Capítulo 12

Clases y objetos

12.1. Tipos compuestos definidos por el usuario

Una vez utilizados algunos de los tipos internos de Python, estamos listos para crear un tipo definido por el usuario: el `Punto`.

Piense en el concepto de un punto matemático. En dos dimensiones, un punto es dos números (coordenadas) que se tratan colectivamente como un solo objeto. En notación matemática, los puntos suelen escribirse entre paréntesis con una coma separando las coordenadas. Por ejemplo, $(0, 0)$ representa el origen, y (x, y) representa el punto x unidades a la derecha e y unidades hacia arriba desde el origen.

Una forma natural de representar un punto en Python es con dos valores en coma flotante. La cuestión es, entonces, cómo agrupar esos dos valores en un objeto compuesto. La solución rápida y burda es utilizar una lista o tupla, y para algunas aplicaciones esa podría ser la mejor opción.

Una alternativa es que el usuario defina un nuevo tipo compuesto, también llamado una **clase**. Esta aproximación exige un poco más de esfuerzo, pero tiene sus ventajas que pronto se harán evidentes.

Una definición de clase se parece a esto:

```
class Punto:
  pass
```

Las definiciones de clase pueden aparecer en cualquier lugar de un programa, pero normalmente están al principio (tras las sentencias `import`). Las reglas

sintácticas de la definición de clases son las mismas que para cualesquiera otras sentencias compuestas. (ver la Sección 4.4).

Esta definición crea una nueva clase llamada `Punto`. La sentencia **pass** no tiene efectos; sólo es necesaria porque una sentencia compuesta debe tener algo en su cuerpo.

Al crear la clase `Punto` hemos creado un nuevo tipo, que también se llama `Punto`. Los miembros de este tipo se llaman **instancias** del tipo u **objetos**. La creación de una nueva instancia se llama **instanciación**. Para instanciar un objeto `Punto` ejecutamos una función que se llama (lo ha adivinado) `Punto`:

```
blanco = Punto()
```

A la variable `blanco` se le asigna una referencia a un nuevo objeto `Punto`. A una función como `Punto` que crea un objeto nuevo se le llama **constructor**.

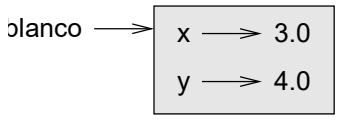
12.2. Atributos

Podemos añadir nuevos datos a una instancia utilizando la notación de punto:

```
>>> blanco.x = 3.0
>>> blanco.y = 4.0
```

Esta sintaxis es similar a la sintaxis para seleccionar una variable de un módulo, como `math.pi` o `string.uppercase`. En este caso, sin embargo, estamos seleccionando un dato de una instancia. Estos ítemes con nombre se llaman **atributos**.

El diagrama de estados que sigue muestra el resultado de esas asignaciones:



La variable `blanco` apunta a un objeto Punto, que contiene dos atributos. Cada atributo apunta a un número en coma flotante.

Podemos leer el valor de un atributo utilizando la misma sintaxis:

```
>>> print blanco.y
4.0
>>> x = blanco.x
>>> print x
3.0
```

La expresión `blanco.x` significa, "ve al objeto al que apunta `blanco` y toma el valor de `x`". En este caso, asignamos ese valor a una variable llamada `x`. No hay conflicto entre la variable `x` y el atributo `x`. El propósito de la notación de punto es identificar de forma inequívoca a qué variable se refiere.

Puede usted usar la notación de punto como parte de cualquier expresión. Así, las sentencias que siguen son correctas:

```
print '(' + str(blanco.x) + ', ' + str(blanco.y) + ')'
distanciaAlCuadrado = blanco.x * blanco.x + blanco.y * blanco.y
```

La primera línea presenta `(3.0, 4.0)`; la segunda línea calcula el valor 25.0.

Puede tentarle imprimir el propio valor de `blanco`:

```
>>> print blanco
<__main__.Punto instance at 80f8e70>
```

El resultado indica que `blanco` es una instancia de la clase `Punto` que se definió en `__main__`. `80f8e70` es el identificador único de este objeto, escrito en hexadecimal. Probablemente no es esta la manera más clara de mostrar un objeto `Punto`. En breve verá cómo cambiarlo.

> *Como ejercicio, cree e imprima un objeto* `Punto` *y luego use* `id` *para imprimir el identificador único del objeto. Traduzca el número hexadecimal a decimal y asegúrese de que coinciden.*

12.3. Instancias como parámetro

Puede usted pasar una instancia como parámetro de la forma habitual. Por ejemplo:

```
def imprimePunto(p):
  print '(' + str(p.x) + ', ' + str(p.y) + ')'
```

`imprimePunto` acepta un punto como argumento y lo muestra en formato estándar. Si llama a `imprimePunto(blanco)`, el resultado es `(3.0, 4.0)`.

> *Como ejercicio, reescriba la función* `distancia` *de la Sección 5.2 de forma que acepte dos* `Puntos` *como parámetros en lugar de cuatro números.*

12.4. Mismidad

El significado de la palabra "mismo" parece totalmente claro hasta que uno se para un poco a pensarlo, y entonces se da cuenta de que hay algo más de lo que suponía.

Por ejemplo, si dice "Pepe y yo tenemos la misma moto", lo que quiere decir es que su moto y la de usted son de la misma marca y modelo, pero que son dos motos distintas. Si dice "Pepe y yo tenemos la misma madre", quiere decir que su madre y la de usted son la misma persona[1]. Así que la idea de "identidad" es diferente según el contexto.

Cuando habla de objetos, hay una ambigüedad parecida. Por ejemplo, si dos `Puntos` son el mismo, ¿significa que contienen los mismos datos (coordenadas) o que son de verdad el mismo objeto?

