20.1. Introducción: Clases y Objetos – Básicos

20.1.1. Programación Orientada a Objetos.

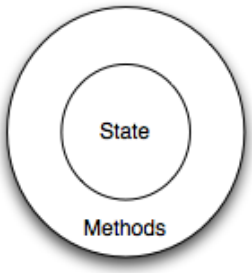
Python es un lenguaje de programación orientado a objetos. Eso significa que proporciona características que soportan la programación orientada a objetos (POO).

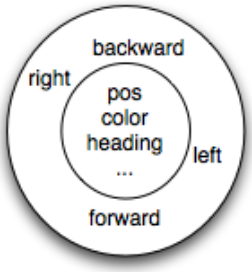
La programación orientada a objetos tiene sus raíces en los años 60, pero no fue hasta mediados de los 80 cuando se convirtió en el principal paradigma de programación utilizado en la creación de nuevo software. Se desarrolló como una forma de manejar el rápido aumento del tamaño y la complejidad de los sistemas de software y para facilitar la modificación de estos grandes y complejos sistemas a lo largo del tiempo.

Hasta ahora, algunos de los programas que hemos escrito utilizan un paradigma de programación procedimental. En la programación procedimental, la atención se centra en la escritura de funciones o procedimientos que operan, sobre los datos. En la programación orientada a objetos, la atención se centra en la creación de objetos que contienen tanto datos como funcionalidad juntos. Normalmente, cada definición de objeto corresponde a algún objeto o concepto del mundo real y las funciones que operan sobre ese objeto corresponden a las formas de interacción de los objetos del mundo real.

20.2. Objetos Revisados

En Python, cada valor es en realidad un objeto. Ya sea un diccionario, una lista, o incluso un entero, todos son objetos. Los programas manipulan esos objetos ya sea realizando cálculos con ellos o pidiéndoles que realicen métodos. Para ser más específicos, decimos que un objeto tiene un estado y una colección de métodos que puede realizar. El estado de un objeto representa aquellas cosas que el objeto conoce sobre sí mismo. El estado se almacena en variables de instancia. Por ejemplo, como hemos visto con los objetos tortuga, cada tortuga tiene un estado que consiste en la posición de la tortuga, su color, su dirección, etc. Cada tortuga también tiene la capacidad de avanzar, retroceder o girar a la derecha o a la izquierda. Las tortugas individuales son diferentes en el sentido de que, aunque todas son tortugas, difieren en los valores específicos de los atributos de estado individuales (tal vez están en una ubicación diferente o tienen un rumbo diferente).



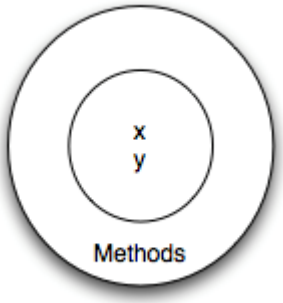


20.3. Clases definidas por el usuario

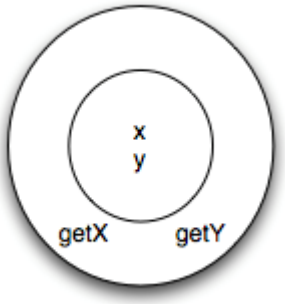
Ya hemos visto clases como str, int, float y list. Estas fueron definidas por Python y puestas a nuestra disposición para su uso. Sin embargo, en muchos casos cuando estamos resolviendo problemas necesitamos crear objetos de datos que estén relacionados con el problema que estamos tratando de resolver. Necesitamos crear nuestras propias clases.

Como ejemplo, consideremos el concepto de punto matemático. En dos dimensiones, un punto son dos números (coordenadas) que se tratan colectivamente como un solo objeto. Los puntos suelen escribirse entre paréntesis con una coma que separa las coordenadas. Por ejemplo, (0, 0) representa el origen, y (x, y) representa el punto x unidades a la derecha y unidades arriba del origen. Este (x,y) es el estado del punto.

