9. Clases

Las clases proveen una forma de empaquetar datos y funcionalidad juntos. Al crear una nueva clase, se crea un nuevo *tipo* de objeto, permitiendo crear nuevas *instancias* de ese tipo. Cada instancia de clase puede tener atributos adjuntos para mantener su estado. Las instancias de clase también pueden tener métodos (definidos por su clase) para modificar su estado.

Comparado con otros lenguajes de programación, el mecanismo de clases de Python agrega clases con un mínimo de nuevas sintaxis y semánticas. Es una mezcla de los mecanismos de clases encontrados en C++ y Modula-3. Las clases de Python proveen todas las características normales de la Programación Orientada a Objetos: el mecanismo de la herencia de clases permite múltiples clases base, una clase derivada puede sobre escribir cualquier método de su(s) clase(s) base, y un método puede llamar al método de la clase base con el mismo nombre. Los objetos pueden tener una cantidad arbitraria de datos de cualquier tipo. Igual que con los módulos, las clases participan de la naturaleza dinámica de Python: se crean en tiempo de ejecución, y pueden modificarse luego de la creación.

En terminología de C++, normalmente los miembros de las clases (incluyendo los miembros de datos), son *públicos* (excepto ver abajo Variables privadas), y todas las funciones miembro son *virtuales*. Como en Modula-3, no hay atajos para hacer referencia a los miembros del objeto desde sus métodos: la función método se declara con un primer argumento explícito que representa al objeto, el cual se provee implícitamente por la llamada. Como en Smalltalk, las clases mismas son objetos. Esto provee una semántica para importar y renombrar. A diferencia de C++ y Modula-3, los tipos de datos integrados pueden usarse como clases base para que el usuario los extienda. También, como en C++ pero a diferencia de Modula-3, la mayoría de los operadores integrados con sintaxis especial (operadores aritméticos, de sub-índice, etc.) pueden volver a ser definidos por instancias de la clase.

(Sin haber una terminología universalmente aceptada sobre clases, haré uso ocasional de términos de Smalltalk y C++. Usaría términos de Modula-3, ya que su semántica orientada a objetos es más cercana a Python que C++, pero no espero que muchos lectores hayan escuchado hablar de él.)

9.1. Unas palabras sobre nombres y objetos

Los objetos tienen individualidad, y múltiples nombres (en muchos ámbitos) pueden vincularse al mismo objeto. Esto se conoce como *aliasing* en otros lenguajes. Normalmente no se aprecia esto a primera vista en Python, y puede ignorarse sin problemas cuando se maneja tipos básicos inmutables (números, cadenas, tuplas). Sin embargo, el *aliasing*, o renombrado, tiene un efecto posiblemente sorpresivo sobre la semántica de código Python que involucra objetos mutables como listas, diccionarios, y la mayoría de otros tipos. Esto se usa normalmente para beneficio del programa, ya que los renombres funcionan como punteros en algunos aspectos. Por ejemplo, pasar un objeto es barato ya que la implementación solamente pasa el puntero; y si una función modifica el objeto que fue pasado, el que la llama verá el cambio; esto elimina la necesidad de tener dos formas diferentes de pasar argumentos, como en Pascal.

9.2. Ámbitos y espacios de nombres en Python

Antes de ver clases, primero debo decirte algo acerca de las reglas de ámbito de Python. Las definiciones de clases hacen unos lindos trucos con los espacios de nombres, y necesitás saber cómo funcionan los alcances y espacios de nombres para entender por completo cómo es la cosa. De paso, los conocimientos en este tema son útiles para cualquier programador Python avanzado.

Comencemos con unas definiciones.

desempeño), y puede cambiar en el futuro. Como ejemplos de espacios de nombres tenés: el conjunto de nombres incluidos (conteniendo funciones como `abs()`, y los nombres de excepciones integradas); los nombres globales en un módulo; y los nombres locales en la invocación a una función. Lo que es importante saber de los espacios de nombres es que no hay relación en absoluto entre los nombres de espacios de nombres distintos; por ejemplo, dos módulos diferentes pueden tener definidos los dos una función `maximizar` sin confusión; los usuarios de los módulos deben usar el nombre del módulo como prefijo.