Para averiguar si dos referencias se refieren al mismo objeto, utilice el operador `==`. Por ejemplo:

```
>>>   p1 = Punto()
>>>   p1.x = 3
>>>   p1.y = 4
>>>   p2 = Punto()
>>>   p2.x = 3
>>>   p2.y = 4
>>>   p1 == p2
0
```

Aunque `p1` y `p2` contienen las mismas coordenadas, no son el mismo objeto. Si asignamos `p1` a `p2`, las dos variables son alias del mismo objeto:

```
>>>   p2 = p1
>>>   p1 == p2
1
```

Este tipo de igualdad se llama **igualdad superficial** porque sólo compara las referencias, pero no el contenido de los objetos.

Para comparar los contenidos de los objetos (**igualdad profunda**) podemos escribir una función llamada `mismoPunto`:

```
def mismoPunto(p1, p2) :
  return (p1.x == p2.x) and (p1.y == p2.y)
```

[1] No todas las lenguas tienen el mismo problema. Por ejemplo, el alemán tiene palabras diferentes para los diferentes tipos de identidad. "Misma moto" en este contexto sería "gleiche Motorrad" y "misma madre" sería "selbe Mutter".

Si ahora creamos dos objetos diferentes que contienen los mismos datos podremos usar `mismoPunto` para averiguar si representan el mismo punto:

```
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = Punto()
>>> p2.x = 3
>>> p2.y = 4
>>> mismoPunto(p1, p2)
1
```

Por supuesto, si las dos variables apuntan al mismo objeto `mismoPunto` devuelve verdadero.

12.5. Rectángulos

Digamos que queremos una clase que represente un rectángulo. La pregunta es, ¿qué información tenemos que proporcionar para definir un rectángulo? Para simplificar las cosas, supongamos que el rectángulo está orientado vertical u horizontalmente, nunca en diagonal.

Tenemos varias posibilidades: podemos señalar el centro del rectángulo (dos coordenadas) y su tamaño (anchura y altura); o podemos señalar una de las esquinas y el tamaño; o podemos señalar dos esquinas opuestas. Un modo convencional es señalar la esquina superior izquierda del rectángulo y el tamaño.

De nuevo, definiremos una nueva clase:

```
class Rectangulo: # Prohibidos los acentos fuera de las cadenas!
  pass
```

Y la instanciaremos:

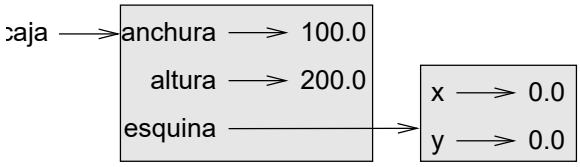
```
caja = Rectangulo()
caja.anchura = 100.0
caja.altura = 200.0
```

Este código crea un nuevo objeto `Rectangulo` con dos atributos en coma flotante. ¡Para señalar la esquina superior izquierda podemos incrustar un objeto dentro de otro!

```
caja.esquina = Punto()
caja.esquina.x = 0.0;
caja.esquina.y = 0.0;
```

El operador punto compone. La expresión `caja.esquina.x` significa "ve al objeto al que se refiere `caja` y selecciona el atributo llamado `esquina`; entonces ve a ese objeto y selecciona el atributo llamado `x`".

La figura muestra el estado de este objeto:



12.6. Instancias como valores de retorno

Las funciones pueden devolver instancias. Por ejemplo, `encuentraCentro` acepta un `Rectangulo` como argumento y devuelve un `Punto` que contiene las coordenadas del centro del `Rectangulo`:

```
def encuentraCentro(caja):
  p = Punto()
  p.x = caja.esquina.x + caja.anchura/2.0
  p.y = caja.esquina.y + caja.altura/2.0
  return p
```

Para llamar a esta función, pase `caja` como argumento y asigne el resultado a una variable:

```
>>> centro = encuentraCentro(caja)
>>> imprimePunto(centro)
(50.0, 100.0)
```

12.7. Los objetos son mudables

Podemos cambiar el estado de un objeto efectuando una asignación sobre uno de sus atributos. Por ejemplo, para cambiar el tamaño de un rectángulo sin cambiar su posición, podemos cambiar los valores de `anchura` y `altura`:

```
caja.anchura = caja.anchura + 50
caja.altura = caja.altura + 100
```

Podemos encapsular este código en un método y generalizarlo para agrandar el rectángulo en cualquier cantidad:

```
def agrandaRect(caja, danchura, daltura) :
  caja.anchura = caja.anchura + danchura
  caja.altura = caja.altura + daltura
```

Las variables `danchura` y `daltura` indican cuánto debe agrandarse el rectángulo en cada dirección. Invocar este método tiene el efecto de modificar el `Rectangulo` que se pasa como argumento.

Por ejemplo, podemos crear un nuevo `Rectangulo` llamado `bob` y pasárselo a `agrandaRect`:

```
>>> bob = Rectangulo()
>>> bob.anchura = 100.0
>>> bob.altura = 200.0
>>> bob.esquina = Punto()
>>> bob.esquina.x = 0.0;
>>> bob.esquina.y = 0.0;
>>> agrandaRect(bob, 50, 100)
```

Mientras `agrandaRect` se está ejecutando, el parámetro `caja` es un alias de `bob`. Cualquier cambio que haga a `caja` afectará también a `bob`.

> *A modo de ejercicio, escriba una función llamada* `mueveRect` *que tome un* `Rectangulo` *y dos parámetros llamados* `dx` *y* `dy`*. Tiene que cambiar la posición del rectángulo añadiendo* `dx` *a la coordenada* `x` *de* `esquina` *y añadiendo* `dy` *a la coordenada* `y` *de* `esquina`*.*

12.8. Copiado

El uso de alias puede hacer que un programa sea difícil de leer, porque los cambios hechos en un lugar pueden tener efectos inesperados en otro lugar. Es difícil estar al tanto de todas las variables a las que puede apuntar un objeto dado.