Pensando en nuestro diagrama anterior, podríamos dibujar un objeto punto como se muestra aquí.



Algunas de las operaciones típicas que se asocian a los puntos podrían ser preguntar al punto por su coordenada x, getX, o preguntar por su coordenada y, getY. Se querrá disponer de este tipo de funciones para evitar cambios accidentales en estas variables de instancia, ya que al hacerlo se podrán ver los valores sin acceder a ellos directamente. También es posible que quieras calcular la distancia de un punto al origen, o la distancia de un punto a otro punto, o encontrar el punto medio entre dos puntos, o responder a la pregunta de si un punto cae dentro de un rectángulo o círculo determinado. En breve veremos cómo podemos organizar esto junto con los datos.



Ahora que entendemos el aspecto de un objeto punto, podemos definir una nueva clase. Querremos que nuestros puntos tengan cada uno un atributo x y un atributo y, por lo tanto, nuestra primera definición de clase se ve así.

```
1  class Point:
2      """ Point class for representing and manipulating x,y coordinates. """
3
4      def __init__(self):
5          """ Create a new point at the origin """
6          self.x = 0
7          self.y = 0
```

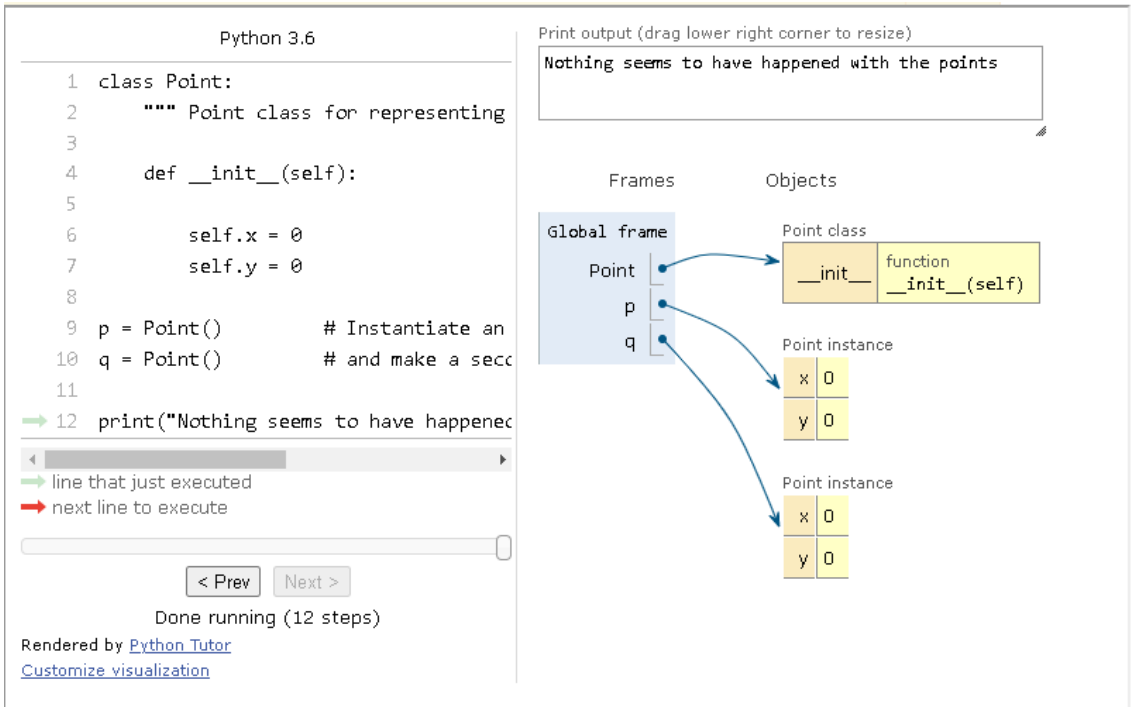
Las definiciones de clase pueden aparecer en cualquier parte de un programa, pero suelen estar cerca del principio (después de las sentencias import). Las reglas sintácticas de una definición de clase son las mismas que las de otras sentencias compuestas. Hay una cabecera que comienza con la palabra class, seguida del nombre de la clase, y que termina con dos puntos.