Por cierto, yo uso la palabra *atributo* para cualquier cosa después de un punto; por ejemplo, en la expresión `z.real`, `real` es un atributo del objeto `z`. Estrictamente hablando, las referencias a nombres en módulos son referencias a atributos: en la expresión `modulo.funcion`, `modulo` es un objeto módulo y `funcion` es un atributo de éste. En este caso hay una relación directa entre los atributos del módulo y los nombres globales definidos en el módulo: ¡están compartiendo el mismo espacio de nombres! [1]

Los atributos pueden ser de sólo lectura, o de escritura. En el último caso es posible la asignación a atributos. Los atributos de módulo pueden escribirse: `modulo.la_respuesta = 42`. Los atributos de escritura se pueden borrar también con la declaración `del`. Por ejemplo, `del modulo.la_respuesta` va a eliminar el atributo `la_respuesta` del objeto con nombre `modulo`.

Los espacios de nombres se crean en diferentes momentos y con diferentes tiempos de vida. El espacio de nombres que contiene los nombres incluidos se crea cuando se inicia el intérprete, y nunca se borra. El espacio de nombres global de un módulo se crea cuando se lee la definición de un módulo; normalmente, los espacios de nombres de módulos también duran hasta que el intérprete finaliza. Las instrucciones ejecutadas en el nivel de llamadas superior del intérprete, ya sea desde un script o interactivamente, se consideran parte del módulo llamado `__main__`, por lo tanto tienen su propio espacio de nombres global. (Los nombres incluidos en realidad también viven en un módulo; este se llama `builtins`.)

El espacio de nombres local a una función se crea cuando la función es llamada, y se elimina cuando la función retorna o lanza una excepción que no se maneje dentro de la función. (Podríamos decir que lo que pasa en realidad es que ese espacio de nombres se «olvida».) Por supuesto, las llamadas recursivas tienen cada una su propio espacio de nombres local.

Un *ámbito* es una región textual de un programa en Python donde un espacio de nombres es accesible directamente. «Accesible directamente» significa que una referencia sin calificar a un nombre intenta encontrar dicho nombre dentro del espacio de nombres.

Aunque los alcances se determinan de forma estática, se utilizan de forma dinámica. En cualquier momento durante la ejecución, hay 3 o 4 ámbitos anidados cuyos espacios de nombres son directamente accesibles:

- el alcance más interno, que es inspeccionado primero, contiene los nombres locales
- los alcances de las funciones que encierran a la función actual, que son inspeccionados a partir del alcance más cercano, contienen nombres no locales, pero también no globales
- el penúltimo alcance contiene nombres globales del módulo actual
- el alcance más externo (el último inspeccionado) es el espacio de nombres que contiene los nombres integrados

Si un nombre se declara como global, entonces todas las referencias y asignaciones al mismo van directo al ámbito intermedio que contiene los nombres globales del módulo. Para reasignar nombres encontrados afuera del ámbito más interno, se puede usar la declaración `nonlocal`; si no se declara nonlocal, esas variables serán de sólo lectura (un intento de escribir a esas variables simplemente crea una *nueva* variable local en el ámbito interno, dejando intacta la variable externa del mismo nombre).

Habitualmente, el ámbito local referencia los nombres locales de la función actual. Fuera de una función, el ámbito local referencia al mismo espacio de nombres que el ámbito global: el espacio de nombres del módulo. Las definiciones de clases crean un espacio de nombres más en el ámbito local.

Es importante notar que los alcances se determinan textualmente: el ámbito global de una función definida en un módulo es el espacio de nombres de ese módulo, no importa desde dónde o con qué alias se llame a la función. Por otro lado, la búsqueda de nombres se hace dinámicamente, en tiempo de ejecución; sin embargo, la definición del lenguaje está evolucionando a hacer resolución de nombres estáticamente, en tiempo de «compilación», ¡así que no te confíes de la resolución de nombres dinámica! (De hecho, las variables locales ya se determinan estáticamente.)

Una peculiaridad especial de Python es que, si no hay una declaración `global` o `nonlocal` en efecto, las asignaciones a nombres siempre van al ámbito interno. Las asignaciones no copian datos, solamente asocian nombres a objetos. Lo mismo cuando se borra: la declaración `del x` quita la asociación de `x` del espacio de nombres referenciado por el ámbito local. De hecho, todas las operaciones que introducen nuevos nombres usan el ámbito local: en particular, las instrucciones `import` y las definiciones de funciones asocian el módulo o nombre de la función al espacio de nombres en el ámbito local.