Copiar un objeto es, muchas veces, una alternativa a la creación de un alias. El módulo `copy` contiene una función llamada `copy` que puede duplicar cualquier objeto:

```
>>> import copy
>>> p1 = Punto()
>>> p1.x = 3
>>> p1.y = 4
>>> p2 = copy.copy(p1)
>>> p1 == p2
```

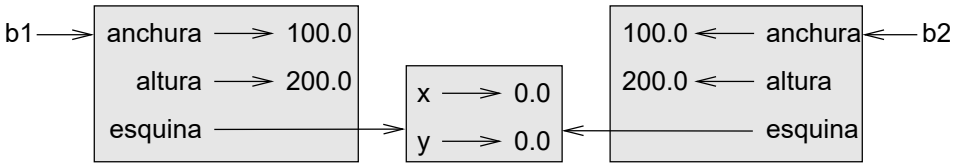
```
0
>>> mismoPunto(p1, p2)
1
```

Una vez que hemos importado el módulo `copy`, podemos usar el método `copy` para hacer un nuevo `Punto`. `p1` y `p2` no son el mismo punto, pero contienen los mismos datos.

Para copiar un objeto simple como un `Punto`, que no contiene objetos incrustados, `copy` es suficiente. Esto se llama **copiado superficial**.

Para algo como un `Rectangulo`, que contiene una referencia a un `Punto`, `copy` no lo hace del todo bien. Copia la referencia al objeto `Punto`, de modo que tanto el `Rectangulo` viejo como el nuevo apuntan a un único `Punto`.

Si creamos una caja, `b1`, de la forma habitual y entonces hacemos una copia, `b2`, usando `copy`, el diagrama de estados resultante se ve así:



Es casi seguro que esto no es lo que queremos. En este caso, la invocación de `agrandaRect` sobre uno de los `Rectangulo`s no afectaría al otro, ¡pero la invocación de `mueveRect` sobre cualquiera afectaria a ambos! Este comportamiento es confuso y propicia los errores.

Afortunadamente, el módulo `copy` contiene un método llamado `deepcopy` que copia no sólo el objeto sino también cualesquiera objetos incrustados. No le sorprenderá saber que esta operación se llama **copia profunda** (deep copy).

```
>>> b2 = copy.deepcopy(b1)
```

Ahora `b1` y `b2` son objetos totalmente independientes.

Podemos usar `deepcopy` para reescribir `agrandaRect` de modo que en lugar de modificar un `Rectangulo` existente, cree un nuevo `Rectangulo` que tiene la misma localización que el viejo pero nuevas dimensiones:

```
def agrandaRect(caja, danchura, daltura) :
  import copy
  nuevaCaja = copy.deepcopy(caja)
  nuevaCaja.anchura = nuevaCaja.anchura + danchura
  nuevaCaja.altura = nuevaCaja.altura + daltura
  return nuevaCaja
```

Como ejercicio, resscriba `mueveRect` *de modo que cree y devuelva un nuevo* `Rectangulo` *en lugar de modificar el viejo.*

12.9. Glosario

clase: Un tipo compuesto definido por el usuario. También se puede pensar en una clase como una plantilla para los objetos que son instancias de la misma.

instanciar: Crear una instancia de una clase.

instancia: Un objeto que pertenece a una clase.

objeto: Un tipo de dato compuesto que suele usarse para representar una cosa o concepto del mundo real.

constructor: Un método usado para crear nuevos objetos.

atributo: Uno de los elementos de datos con nombre que constituyen una instancia.

igualdad superficial: Igualdad de referencias, o dos referencias que apuntan al mismo objeto.

igualdad profunda: Igualdad de valores, o dos referencias que apuntan a objetos que tienen el mismo valor.

copia superficial: Copiar el contenido de un objeto, incluyendo cualquier referencia a objetos incrustados; implementada por la función `copy` del módulo `copy`.

copia profunda: Copiar el contenido de un objeto así como cualesquiera objetos incrustados, y los incrustados en estos, y así sucesivamente; implementada por la función `deepcopy` del módulo `copy`.

Capítulo 13

Clases y funciones

13.1. Hora

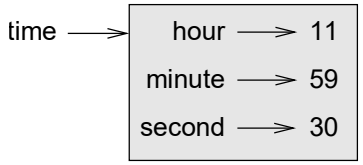
Como otro ejemplo de un tipo definido por el usuario, definiremos una clase llamada `Hora` que registra la hora del día. La definición de la clase es como sigue:

```
class Hora:
  pass
```

Podemos crear un nuevo objeto `Hora` y asignar atributos para contener las horas, minutos y segundos:

```
hora = Hora()
hora.horas = 11
hora.minutos = 59
hora.segundos = 30
```

El diagrama de estado del objeto `Hora` es así:



A modo de ejercicio, escriba una función `imprimeHora` *que acepte un objeto* `Hora` *como argumento y lo imprima en el formato* `horas:minutos:segundos`.

Como un segundo ejercicio, escriba una función booleana `despues` *que tome dos objetos* `Hora`*,* `t1` *y* `t2`*, como argumentos y devuelva verdadero (1) si* `t1` *sigue cronológicamente a* `t2` *y falso (0) en caso contrario.*

13.2. Funciones puras

En las próximas secciones, escribiremos dos versiones de una función llamada `sumaHora` que calcule la suma de dos `Hora`s. Mostrarán dos tipos de funciones: funciones puras y modificadores.

Éste es un esbozo de `sumaHora`:

```
def sumaHora(t1, t2):
  suma = Hora()
  suma.horas = t1.horas + t2.horas
  suma.minutos = t1.minutos + t2.minutos
  suma.segundos = t1.segundos + t2.segundos
  return suma
```

La función crea un nuevo objeto `Hora`, inicializa sus atributos y devuelve una referencia al nuevo objeto. A esto se le llama **función pura** porque no modifica ninguno de los objetos que se le pasan y no tiene efectos laterales, como mostrar un valor o tomar una entrada del usuario.