Si la primera línea después de la cabecera de la clase es una cadena, se convierte en el docstring de la clase, y será reconocido por varias herramientas. (Esta es también la forma en que los docstrings funcionan en las funciones).

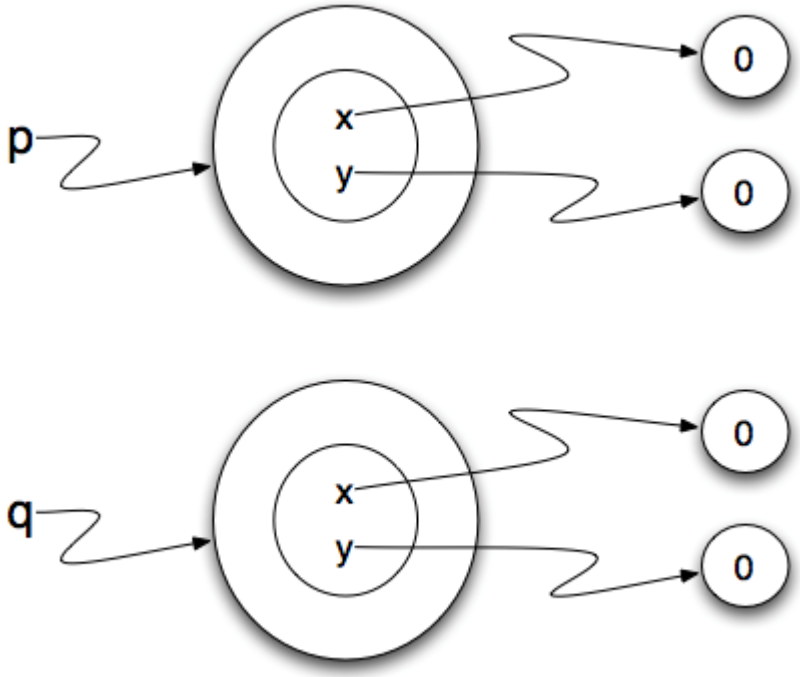
Toda clase debe tener un método con el nombre especial __init__. Este método **inicializador**, a menudo llamado **constructor**, es llamado automáticamente cada vez que se crea una nueva instancia de Point. Le da al programador la oportunidad de establecer los atributos requeridos dentro de la nueva instancia dándoles sus valores de estado iniciales. El parámetro self (se podría elegir cualquier otro nombre, ¡pero nadie lo hace nunca!) se establece automáticamente para hacer referencia al objeto recién creado que necesita ser inicializado.

Así que vamos a utilizar nuestra nueva clase Point ahora. La siguiente parte debería resultarte un poco familiar, si recuerdas algo de la sintaxis de cómo creamos instancias de la clase Tortuga, en el capítulo sobre gráficos de Tortuga.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. "'

    def __init__(self):

        self.x = 0
        self.y = 0

p = Point()          # Instantiate an object of type Point
q = Point()          # and make a second point

print("Nothing seems to have happened with the points")
```



Durante la inicialización de los objetos, creamos dos atributos llamados x e y para cada objeto, y les dimos a ambos el valor 0. Observarás que al ejecutar el programa no ocurre nada. Resulta que esto no es del todo así. De hecho, se han creado dos puntos, cada uno con una coordenada x e y con valor 0. Sin embargo, como no hemos pedido al programa que haga nada con los puntos, no vemos ningún otro resultado.



El siguiente programa añade unas cuantas sentencias de impresión. Puedes ver que la salida sugiere que cada uno es un objeto Punto. Sin embargo, observe que el operador is devuelve False, lo que significa que son objetos diferentes (tendremos más que decir sobre esto en una sección posterior).