La declaración `global` puede usarse para indicar que ciertas variables viven en el ámbito global y deberían reasignarse allí; la declaración `nonlocal` indica que ciertas variables viven en un ámbito encerrado y deberían reasignarse allí.

9.2.1. Ejemplo de ámbitos y espacios de nombre

Este es un ejemplo que muestra como hacer referencia a distintos ámbitos y espacios de nombres, y cómo las declaraciones `global` y `nonlocal` afectan la asignación de variables:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

El resultado del código ejemplo es:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Notá como la asignación *local* (que es el comportamiento normal) no cambió la vinculación de *algo* de *prueba_ambitos*. La asignación `nonlocal` cambió la vinculación de *algo* de *prueba_ambitos*, y la asignación `global` cambió la vinculación a nivel de módulo.

9.3. Un primer vistazo a las clases

Las clases introducen un poquito de sintaxis nueva, tres nuevos tipos de objetos y algo de semántica nueva.

9.3.1. Sintaxis de definición de clases

La forma más sencilla de definición de una clase se ve así:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Las definiciones de clases, al igual que las definiciones de funciones (instrucciones `def`) deben ejecutarse antes de que tengan efecto alguno. (Es concebible poner una definición de clase dentro de una rama de un `if`, o dentro de una función.)

En la práctica, las declaraciones dentro de una clase son definiciones de funciones, pero otras declaraciones son permitidas, y a veces resultan útiles; veremos esto más adelante. Las definiciones de funciones dentro de una clase normalmente tienen una lista de argumentos peculiar, dictada por las convenciones de invocación de métodos; a esto también lo veremos más adelante.

Cuando se ingresa una definición de clase, se crea un nuevo espacio de nombres, el cual se usa como ámbito local; por lo tanto, todas las asignaciones a variables locales van a este nuevo espacio de nombres. En particular, las definiciones de funciones asocian el nombre de las funciones nuevas allí.

Cuando una definición de clase se finaliza normalmente se crea un *objeto clase*. Básicamente, este objeto envuelve los contenidos del espacio de nombres creado por la definición de la clase; aprenderemos más acerca de los objetos clase en la sección siguiente. El ámbito local original (el que tenía efecto justo antes de que ingrese la definición de la clase) es restablecido, y el objeto clase se asocia allí al nombre que se le puso a la clase en el encabezado de su definición (`Clase` en el ejemplo).

9.3.2. Objetos clase

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación.

Para *hacer referencia a atributos* se usa la sintaxis estándar de todas las referencias a atributos en Python: `objeto.nombre`. Los nombres de atributo válidos son todos los nombres que estaban en el espacio de nombres de la clase cuando ésta se creó. Por lo tanto, si la definición de la clase es así:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

…entonces `MiClase.i` y `MiClase.f` son referencias de atributos válidas, que retornan un entero y un objeto función respectivamente. Los atributos de clase también pueden ser asignados, o sea que podés cambiar el valor de `MiClase.i` mediante asignación. `__doc__` también es un atributo válido, que retorna la documentación asociada a la clase: `"Simple clase de ejemplo"`.

La *instanciación* de clases usa la notación de funciones. Hacé de cuenta que el objeto de clase es una fun-

```
x = MyClass()
```

…crea una nueva *instancia* de la clase y asigna este objeto a la variable local `x`.

La operación de instanciación («llamar» a un objeto clase) crea un objeto vacío. Muchas clases necesitan crear objetos con instancias en un estado inicial particular. Por lo tanto una clase puede definir un método especial llamado `__init__()`, de esta forma:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Por supuesto, el método `__init__()` puede tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método `__init__()`. Por ejemplo,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Objetos instancia

Ahora, ¿Qué podemos hacer con los objetos instancia? La única operación que es entendida por los objetos instancia es la referencia de atributos. Hay dos tipos de nombres de atributos válidos, atributos de datos y métodos.