Aquí tiene un ejemplo de cómo usar esta función. Crearemos dos objetos `Hora`: `horaActual`, que contiene la hora actual, y `horaPan`, que contiene la cantidad de tiempo que necesita un panadero para hacer pan. Luego usaremos `sumaHora` para averiguar cuándo estará hecho el pan. Si aún no ha terminado de escribir `imprimeHora`, eche un vistazo a la Sección 14.2 antes de probar esto:

```
>>> horaActual = Hora()
>>> horaActual.horas = 9
>>> horaActual.minutos = 14
>>> horaActual.segundos =  30

>>> horaPan = Hora()
>>> horaPan.horas =  3
>>> horaPan.minutos =  35
>>> horaPan.segundos =  0

>>> horaHecho = sumaHora(horaActual, horaPan)
>>> imprimeHora(horaHecho)
```

La salida de este programa es `12:49:30`, lo que es correcto. Por otra parte, hay casos en los que el resultado no es correcto. ¿Puede imaginar uno?

El problema es que esta función no trata los casos en los que el número de segundos o minutos suma más que sesenta. Cuando ocurre eso, debemos "llevar" los segundos sobrantes a la columna de los minutos o los minutos extras a la columna de las horas.

He aquí una versión corregida de la función:

```
def sumaHora(t1, t2):
  suma = Hora()
  suma.horas = t1.horas + t2.horas
  suma.minutos = t1.minutos + t2.minutos
  suma.segundos = t1.segundos + t2.segundos

  if suma.segundos >= 60:
    suma.segundos = suma.segundos - 60
    suma.minutos = suma.minutos + 1

  if suma.minutos >= 60:
    suma.minutos = suma.minutos - 60
    suma.horas = suma.horas + 1

  return suma
```

Aunque esta función es correcta, empieza a ser grande. Más adelante sugeriremos una aproximación alternativa que nos dará un código más corto.

13.3. Modificadores

Hay veces en las que es útil que una función modifique uno o más de los objetos que recibe como parámetros. Normalmente, el llamante conserva una referencia a los objetos que pasa, así que cualquier cambio que la función haga será visible para el llamante. Las funciones que trabajan así se llaman **modificadores**.

`incremento`, que añade un número dado de segundos a un objeto `Hora`, se escribiría de forma natural como un modificador. Un esbozo rápido de la función podría ser éste:

```
def incremento(hora, segundos):
  hora.segundos = hora.segundos + segundos

  if hora.segundos >= 60:
    hora.segundos = hora.segundos - 60
    hora.minutos = hora.minutos + 1

  if hora.minutos >= 60:
    hora.minutos = hora.minutos - 60
    hora.horas = hroa.horas + 1
```

La primera línea realiza la operación básica, las restantes tratan con los casos especiales que vimos antes.

¿Es correcta esta función? ¿Qué ocurre si el parámetro `segundos` es mucho mayor que sesenta? En tal caso, no es suficiente con acarrear una vez; debemos seguir haciéndolo hasta que `segundos` sea menor que sesenta. Una solución es sustituir las sentencias `if` por sentencias `while`:

```
def incremento(hora, segundos):
  hora.segundos = hora.segundos + segundos

  while hora.segundos >= 60:
    hora.segundos = hora.segundos - 60
    hora.minutos = hora.minutos + 1

  while hora.minutos >= 60:
    hora.minutos = hora.minutos - 60
    hora.horas = hroa.horas + 1
```

Ahora esta función es correcta, pero no es la solución más eficiente.

> *Como ejercicio, reescriba esta función de modo que no contenga tantos bucles.*
>
> *Como un segundo ejercicio, reescriba* `incremento` *como una función pura, y escriba una función que llame a ambas versiones.*

13.4. ¿Qué es mejor?

Todo lo que se pueda hacer con modificadores puede hacerse también con funciones puras. En realidad, algunos lenguajes de programación sólo permiten funciones puras. Hay ciertas evidencias de que los programas que usan funciones puras son más rápidos de desarrollar y menos propensos a los errores que

los programas que usan modificadores. Sin embargo, a veces los modificadores son útiles, y en algunos casos los programas funcionales son menos eficientes.

En general, recomendamos que escriba funciones puras siempre que sea razonable hacerlo así y recurra a los modificadores sólo si hay una ventaja convincente. Este enfoque podría llamarse **estilo funcional de programación**.

13.5. Desarrollo de prototipos frente a planificación

En este capítulo mostramos una aproximación al desarrollo de programas a la que llamamos **desarrollo de prototipos**. En cada caso, escribimos un esbozo basto (o prototipo) que realizaba el cálculo básico y luego lo probamos sobre unos cuantos casos, corrigiendo los fallos tal como los encontrábamos.

Aunque este enfoque puede ser efecitvo, puede conducirnos a código que es innecesariamente complicado, ya que trata con muchos casos especiales, y poco fiable, porque es difícil saber si encontró todos los errores.

Una alternativa es el **desarrollo planificado**, en el que una comprensión del problema en profundidad puede hacer la programación mucho más fácil. En este caso, el enfoque es que un objeto `Hora` es en realidad ¡un número de tres dígitos en base 60! El componente `segundo` es la "columna de unidades", el componente `minuto` es la "columna de las sesententas" y el componente `hora` es la "columna de las tresmilseiscentenas".

Cuando escribimos `sumaHora` e `incremento`, en realidad estábamos haciendo una suma en base 60, que es por lo que debíamos acarrear de una columna a la siguiente.