```
3
4       def __init__(self):
5
6           self.x = 0
7           self.y = 0
8
9   p = Point()         # Instantiate an object of type Point
10  q = Point()         # and make a second point
11
12  print(p)
13  print(q)
14
15  print(p is q)
16
```

Una función como Point que crea una nueva instancia de objeto se llama constructor. Toda clase utiliza automáticamente el nombre de la clase como nombre de la función constructora. La definición de la función constructora se realiza cuando se escribe la función (método) __init__ dentro de la definición de la clase.

Puede ser útil pensar en una clase como una fábrica para hacer objetos. La clase en sí no es una instancia de un punto, pero contiene la maquinaria para hacer instancias de puntos. Cada vez que llamas al constructor, estás pidiendo a la fábrica que te haga un nuevo objeto.

A medida que el objeto sale de la línea de producción, su método de inicialización se ejecuta para que el objeto se configure adecuadamente con su configuración predeterminada de fábrica.

El proceso combinado de "hazme un nuevo objeto" y "obtén su configuración inicializada a los valores predeterminados de fábrica" se llama **instanciación**.

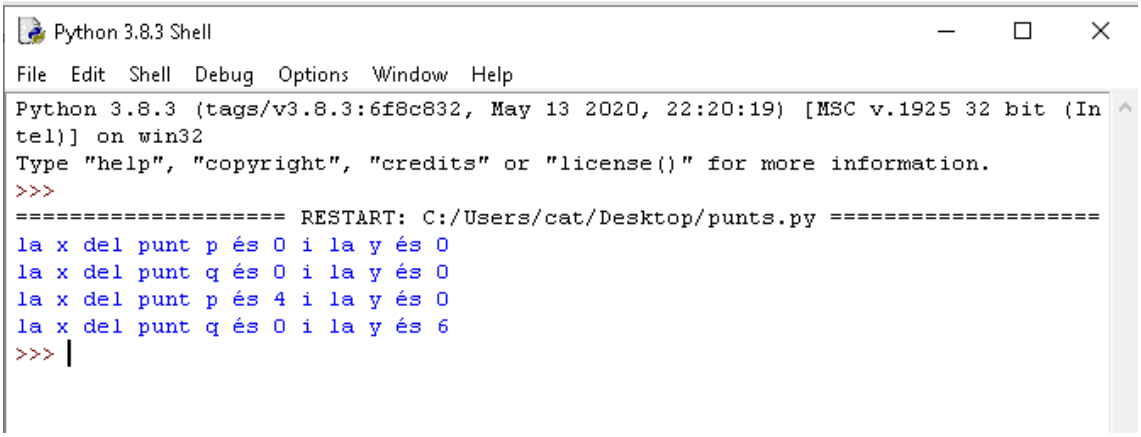
Para entender mejor lo que ocurre al instanciar una nueva instancia, examine el código anterior utilizando CodeLens.

En el paso 2 de la ejecución de CodeLens, se puede ver que Point ha sido vinculado a un objeto que representa la clase Point, pero todavía no hay ninguna instancia. La ejecución de la línea 9, p = Point(), ocurre en los pasos 3-5. Primero, en el paso 3, puedes ver que se ha creado una instancia vacía de la clase, y se pasa como primer (y único parámetro) al método __init__. El código de ese método se ejecuta, con la variable self vinculada a esa instancia. En los pasos 4 y 5, se rellenan dos variables de instancia: x e y se ponen a 0. El método __init__ no devuelve nada, pero la llamada a Point() devuelve el propio objeto punto. Así, en el paso 7, p está ligado al nuevo punto que fue creado e inicializado.

Saltando hacia adelante, en el momento en que llegamos al paso 14, p y q están cada uno ligado a diferentes instancias de Point. Aunque ambos tienen las variables de instancia x e y establecidas a 0, son objetos diferentes. Por lo tanto, p is q se evalúa como False.

Comprueba que hace este código:

```
class Point:
    """Classe Punt per crear Punts"""


    def __init__(self):
        self.x = 0
        self.y = 0


p = Point()
q = Point()
print(f"la x del punt p és {p.x} i la y és {p.y}")
print(f"la x del punt q és {q.x} i la y és {q.y}")
p.x = 4
q.y = 6
print(f"la x del punt p és {p.x} i la y és {p.y}")
print(f"la x del punt q és {q.x} i la y és {q.y}")
```



20.4. Añadir parámetros al constructor

Nuestro constructor hasta ahora sólo puede crear puntos en la posición (0,0). Para crear un punto en la posición (7, 6) se requiere que proporcionemos alguna capacidad adicional para que el usuario pase información al constructor. Dado que los constructores son simplemente funciones con un nombre especial, podemos utilizar parámetros (como hemos visto antes) para proporcionar la información específica.

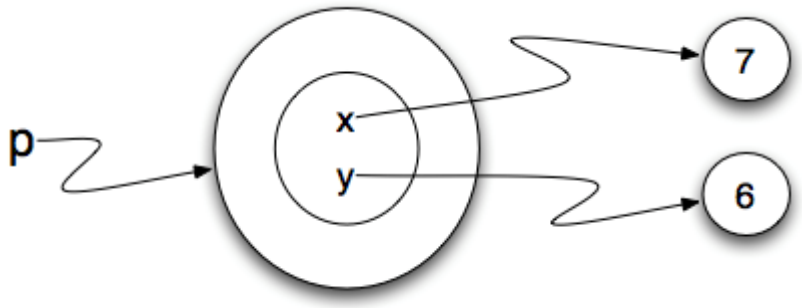
Podemos hacer que el constructor de nuestra clase tenga un uso más generalizado poniendo parámetros adicionales en el método __init__, como se muestra en este ejemplo.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. """

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

p = Point(7,6)
```

Ahora, cuando creamos nuevos puntos, suministramos las coordenadas x e y como parámetros. Cuando se crea el punto, los valores de initX e initY se asignan al estado del objeto, en las variables de instancia x e y.

Esto es algo común en el método __init__ de una clase: tomar algunos parámetros y guardarlos como variables de instancia. ¿Por qué es útil? Ten en cuenta que las variables de parámetro desaparecerán cuando el método termine de ejecutarse. Las variables de instancia, sin embargo, seguirán siendo accesibles en cualquier lugar en el que se tenga un control sobre la instancia del objeto. Esta es una forma de guardar los valores iniciales que se proporcionan cuando se invoca el constructor de la clase.

Más adelante, verás clases en las que el método __init__ hace algo más que guardar los parámetros como variables de instancia. Por ejemplo, puede analizar el contenido de esas variables y hacer algún cálculo sobre ellas, almacenando los resultados en variables de instancia. Incluso puede hacer una conexión a Internet, descargar algún contenido y almacenarlo en variables de instancia.



Comprueba tu comprensión

Crea una clase llamada NumberSet que acepte 2 enteros como entrada, y define dos variables de instancia: num1 y num2, que contienen cada uno de los enteros de entrada. Luego, crea una instancia de NumberSet donde num1 sea 6 y num2 sea 10. Guarda esta instancia en una variable t.

```
class NumberSet:

   def __init__(self, num1, num2):

      self.num1 = num1

      self.num2 = num2

t = NumberSet(6, 10)

print(t.num1 + t.num2)
```

20.5. Añadiendo otros métodos a una clase¶

La ventaja clave de utilizar una clase como Point en lugar de algo como una simple tupla (7, 6) se hace ahora evidente. Podemos añadir métodos a la clase Point que son operaciones sensibles para los puntos. Si hubiéramos optado por utilizar una tupla para representar el punto, no tendríamos esta capacidad. La creación de una clase como Point aporta una cantidad excepcional de "poder organizativo" a nuestros programas, y a nuestro pensamiento. Podemos agrupar las operaciones sensibles y los tipos de datos a los que se aplican, y cada instancia de la clase puede tener su propio estado.

