend execution while waiting for some data. At this point other parts of the program can continue to execute (usually other coroutines that aren't suspended). Once the data is received the coroutine resumes from the point it was suspended, performs processing (presumably based on the data it got), and possibly sending its results to another coroutine.

In Python, a coroutine is a function that takes its input from a **yield** expression. It may also send results to a receiver function (which itself must be a coroutine). Whenever a coroutine reaches a **yield** expression it suspends waiting for data; and once it receives data, it resumes execution from that point.

Chapter 3

Object oriented programming

3.1 Attribute controlling

The `__slot__` attribute in a class controls whether we can add or remove attributes to and from a class instance.

When a class is created without the use of `__slot__`, behind the scences Python creates a private dictionary called `__dict__` for each **instance**, and this dictionary holds the instances's data attributes. This is why we can add or remove attributes from object.

If we only need objects where we access the original attributes and don't need to add or remove attributes, we can create classes that don't have a `__dict__`. This is achieved simply by defining a class attribute called `__slot__` whose value is a tuple of **attribute** names. (Here attributes is different from method.) Each object of such a class will have attributes of the specified names and no `__dict__`; no attributes can be added or removed from such classes.

3.2 Special methods

Special methods are methods start and end with two underscores. We can use different but convenient usage syntax to access those special methods.

Special Method	**Usage**	**Description**
__delattr__(self, name)	del x.n	Deletes object x's n attibute
__dir__(self)	dir(x)	Returns a list of x's attibute names
__getattr(self, name)	x.n	Returns the value of object x's n attribute if it isn't found directly
__setattr__(self, name, value)	x.n = v	Sets object x's n attribute's value to v

Tables 3.1

3.3 Functors

A **functor** is an object that can be called. Any class that has a **__call__()** special method is a functor.

The key benefit that functors offer is that they can maintain some state information.

```python
class Strip:
    def __init__(self, characters):
        self.characters = characters

    def __call__(self, string):
        return string.strip(self.characters)

strip_punctuation = Strip(',;:.!?')
print(strip_punctuation('Mingming Li!'))
```

The **Strip** class maintain a state of `characters` and when the class instance is called it can use this state.

This can also be implemented using a functor (also a closure[1]) but without using a class.

```python
def make_strip_function(characters):
    def strip_function(string):
        return string.strip(characters)

    return strip_function

strip_punctuation = make_strip_function(',;:.!?')
print(strip_punctuation('Mingming Li!'))
```

3.4 Context manager

The syntax for using context managers is

```
with expression as variable
    suite
```

Context managers allow us to simplify code by ensuring that certain operations are performed before and after a particular block of code is executed. The behavior is achieved because context managers define two special methods, **__enter__()** and **__exit__()**. When a context manager is created in a **with** statement its **__enter__()** method is automatically called, and when the context manager goes out of scope after its with statement its **__exit__()** method is automatically called.

If we want to create a custom context manager we must create a class that provides two methods: **__enter__()** and **__exit__()**. Whenever a **with** statement is used on an instance of such a class, the **__enter__()** method is called and the return value is used for the **as variable** (or thrown away if there isn't one). When control leaves the scope of the **with** statement the **__exit__()** method is called (with details of an exception if one has occurred passed as arguments).

3.5 Descriptors

Descriptors are **classes** which provide access control for the attributes of other **classes**. Any class that implements one or more of the descriptor special methods, **__get__()**, **__set__()**, and **__delete__()**, is called (and can be used as) a descriptor.

[1]A closure is a function or method that captures some external state.

```python
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = '_' + name

    def __get__(self, obj, objtype=None):
        value = getattr(obj, self.private_name)
        logging.info('Accessing %r giving %r', self.public_name, value)
        return value

    def __set__(self, obj, value):
        logging.info('Updating %r to %r', self.public_name, value)
        setattr(obj, self.private_name, value)

class Person:

    name = LoggedAccess()                # First descriptor instance
    age = LoggedAccess()                 # Second descriptor instance

    def __init__(self, name, age):
        self.name = name                 # Calls the first descriptor
        self.age = age                   # Calls the second descriptor

    def birthday(self):
        self.age += 1
```

The obj in LoggedAccess is the instance of Person.