Esta observación sugiere otro enfoque para el problema. Podemos convertir un objeto `Hora` en un simple número y sacar provecho del hecho de que la máquina sabe la aritmética necesaria. La siguiente función convierte un objeto `Hora` en un entero:

```
def convierteASegundos(t):
  minutos = t.horas * 60 + t.minutos
  segundos = minutos * 60 + t.segundos
  return segundos
```

Ahora, sólo necesitamos una forma de convertir un entero en un objeto `Hora`:

```
def haceHora(segundos):
  hora = Hora()
  hora.horas = segundos/3600
```

```
  segundos = segundos - hora.horas * 3600
  hora.minutos = segundos/60
  segundos = segundos - hora.minutos * 60
  hora.segundos = segundos
  return hora
```

Puede que tenga usted que pensar un poco para convencerse de que esta técnica para convertir de una base a otra es correcta. Suponiendo que está usted convencido, puede usar estas funciones para reescribir `sumaHora`:

```
def sumaHora(t1, t2):
  segundos = convierteASegundos(t1) + convierteASegundos(t2)
  return haceHora(segundos)
```

Esta versión es mucho más corta que la original, y es mucho más fácil de demostrar que es correcta (suponiendo, como es habitual, que las funciones a las que llama son correctas).

Como ejercicio, reescriba `incremento` *de la misma forma.*

13.6. Generalización

De algún modo, convertir de base 60 a base 10 y de vuelta es más difícil que simplemente manejarse con las horas. La conversión de base es más abstracta; nuestra intuición para tratar con las horas es mejor.

Pero si tenemos la comprensión para tratar las horas como números en base 60, y hacer la inversión de escribir las funciones de conversión (`convierteASegundos` y `haceHora`), obtenemos un programa que es más corto, más fácil de leer y depurar y más fiable.

También es más fácil añadir funcionalidades más tarde. Por ejemplo, imagine restar dos `Hora`s para hallar el intervalo entre ellas. La aproximación ingenua sería implementar la resta con acarreo. Con el uso de las funciones de conversión será más fácil y con mayor probabilidad, correcto.

Irónicamente, a veces hacer un poblema más complejo (o más general) lo hace más fácil (porque hay menos casos especiales y menos oportunidades de error).

13.7. Algoritmos

Cuando escribe una solución general para una clase de problemas, en contraste con una solución específica a un problema concreto, ha escrito un **algoritmo**.

Mencionamos esta palabra antes pero no la definimos con precisión. No es fácil de definir, así que probaremos un par de enfoques.

Primero, piense en algo que no es un algoritmo. Cuando usted aprendió a multiplicar números de una cifra, probablemente memorizó la tabla de multiplicar. En efecto, memorizó 100 soluciones específicas. Ese tipo de conocimiento no es algorítmico.

Pero si usted era "haragán" probablemente hizo trampa aprendiendo algunos trucos. Por ejemplo, para encontrar el producto de n por 9, puede escribir $n - 1$ como el primer dígito y $10 - n$ como el segundo dígito. Este truco es una solución general para multiplicar cualquier número de una cifra por 9. ¡Eso es un algoritmo!

De forma similar, las técnicas que aprendió para la suma y la resta con acarreo y la división larga son todas algoritmos. Una de las características de los algoritmos es que no requieren inteligencia para llevarse a cabo. Son procesos mecánicos en los que cada paso sigue al anterior de acuerdo a un conjunto simple de reglas.

En nuestra opinión, es un poco vergonzoso que los humanos pasen tanto tiempo en la escuela aprendiendo a ejecutar algoritmos que, de forma bastante similar, no exigen inteligencia.

Por otra parte, el proceso de diseñar algoritmos es interesante, un desafío intelectual y una parte primordial de lo que llamamos programar.

Algunas de las cosas que la gente hace naturalmente, sin dificultad ni pensamiento consciente, son las más difíciles de expresar algorítmicamente. Entender el lenguaje natural es un buen ejemplo. Todos lo hacemos, pero hasta el momento nadie ha sido capaz de explicar *cómo* lo hacemos, al menos no en la forma de un algoritmo.

13.8. Glosario

función pura: Una función que no modifica ninguno de los objetos que recibe como parámetros. La mayoría de las funciones puras son rentables.

modificador: Una función que modifica uno o más de los objetos que recibe como parámetros. La mayoría de los modificadores no entregan resultado.

estilo funcional de programación: Un estilo de programación en el que la mayoría de las funciones son puras.

desarrollo de prototipos: Una forma de desarrollar programas empezando con un prototipo y probándolo y mejorándolo gradualmente.

desarrollo planificado: Una forma de desarrollar programas que implica una comprensión de alto nivel del problema y más planificación que desarrollo incremental o desarrollo de prototipos.

algoritmo: Un conjunto de instrucciones para solucionar una clase de problemas por medio de un proceso mecánico sin intervención de inteligencia.

Capítulo 14

Clases y métodos

14.1. Características de la orientación a objetos

Python es un **lenguaje de programación orientado a objetos**, lo que significa que porporciona características que apoyan la **programación orientada a objetos**.

No es fácil definir la programación orientada a objetos, pero ya hemos visto algunas de sus características:

- Los programas se hacen a base de definiciones de objetos y definiciones de funciones, y la mayor parte de la computación se expresa en términos de operaciones sobre objetos.
- Cada definición de un objeto se corresponde con un objeto o concepto del mundo real, y las funciones que operan en ese objeto se corresponden con las formas en que interactúan los objetos del mundo real.

Por ejemplo, la clase `Hora` definida en el Capítulo 13 se corresponde con la forma en la que la gente registra la hora del día, y las funciones que definimos se corresponden con el tipo de cosas que la gente hace con las horas. De forma similar, las clases `Punto` y `Rectangulo` se corresponden con los conceptos matemáticos de un punto y un rectángulo.