Los *atributos de datos* se corresponden con las «variables de instancia» en Smalltalk, y con las «variables miembro» en C++. Los atributos de datos no necesitan ser declarados; tal como las variables locales son creados la primera vez que se les asigna algo. Por ejemplo, si `x` es la instancia de `MiClase` creada más arriba, el siguiente pedazo de código va a imprimir el valor `16`, sin dejar ningún rastro:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

El otro tipo de atributo de instancia es el *método*. Un método es una función que «pertenece a» un objeto. En Python, el término método no está limitado a instancias de clase: otros tipos de objetos pueden tener métodos también. Por ejemplo, los objetos lista tienen métodos llamados append, insert, remove, sort, y así sucesivamente. Pero, en la siguiente explicación, usaremos el término método para referirnos exclusivamente a métodos de objetos instancia de clase, a menos que se especifique explícitamente lo contrario.

Los nombres válidos de métodos de un objeto instancia dependen de su clase. Por definición, todos los atributos de clase que son objetos funciones definen métodos correspondientes de sus instancias. Entonces, en

lo es, dado que `MiClase.i` no lo es. Pero `x.f` no es la misma cosa que `MiClase.f`; es un *objeto método*, no un objeto función.

9.3.4. Objetos método

Generalmente, un método es llamado luego de ser vinculado:

```
x.f()
```

En el ejemplo `MiClase`, esto retorna la cadena `'hola mundo'`. Pero no es necesario llamar al método justo en ese momento: `x.f` es un objeto método, y puede ser guardado y llamado más tarde. Por ejemplo:

```
xf = x.f
while True:
    print(xf())
```

…continuará imprimiendo `hola mundo` hasta el fin de los días.

¿Qué sucede exactamente cuando un método es llamado? Debés haber notado que `x.f()` fue llamado más arriba sin ningún argumento, a pesar de que la definición de función de `f()` especificaba un argumento. ¿Qué pasó con ese argumento? Seguramente Python lanza una excepción cuando una función que requiere un argumento es llamada sin ninguno, aún si el argumento no es utilizado…

De hecho, tal vez hayas adivinado la respuesta: lo que tienen de especial los métodos es que el objeto es pasado como el primer argumento de la función. En nuestro ejemplo, la llamada `x.f()` es exactamente equivalente a `MiClase.f(x)`. En general, llamar a un método con una lista de *n* argumentos es equivalente a llamar a la función correspondiente con una lista de argumentos que es creada insertando el objeto del método antes del primer argumento.

Si todavía no entiendes como funcionan los métodos, una mirada a su implementación quizás pueda aclarar dudas. Cuando un atributo sin datos de una instancia es referenciado, la clase de la instancia es accedida. Si el nombre indica un atributo de clase válido que sea un objeto función, se crea un objeto método empaquetando (apunta a) la instancia y al objeto función, juntados en un objeto abstracto: este es el objeto método. Cuando el objeto método es llamado con una lista de argumentos, se crea una nueva lista de argumentos a partir del objeto instancia y la lista de argumentos. Finalmente el objeto función es llamado con esta nueva lista de argumentos.

9.3.5. Variables de clase y de instancia

En general, las variables de instancia son para datos únicos de cada instancia y las variables de clase son para atributos y métodos compartidos por todas las instancias de la clase:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
```

```
>>> e.name                  # unique to e
'Buddy'
```

Como se vio en Unas palabras sobre nombres y objetos, los datos compartidos pueden tener efectos inesperados que involucren objetos mutable como ser listas y diccionarios. Por ejemplo, la lista *trucos* en el siguiente código no debería ser usada como variable de clase porque una sola lista sería compartida por todos las instancias de *Perro*:

```
class Dog:

    tricks = []             # mistaken use of a class variable

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

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

El diseño correcto de esta clase sería usando una variable de instancia:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4. Algunas observaciones

Si el mismo nombre de atributo aparece tanto en la instancia como en la clase, la búsqueda del atributo prioriza la instancia:

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
```

```
>>> print(w2.purpose, w2.region)
storage east
```

A los atributos de datos los pueden hacer referencia tanto los métodos como los usuarios («clientes») ordinarios de un objeto. En otras palabras, las clases no se usan para implementar tipos de datos abstractos puros. De hecho, en Python no hay nada que haga cumplir el ocultar datos; todo se basa en convención. (Por otro lado, la implementación de Python, escrita en C, puede ocultar por completo detalles de implementación y el control de acceso a un objeto si es necesario; esto se puede usar en extensiones a Python escritas en C.)

Los clientes deben usar los atributos de datos con cuidado; éstos pueden romper invariantes que mantienen los métodos si pisan los atributos de datos. Observá que los clientes pueden añadir sus propios atributos de datos a una instancia sin afectar la validez de sus métodos, siempre y cuando se eviten conflictos de nombres; de nuevo, una convención de nombres puede ahorrar un montón de dolores de cabeza.

No hay un atajo para hacer referencia a atributos de datos (¡u otros métodos!) desde dentro de un método. A mi parecer, esto en realidad aumenta la legibilidad de los métodos: no existe posibilidad alguna de confundir variables locales con variables de instancia cuando repasamos un método.

A menudo, el primer argumento de un método se llama `self` (uno mismo). Esto no es nada más que una convención: el nombre `self` no significa nada en especial para Python. Observá que, sin embargo, si no seguís la convención tu código puede resultar menos legible a otros programadores de Python, y puede llegar a pasar que un programa *navegador de clases* pueda escribirse de una manera que dependa de dicha convención.

Cualquier objeto función que es un atributo de clase define un método para instancias de esa clase. No es necesario que el la definición de la función esté textualmente dentro de la definición de la clase: asignando un objeto función a una variable local en la clase también está bien. Por ejemplo:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Ahora `f`, `g` y `h` son todos atributos de la clase `C` que hacen referencia a objetos función, y consecuentemente son todos métodos de las instancias de `C`; `h` siendo exactamente equivalente a `g`. Fijate que esta práctica normalmente sólo sirve para confundir al que lea un programa.

Los métodos pueden llamar a otros métodos de la instancia usando el argumento `self`:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Los métodos pueden hacer referencia a nombres globales de la misma manera que lo hacen las funciones comunes. El ámbito global asociado a un método es el módulo que contiene su definición. (Una clase nunca se usa como un ámbito global.) Si bien es raro encontrar una buena razón para usar datos globales en un método, hay muchos usos legítimos del ámbito global: por lo menos, las funciones y módulos importados en el ámbito global pueden usarse por los métodos, al igual que las funciones y clases definidas en él. Habitualmente, la clase que contiene el método está definida en este ámbito global, y en la siguiente sección veremos algunas buenas razones por las que un método querría hacer referencia a su propia clase.

Todo valor es un objeto, y por lo tanto tiene una *clase* (también llamado su *tipo*). Ésta se almacena como `objeto.__class__`.

9.5. Herencia

Por supuesto, una característica del lenguaje no sería digna del nombre «clase» si no soportara herencia. La sintaxis para una definición de clase derivada se ve así:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

El nombre `ClaseBase` debe estar definido en un ámbito que contenga a la definición de la clase derivada. En el lugar del nombre de la clase base se permiten otras expresiones arbitrarias. Esto puede ser útil, por ejemplo, cuando la clase base está definida en otro módulo:

```
class DerivedClassName(modname.BaseClassName):
```

La ejecución de una definición de clase derivada procede de la misma forma que una clase base. Cuando el objeto clase se construye, se tiene en cuenta a la clase base. Esto se usa para resolver referencias a atributos: si un atributo solicitado no se encuentra en la clase, la búsqueda continúa por la clase base. Esta regla se aplica recursivamente si la clase base misma deriva de alguna otra clase.

No hay nada en especial en la instanciación de clases derivadas: `ClaseDerivada()` crea una nueva instancia de la clase. Las referencias a métodos se resuelven de la siguiente manera: se busca el atributo de clase correspondiente, descendiendo por la cadena de clases base si es necesario, y la referencia al método es válida si se entrega un objeto función.

Las clases derivadas pueden redefinir métodos de su clase base. Como los métodos no tienen privilegios especiales cuando llaman a otros métodos del mismo objeto, un método de la clase base que llame a otro método definido en la misma clase base puede terminar llamando a un método de la clase derivada que lo haya redefinido. (Para los programadores de C++: en Python todos los métodos son en efecto `virtuales`.)

Un método redefinido en una clase derivada puede de hecho querer extender en vez de simplemente reemplazar al método de la clase base con el mismo nombre. Hay una manera simple de llamar al método de la clase base directamente: simplemente llamás a `ClaseBase.metodo(self, argumentos)`. En ocasiones esto es útil para los clientes también. (Observá que esto sólo funciona si la clase base es accesible como `ClaseBase` en el ámbito global.)

Python tiene dos funciones integradas que funcionan con herencia:

- Usar `isinstance()` para verificar el tipo de una instancia: `isinstance(obj, int)` será `True` sólo si `obj.__class__` es `int` o alguna clase derivada de `int`.

- Usar `issubclass()` para verificar la herencia de clases: `issubclass(bool, int)` es `True` ya que `bool` es una subclase de `int`. Sin embargo, `issubclass(float, int)` es `False` ya que `float` no es una subclase de `int`.

9.5.1. Herencia múltiple

Python también soporta una forma de herencia múltiple. Una definición de clase con múltiples clases base se ve así:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Para la mayoría de los propósitos, en los casos más simples, podés pensar en la búsqueda de los atributos heredados de clases padres como primero en profundidad, de izquierda a derecha, sin repetir la misma clase cuando está dos veces en la jerarquía. Por lo tanto, si un atributo no se encuentra en `ClaseDerivada`, se busca en `Base1`, luego (recursivamente) en las clases base de `Base1`, y sólo si no se encuentra allí se lo busca en `Base2`, y así sucesivamente.

En realidad es un poco más complejo que eso; el orden de resolución de métodos cambia dinámicamente para soportar las llamadas cooperativas a `super()`. Este enfoque es conocido en otros lenguajes con herencia múltiple como «llámese al siguiente método» y es más poderoso que la llamada al superior que se encuentra en lenguajes con sólo herencia simple.

El ordenamiento dinámico es necesario porque todos los casos de herencia múltiple exhiben una o más relaciones en diamante (cuando se puede llegar al menos a una de las clases base por distintos caminos desde la clase de más abajo). Por ejemplo, todas las clases heredan de `object`, por lo tanto cualquier caso de herencia múltiple provee más de un camino para llegar a `object`. Para que las clases base no sean accedidas más de una vez, el algoritmo dinámico hace lineal el orden de búsqueda de manera que se preserve el orden de izquierda a derecha especificado en cada clase, que se llame a cada clase base sólo una vez, y que sea monótona (lo cual significa que una clase puede tener clases derivadas sin afectar el orden de precedencia de sus clases bases). En conjunto, estas propiedades hacen posible diseñar clases confiables y extensibles con herencia múltiple. Para más detalles mirá https://www.python.org/download/releases/2.3/mro/.

9.6. Variables privadas

Las variables «privadas» de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python. Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guión bajo (por ejemplo, `_spam`) debería tratarse como una parte no pública de la API (más allá de que sea una función, un método, o un dato). Debería considerarse un detalle de implementación y que está sujeto a cambios sin aviso.

Ya que hay un caso de uso válido para los identificadores privados de clase (a saber: colisión de nombres con nombres definidos en las subclases), hay un soporte limitado para este mecanismo. Cualquier identificador con la forma `__spam` (al menos dos guiones bajos al principio, como mucho un guión bajo al final) es textualmente reemplazado por `_nombredeclase__spam`, donde `nombredeclase` es el nombre de clase actual al que se le sacan guiones bajos del comienzo (si los tuviera). Se modifica el nombre del identificador sin importar su posición sintáctica, siempre y cuando ocurra dentro de la definición de una clase.

La modificación de nombres es útil para dejar que las subclases sobreescriban los métodos sin romper las lla-

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

El ejemplo de arriba funcionaría incluso si `MappingSubclass` introdujera un identificador `__update` ya que se reemplaza con `_Mapping__update` en la clase `Mapping` y `_MappingSubclass__update` en la clase `MappingSubclass` respectivamente.

Hay que aclarar que las reglas de modificación de nombres están diseñadas principalmente para evitar accidentes; es posible acceder o modificar una variable que es considerada como privada. Esto hasta puede resultar útil en circunstancias especiales, tales como en el depurador.

Notar que el código pasado a `exec` o `eval()` no considera que el nombre de clase de la clase que invoca sea la clase actual; esto es similar al efecto de la sentencia `global`, efecto que es de similar manera restringido a código que es compilado en conjunto. La misma restricción aplica a `getattr()`, `setattr()` y `delattr()`, así como cuando se referencia a `__dict__` directamente.

9.7. Cambalache

A veces es útil tener un tipo de datos similar al «registro» de Pascal o la «estructura» de C, que sirva para juntar algunos pocos ítems con nombre. Una definición de clase vacía funcionará perfecto:

```
class Employee:
    pass

john = Employee()  # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Algún código Python que espera un tipo abstracto de datos en particular puede frecuentemente recibir en cambio una clase que emula los métodos de aquel tipo de datos. Por ejemplo, si tenés una función que formatea algunos datos a partir de un objeto archivo, podés definir una clase con métodos `read()` y `readline()` que obtengan los datos de alguna cadena en memoria intermedia, y pasarlo como argumento.

Los objetos método de instancia tienen atributos también: `m.__self__` es el objeto instancia con el método `m()`, y `m.__func__` es el objeto función correspondiente al método.

9.8. Iteradores

Es probable que hayas notado que la mayoría de los objetos contenedores pueden ser recorridos usando una sentencia `for`:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

Este estilo de acceso es limpio, conciso y conveniente. El uso de iteradores está impregnado y unifica a Python. En bambalinas, la sentencia `for` llama a `iter()` en el objeto contenedor. La función retorna un objeto iterador que define el método `__next__()` que accede elementos en el contenedor de a uno por vez. Cuando no hay más elementos, `__next__()` lanza una excepción `StopIteration` que le avisa al bucle del `for` que hay que terminar. Podés llamar al método `__next__()` usando la función integrada `next()`; este ejemplo muestra como funciona todo esto:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Habiendo visto la mecánica del protocolo de iteración, es fácil agregar comportamiento de iterador a tus clases. Definí un método `__iter__()` que retorne un objeto con un método `__next__()`. Si la clase define `__next__()`, entonces alcanza con que `__iter__()` retorne `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
```

```
...     print(char)
...
m
a
p
s
```

9.9. Generadores

Generators son una herramienta simple y poderosa para crear iteradores. Están escritas como funciones regulares pero usan la palabra clave `yield` siempre que quieran retornar datos. Cada vez que se llama a `next()`, el generador se reanuda donde lo dejó (recuerda todos los valores de datos y qué instrucción se ejecutó por última vez). Un ejemplo muestra que los generadores pueden ser trivialmente fáciles de crear:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Todo lo que puede ser hecho con generadores también puede ser hecho con iteradores basados en clases, como se describe en la sección anterior. Lo que hace que los generadores sean tan compactos es que los métodos `__iter__()` y `__next__()` son creados automáticamente.

Otra característica clave es que las variables locales y el estado de la ejecución son guardados automáticamente entre llamadas. Esto hace que la función sea más fácil de escribir y quede mucho más claro que hacerlo usando variables de instancia tales como `self.indice` y `self.datos`.

Además de la creación automática de métodos y el guardar el estado del programa, cuando los generadores terminan automáticamente lanzan `StopIteration`. Combinadas, estas características facilitan la creación de iteradores, y hacen que no sea más esfuerzo que escribir una función regular.

9.10. Expresiones generadoras

Algunos generadores simples pueden ser escritos de manera concisa como expresiones usando una sintaxis similar a las comprensiones de listas pero con paréntesis en lugar de corchetes. Estas expresiones están hechas para situaciones donde el generador es utilizado de inmediato por la función que lo encierra. Las expresiones generadoras son más compactas pero menos versátiles que las definiciones completas de generadores y tienden a ser más amigables con la memoria que sus comprensiones de listas equivalentes.

Ejemplos:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260
```

```
>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

Notas al pie

[1] Excepto por una cosa. Los objetos módulo tienen un atributo de sólo lectura secreto llamado `__dict__` que retorna el diccionario usado para implementar el espacio de nombres del módulo; el nombre `__dict__` es un atributo pero no un nombre global. Obviamente, usar esto viola la abstracción de la implementación del espacio de nombres, y debería ser restringido a cosas como depuradores post-mortem.