When a class uses descriptors, it can inform each descriptor about which variable name was used. In this example, the Person class has two descriptor instances, name and age. When the Person class is defined, it makes a callback to __set_name__() in LoggedAccess so that the field names can be recorded, giving each descriptor its own public_name and private_name.

An interactive session shows that the Person class has called __set_name__() so that the field names would be recorded. Here we call vars() to look up the descriptor without triggering it:

```
>>> vars(vars(Person)['name'])
{'public_name': 'name', 'private_name': '_name'}
>>> vars(vars(Person)['age'])
{'public_name': 'age', 'private_name': '_age'}
```

The new class now logs access to both name and age:

```
>>> pete = Person('Peter P', 10)
INFO:root:Updating 'name' to 'Peter P'
INFO:root:Updating 'age' to 10
>>> kate = Person('Catherine C', 20)
```

```
INFO:root:Updating 'name' to 'Catherine C'
INFO:root:Updating 'age' to 20
```

The two Person instances contain only the private names:

```
>>> vars(pete)
{'_name': 'Peter P', '_age': 10}
>>> vars(kate)
{'_name': 'Catherine C', '_age': 20}
```

Descriptors get invoked by the dot operator during attribute lookup. If a descriptor is accessed indirectly with `vars(some_class)[descriptor_name]`, the descriptor instance is returned without invoking it.

Descriptors only work when used as class variables. When put in instances, they have no effect.

The main motivation for descriptors is to provide a hook allowing objects stored in class variables to control what happens during attribute lookup. Traditionally, the calling class controls what happens during lookup. Descriptors invert that relationship and allow the data being looked-up to have a say in the matter.

Descriptors are used throughout the language. It is how functions turn into bound methods. Common tools like **classmethod()**, **staticmethod()**, **property()**, and **functools.cached_property()** are all implemented as descriptors.

3.6 Class decorators

Class decorators takes a class object and return a modified version of the class they decorate.

In the following example, only **__lt__()** special method is supplied, and the other comparison method is created by the class decorator.

```
@complete_comparisons
class FuzzyBool:
    def __init__(self, value=0.0):
        self.__value = value if 0.0 <= value <= 1.0 else 0.0

    def __lt__(self, other):
        return self.__value < other.__value
```

Here’s the decorator:

```
def complete_comparisons(cls):
    assert cls.__lt__ is not object.__lt__, (
        f'{cls.__name__} must define < and ideally =='
    )
    if cls.__eq__ is object.__eq__:
        cls.__eq__ = lambda self, other: (
            not (cls.__lt__(self, other) or cls.__lt__(other, self))
        )
    cls.__ne__ = lambda self, other: not cls.__eq__(self, other)
    cls.__gt__ = lambda self, other: cls.__lt__(other, self)
    cls.__le__ = lambda self, other: not cls.__lt__(other, self)
    cls.__ge__ = lambda self, other: not cls.__lt__(self, other)
```

3.7 Abstract base classes

An **abstract base class** (ABC) is a class that cannot be used to create objects. Instead, the purpose of such classes is to define interface, that is, to in effect list the methods and properites that classes that inherit the abstract base class must provide. Abstract base classes are classes that have at least one abstract method or property. All ABCs must have a metaclass of `abc.ABCMeta` (from the `abc` module), or from one of its subclasses. The `abc` module provides the metaclass `ABCMeta` for defining ABCs and a helper class `ABC` to alternatively define ABCs through inheritance. `ABC` is a helper class that has `ABCMeta` as its metaclass. The `abc` model provides the **abstractmethod** decorator to define abstract method and property.

```python
class C(ABC):
    @abstractmethod
    def my_abstract_method(self, arg1):
        ...
    @classmethod
    @abstractmethod
    def my_abstract_classmethod(cls, arg2):
        ...
    @staticmethod
    @abstractmethod
    def my_abstract_staticmethod(arg3):
        ...

    @property
    @abstractmethod
    def my_abstract_property(self):
        ...
    @my_abstract_property.setter
    @abstractmethod
    def my_abstract_property(self, val):
        ...

    @abstractmethod
    def _get_x(self):
        ...
    @abstractmethod
    def _set_x(self, val):
        ...
    x = property(_get_x, _set_x)
```