Hasta ahora, no nos hemos aprovechado de las características que Python nos ofrece para dar soporte a la programación orientada a objetos. Hablando estrictamente, estas características no son necesarias. En su mayoría, proporcionan una sintaxis alternativa para cosas que ya hemos hecho, pero en muchos casos,

la alternativa es más concisa y expresa con más precisión a la estructura del programa.

Por ejemplo, en el programa `Hora` no hay una conexión obvia entre la definición de la clase y las definiciones de las funciones que siguen. Observando bien, se hace patente que todas esas funciones toman al menos un objeto `Hora` como parámetro.

Esta observación es la que motiva los **métodos**. Ya hemos visto varios métodos, como `keys` y `values`, que se invocan sobre diccionarios. Cada método está asociado con una clase y está pensado para invocarse sobre instancias de esa clase.

Los métodos son como las funciones, con dos diferencias:

- Los métodos se definen dentro de una definición de clase para explicitar la relación entre la clase y el método.
- La sintaxis para invocar un método es diferente de la de una llamada a una función.

En las próximas secciones tomaremos las funciones de los capítulos anteriores y las transformaremos en métodos. Esta transformación es puramente mecánica; puede hacerla simplemente siguiendo una secuencia de pasos. Si se acostumbra a convertir de una forma a la otra será capaz de elegir la mejor forma de hacer lo que quiere.

14.2. imprimeHora

En el Capítulo 13, definimos una clase llamada `Hora` y escribimos una fución llamada `imprimeHora`, que debería ser parecida a esto:

```
class Hora:
  pass

def imprimeHora(hora):
  print str(hora.horas) + ":" +
        str(hora.minutos) + ":" +
        str(hora.segundos)
```

Para llamar a esta función, pasábamos un objeto `Hora` como parámetro:

```
>>> horaActual = Hora()
>>> horaActual.horas = 9
>>> horaActual.minutos = 14
>>> horaActual.segundos = 30
>>> impriemHora(horaActual)
```

Para convertir `imprimeHora` en un método, todo lo que necesitamos hacer es mover la definición de la función al interior de la definición de la clase. Fíjese en cómo cambia el sangrado.

```
class Hora:
  def imprimeHora(hora):
    print str(hora.horas) + ":" +
          str(hora.minutos) + ":" +
          str(hora.segundos)
```

Ahora podemos invocar `imprimeHora` usando la notación de punto.

```
>>> horaActual.imprimeHora()
```

Como es habitual, el objeto sobre el que se invoca el método aparece delante del punto y el nombre del método aparece tras el punto.

El objeto sobre el que se invoca el método se asigna al primer parámetro, así que en este caso `horaActual` se asigna al parámetro `hora`.

Por convenio, el primer parámetro de un método se llama `self`. La razón de esto es un tanto rebuscada, pero se basa en una metáfora útil.

La sintaxis para la llamada a una función, `imprimeHora(horaActual)`, sugiere que la función es el agente activo. Dice algo como "¡Oye `imprimeHora`! Aquí hay un objeto para que lo imprimas".

En programación orientada a objetos, los objetos son los agentes activos. Una invocación como `horaActual.imprimeHora()` dice "¡Oye `horaActual`! ¡Imprímete!"

Este cambio de perspectiva puede ser más elegante, pero no es obvio que sea útil. En los ejemplos que hemos visto hasta ahora, puede no serlo. Pero a veces transferir la responsabilidad de las funciones a los objetos hace posible escribir funciones más versátiles, y hace más fácil mantener y reutilizar código.

14.3. Otro ejemplo

Vamos a convertir `incremento` (de la Sección 13.3) en un método. Para ahorrar espacio, dejaremos a un lado los métodos ya definidos, pero usted debería mantenerlos en su versión:

```
class Hora:
  #aquí van las definiciones anteriores de métodos...
```

```
  def incremento(self, segundos):
    self.segundos = segundos + self.segundos

    while self.segundos >= 60:
      self.segundos = self.segundos - 60
      self.minutos = self.minutos + 1

    while self.minutos >= 60:
      self.minutos = self.minutos - 60
      self.horas = self.horas + 1
```

La transformación es puramente mecánica; hemos llevado la definición del método al interior de la definición de la clase y hemos cambiado el nombre del primer parámetro.

Ahora podemos invocar `incremento` como un método.

```
horaActual.incremento(500)
```

De nuevo, el objeto sobre el que invocamos el método se asigna al primer parámetro, `self`. El segundo parámetro, `segundos` toma el valor de `500`.

> *Como ejercicio, convierta* `convertirASegundos` *(de la Sección 13.5) en un método de la clase* `Hora`.

14.4. Un ejemplo más complicado

La función `despues` es ligeramente más complicada porque opera sobre dos objetos `Hora`, no sólo sobre uno. Sólo podemos convertir uno de los parámetros en `self`; el otro se queda como está:

```
class Hora:
  #aquí van las definiciones anteriores de métodos...

  def despues(self, hora2):
    if self.horas > hora2.horas:
      return 1
    if self.horas < hora2.horas:
      return 0

    if self.minutos > hora2.minutos:
      return 1
```

```
    if self.minutos < hora2.minutos:
      return 0

    if self.segundos > hora2.segundos:
      return 1
    return 0
```

Invocamos este método sobre un objeto y pasamos el otro como argumento:

```
if horaHecho.despues(horaActual):
  print "El pan estará hecho después de empezar."
```

Casi puede leer la invocación como una mezcla de inglés y español: "Si la hora-hecho es depués de la hora-actual, entonces..."

14.5. Argumentos opcionales

Hemos visto funciones internas que toman un número variable de argumentos. Por ejemplo, `string.find` puede tomar dos, tres o cuatro argumentos.

Es posible escribir funciones definidas por el usuario con listas de argumentos opcionales. Por ejemplo, podemos modernizar nuestra propia versión de `encuentra` para que haga lo mismo que `string.find`.

Esta es la versión original de la Sección 7.7:

```
def encuentra(cad, c):
  indice = 0
  while indice < len(cad):
    if str[indice] == c:
      return indice
    indice = indice + 1
  return -1
```

Esta es la versión aumentada y mejorada:

```
def encuentra(cad, c, comienzo=0):
  indice = comienzo
  while indice < len(cad):
    if str[indice] == c:
      return indice
    indice = indice + 1
  return -1
```

El tercer parámetro, `comienzo`, es opcional porque se proporciona un valor por omisión, `0`. Si invocamos `encuentra` sólo con dos argumentos, utilizamos el valor por omisión y comenzamos por el principio de la cadena:

```
>>> encuentra("arriba", "r")
1
```

Si le damos un tercer parámetro, **anula** el predefinido:

```
>>> encuentra("arriba", "r", 2)
2
>>> encuentra("arriba", "r", 3)
-1
```

> *Como ejercicio, añada un cuarto parámetro,* `fin`*, que especifique dónde dejar de buscar.*
>
> *Cuidado: Este ejercicio tiene truco. El valor por omisión de* `fin` *debería ser* `len(cad)`*, pero eso no funciona. Los valores por omisión se evalúan al definir la función, no al llamarla. Cuando se define* `encuentra`*,* `cad` *aún no existe, así que no puede averiguar su longitud.*

14.6. El método de inicialización

El **método de inicialización** es un método especial que se invoca al crear un objeto. El nombre de este método es `__init__` (dos guiones bajos, seguidos de `init` y dos guiones bajos más). Un método de inicialización para la clase `Hora` es así:

```
class Hora:
  def __init__(self, horas=0, minutos=0, segundos=0):
    self.horas = horas
    self.minutos = minutos
    self.segundos = segundos
```

No hay conflicto entre el atributo `self.horas` y el parámetro `horas`. la notación de punto especifica a qué variable nos referimos.

Cuando invocamos el constructor `Hora`, los argumentos que damos se pasan a `init`:

```
>>> horaActual = Hora(9, 14, 30)
>>> horaActual.imprimeHora()
>>> 9:14:30
```

Como los parámetros son opcionales, podemos omitirlos:

```
>>> horaActual = Hora()
>>> horaActual.imprimeHora()
>>> 0:0:0
```

O dar sólo el primer parámetro:

```
>>> horaActual = Hora (9)
>>> horaActual.imprimeHora()
>>> 9:0:0
```

O los dos primeros parámetros:

```
>>> horaActual = Hora (9, 14)
>>> horaActual.imprimeHora()
>>> 9:14:0
```

Finalmente, podemos dar un subconjunto de los parámetros nombrándolos explicitamente:

```
>>> horaActual = Hora(segundos = 30, horas = 9)
>>> horaActual.imprimeHora()
>>> 9:0:30
```

14.7. Revisión de los Puntos

Vamos a reescribir la clase `Punto` de la Sección 12.1 con un estilo más orientado a objetos:

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

El método de inicialización toma los valores de x e y como parámetros opcionales; el valor por omisión de cada parámetro es 0.

El siguiente método, `__str__`, devuelve una representación en forma de cadena de un objeto `Punto`. Si una clase ofrece un método llamado `__str__`, se impone al comportamiento por defecto de la función interna `str` de Python.

```
>>> p = Punto(3, 4)
>>> str(p)
'(3, 4)'
```

Imprimir un objeto `Punto` invoca implícitamente a `__str__` sobre el objeto, así que definir `__str__` también cambia el comportamiento de `print`:

```
>>> p = Punto(3, 4)
>>> print p
(3, 4)
```

Cuando escribimos una nueva clase, casi siempre empezamos escribiendo `__init__`, que facilita el instanciar objetos, y `__str__`, que casi siempre es útil para la depuración.

14.8. Sobrecarga de operadores

Algunos lenguajes hacen posible cambiar la definición de los operadores internos cuando se aplican a tipos definidos por el usuario. Esta característica se llama **sobrecarga de operadores**. Es especialmente útil cuando definimos nuevos tipos matemáticos.

Por ejemplo, para suplantar al operador de suma `+` necesitamos proporcionar un método llamado `__add__`:

```
class Punto:
  # aquí van los métodos que ya habíamos definido...

  def __add__(self, otro):
    return Punto(self.x + otro.x, self.y + otro.y)
```

Como es habitual, el primer parámetro es el objeto sobre el que se invoca el método. El segundo parámetro se llama convenientemente `otro` para distinguirlo del mismo (`self`). Para sumar dos `Puntos`, creamos y devolvemos un nuevo `Punto` que contiene la suma de las coordenadas x y la suma de las coordenadas y.

Ahora, cuando apliquemos el operador `+` a objetos `Punto`, Python invocará a `__add__`:

```
>>>   p1 = Punto(3, 4)
>>>   p2 = Punto(5, 7)
>>>   p3 = p1 + p2
>>>   print p3
(8, 11)
```

La expresión `p1 + p2` equivale a `p1.__add__(p2)`, pero es obviamente más elegante.

> *Como ejercicio, añada un método* `__sub__(self, otro)` *que sobrecargue el operador resta y pruébelo.*

Hay varias formas de sobrecargar el comportamiento del operador multiplicación: definiendo un método llamado `__mul__`, o `__rmul__`, o ambos.

Si el operando a la izquierda de `*` es un `Punto`, Python invoca a `__mul__`, lo que presupone que el otro operando es también un `Punto`. Calcula el **producto interno** de dos puntos, definido según las reglas del álgebra lineal:

```
def __mul__(self, otro):
  return self.x * otro.x + self.y * otro.y
```

Si el operando a la izquierda de `*` es un tipo primitivo y el operando de la derecha es un `Punto`, Python invca a `__rmul__`, lo que realiza una **multiplicación escalar**:

```
def __rmul__(self, otro):
  return Punto(otro * self.x,  otro * self.y)
```

El resultado es un nuevo `Punto` cuyas coordenadas son múltiplos de las coordenadas originales. Si `otro` es un tipo que no se puede multiplicar por un número en coma flotante, entonces `__rmul__` causará un error.