3.8 Multiple inheritance

Multiple inheritance is where one class inherits from two or more other classes. One problem is that multiple inheritance can lead to the same class being inherited more than once and this means that the version of a method that is called depends on the method resolution order, which potentially makes classes that use multiple inheritance somewhat fragile.

Multiple inheritance can generally ba avoided by:

- Using single inheritance and setting a metaclass if we want to support an additional API

- Using mutiple inheritance with one concrete class and one or more abstract base classes for addtional APIs
- Using single inheritance and aggregate instances of other classes

However multiple inheritance can be convenient and works well when the inheritance classes have no overlapping APIs.

3.9 Metaclasses

A metaclass is used to create classes, just as classes are used to create instances.

3.9.1 Register

The simplest use of metaclasses is to make custom classes fit into Python's standard ABC hierarchy.

```
import collections


class SortedList:
    pass


collections.abc.Sequence.register(SortedList)
```

Registering a class like this makes it a **virtual subclass**. A virtual subclass reports that it is a subclass of the class or classes it it registed with, but does not inherit any data or methods from any of the classes it is registered with. Registering a class like this provides a promise that the class provides the API of the classes it is registered with, but does not provide any guarantee that it will honor its promise.

3.9.2 Guarantee

We can use a metaclass to provide both a promise and a guarantee about a class's API.

Suppose we want to create a group of classes that all provide load() and save() methods. We can do this by creating a class that when used as a metaclass, checks that these methods are present:

```
# API guarantee
class LoadableSavable(type):
    def __init__(cls, classname, bases, dictionary):
        super().__init__(classname, bases, dictionary)

        assert (
            hasattr(cls, 'load') and
            isinstance(getattr(cls, 'load')), collections.abc.Callable
        ), "class '" + classname + "' must provide a load() method"
        assert (
                hasattr(cls, 'save') and
                isinstance(getattr(cls, 'save'), collections.abc.Callable)
        ), "class '" + classname + "' must provide a save() method"
```

3.9.3 Modify

We can use metaclasses to change the classes use them. If the change involves the name, base classes, or dictionary of the class being created, the we need to reimplement the metaclass's **__new__()** method; but for other changes, such as adding methods or data attributes, reimplementing **__init__()** is sufficient.

Chapter 4

Processes and threading

There are two main approaches to spreading the workload:

- ♥ multiple processes
- ♥ multiple threads

	advantage	**disadvantage**
multiple processes	each process runs independently	communication and data sharing can be inconvenient
multiple threads	can communicate simply by data sharing	more complex than single-threaded program

Tables 4.1: multiple processes and multiple threads

Bibliography

[1] Aston Zhang et al. *Dive into Deep Learning*. 2021.

[2] Debra Cameron et al. *Learning GNU Emacs*. O'Reilly Media, Inc., 2004.

[3] Ian Goodfellow et al. *Deep Learning*. 2016.

[4] Yann LeCun et al. Gradientbased learning applied to document recognition. *Proceedings of the IEEE*, 1998.

[5] D.E. Knuth. *The TEXbook*. Addison Wesley, 1986.

[6] D.E. Knuth. Typesetting concrete mathematics. *TUGboat*, 10(1):31–36, April 1989.

[7] Adrian Rosebrock. *Practical Python and OpenCV*. PyImageSearch, 2016.

[8] Adrian Rosebrock. *Deep Learning for Computer Vision with Python*. PyImageSearch, 2017.