Este ejemplo muestra ambos tipos de multiplicación:

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print p1 * p2
43
>>> print 2 * p2
(10, 14)
```

¿Qué ocurre si intentamos evaluar `p2 * 2`? Como el primer parámetro es un `Punto`, Python invoca a `__mul__` con `2` como el segundo parámetro. Dentro de `__mul__`, el programa intenta acceder a la coordenada `x` de `otro`, pero no lo consigue porque un entero no tiene atributos:

```
>>> print p2 * 2
AttributeError: 'int' object has no attribute 'x'
```

Desgraciadamente, el mensaje de error es un poco opaco. Este ejemplo muestra algunas de las difucultades de la programación orientada a objetos. A veces es difícil averiguar simplemente qué código se está ejecutando.

Para ver un ejemplo más completo de sobrecarga de operadores, vaya al Apéndice B.

14.9. Polimorfismo

La mayoría de los métodos que hemos escrito funcionan sólo para un tipo específico. Cuando usted crea un nuevo objeto, escribe métodos que operan sobre ese tipo.

Pero hay ciertas operaciones que querrá aplicar a muchos tipos, como las operaciones aritméticas de las secciones anteriores. Si muchos tipos admiten el mismo conjunto de operaciones, puede escribir funciones que trabajen sobre cualquiera de esos tipos.

Por ejemplo, la operación `multisuma` (común en álgebra lineal) toma tres parámetros; multiplica los dos primeros y luego suma el tercero. Podemos escribirla en Python así:

```
def multisuma (x, y, z):
  return x * y + z
```

Este método trabajará con cualquier valor de `x` e `y` que se pueda multiplicar y con cualquier valor de `z` que se pueda sumar al producto.

Podemos invocarlo con valores numéricos:

```
>>> multisuma (3, 2, 1)
7
```

O con `Puntos`:

```
>>> p1 = Punto(3, 4)
>>> p2 = Punto(5, 7)
>>> print multisuma (2, p1, p2)
(11, 15)
>>> print multisuma (p1, p2, 1)
44
```

En el primer caso, el `Punto` se multiplica por un escalar y luego se suma a otro `Punto`. En el segundo caso, el producto interior produce un valor numérico, así que el tercer parámetro también debe ser un valor numérico.

Una función como ésta que puede tomar parámetros con diferentes tipos se llama **polimórfica**.

Como un ejemplo más, observe el método `delDerechoYDelReves`, que imprime dos veces una lista, hacia adelante y hacia atrás:

```
def delDerechoYDelReves(derecho):
  import copy
  reves = copy.copy(derecho)
  reves.reverse()
  print str(derecho) + str(reves)
```

Como el método `reverse` es un modificador, hacemos una copia de la lista antes de darle la vuelta. Así, este método no modifica la lista que recibe como parámetro.

He aquí un ejemplo que aplica `delDerechoYDelReves` a una lista:

```
>>>   miLista = [1, 2, 3, 4]
>>>   delDerechoYDelReves(miLista)
[1, 2, 3, 4][4, 3, 2, 1]
```

Por supuesto, pretendíamos aplicar esta función a listas, así que no es sorprendente que funcione. Lo sorprendente es que pudiéramos usarla con un `Punto`.

Para determinar si una función se puede aplicar a un nuevo tipo, aplicamos la regla fundamental del polimorfismo:

> **Si todas las operaciones realizadas dentro de la función se pueden aplicar al tipo, la función se puede aplicar al tipo.**

Las operaciones del método incluyen `copy`, `reverse` y `print`.

`copy` trabaja sobre cualquier objeto, y ya hemos escrito un método `__str__` para los `Puntos`, así que todo lo que necesitamos es un método `reverse` en la clase `Punto`:

```
def reverse(self):
  self.x , self.y = self.y, self.x
```

Ahora podemos pasar `Puntos` a `delDerechoYDelReves`:

```
>>>   p = Punto(3, 4)
>>>   delDerechoYDelReves(p)
(3, 4)(4, 3)
```

El mejor tipo de polimorfismo es el que no se busca, cuando usted descubre que una función que había escrito se puede aplicar a un tipo para el que nunca la había planeado.

14.10. Glosario

lenguaje orientado a objetos: Un lenguaje que ofrece características, como clases definidas por el usuario y herencia, que facilitan la programación orientada a objetos.

programación orientada a objetos: Un estilo de programación en el que los datos y las operaciones que los manipulan están organizadas en clases y métodos.

método: Una función definida dentro de una definición de clase y que se invoca sobre instancias de esa clase.

imponer: Reemplazar una opción por omisión. Los ejemplos incluyen el reemplazo de un parámetro por omisión con un argumento particular y el reemplazo de un método por omisión proporcionando un nuevo método con el mismo nombre.

método de inicialización: Un método especial que se invoca automáticamente al crear un nuevo objeto y que inicializa los atributos del objeto.

sobrecarga de operadores: Ampliar los operadores internos (`+`, `-`, `*`, `>`, `<`, etc.) de modo que trabajen con tipos definidos por el usuario.

producto interno: Una operación definida en álgebra lineal que multiplica dos `Puntos` y entrega un valor numérico.

multiplicación escalar: Una operación definida en álgebra lineal que multiplica cada una de las coordenadas de un `Punto` por un valor numérico.

polimórfica: Una función que puede operar sobra más de un tipo. Si todas las operaciones realizadas dentro de una función se pueden aplicar a un tipo, la función se puede aplicar a ese tipo.