Un **método** se comporta como una función, pero se invoca sobre una instancia específica. Por ejemplo, con una lista ligada a la variable L, L.append(7) llama a la función append, con la propia lista como primer parámetro y 7 como segundo parámetro. Los métodos se

acceden utilizando la notación de puntos. Por eso L.append(7) tiene dos parámetros aunque pienses que sólo tiene uno: la lista almacenada en la variable L es el primer valor del parámetro y 7 es el segundo.

Vamos a añadir dos métodos sencillos para que un punto nos dé información sobre su estado. El método getX, al ser invocado, devolverá el valor de la coordenada x.

La implementación de este método es sencilla, puesto que ya sabemos cómo escribir funciones que devuelvan valores. Una cosa que hay que notar es que aunque el método getX no necesita ninguna otra información de parámetros para hacer su trabajo, todavía hay un parámetro formal, self. Como dijimos antes, todos los métodos definidos en una clase que operan sobre objetos de esa clase tendrán a self como su primer parámetro. Una vez más, esto sirve como una referencia al objeto mismo que a su vez da acceso a los datos de estado dentro del objeto.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. "'

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y


p = Point(7,6)
print(p.getX())
print(p.getY())
```

Observe que el método getX simplemente devuelve el valor de la variable de instancia x del objeto self. En otras palabras, la implementación del método es ir al estado del objeto self y obtener el valor de x. De la misma manera, el método getY es casi igual.

Añadamos otro método, distanceFromOrigin, para ver mejor cómo funcionan los métodos. Este método tampoco necesitará ninguna información adicional para realizar su trabajo, más

allá de los datos almacenados en las variables de instancia. Realizará una tarea más compleja.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. "'

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5


p = Point(7,6)
print(p.distanceFromOrigin())
```

Observe que la llamada a distanceFromOrigin no proporciona explícitamente un argumento que coincida con el parámetro self. Esto es cierto para todas las llamadas a métodos. La definición siempre parecerá tener un parámetro adicional en comparación con la invocación.

Comprueba tu comprensión

1. Crea una clase llamada Animal que acepte dos números como entradas y los asigne respectivamente a dos variables de instancia: arms (brazos) y legs (patas).

2. Crea un método de instancia llamado limbs (extremidades) que, al ser llamado, devuelva el número total de extremidades que tiene el animal. A la variable nombre Spider, asigna una instancia de Animal que tenga 4 arms (brazos) y 4 legs (patas). Llama al método limbs (extremidades) en la instancia Spider y guarda el resultado en la variable nombre spidlimbs.

```
class Animal:

    def __init__(self,arms, legs):
        self.arms = arms
        self.legs = legs

    def limbs(self):
        return self.arms + self.legs

spider = Animal(4,4)
spidlimbs = spider.limbs()
```

20.6. Objetos como Argumentos y Parámetros

Puedes pasar un objeto como argumento a una función, de la forma habitual.

Aquí tenemos una función simple llamada distance que involucra a nuestros nuevos objetos Point. El trabajo de esta función es averiguar la distancia entre dos puntos.

```
import math

class Point:
    """ Point class for representing and manipulating x,y coordinates. "'

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

def distance(point1, point2):
    xdiff = point2.getX()-point1.getX()
    ydiff = point2.getY()-point1.getY()

    dist = math.sqrt(xdiff**2 + ydiff**2)
    return dist

p = Point(4,3)
q = Point(0,0)
print(distance(p,q))
```

distance toma dos puntos y devuelve la distancia entre ellos. Observe que distance no es un método de la clase Point. Puedes ver esto mirando el patrón de sangría. No está dentro de la definición de la clase. La otra forma de saber que distance no es un método de Point

es que no se incluye self como parámetro formal. Además, no invocamos a distancia usando la notación de punto.

Podríamos haber hecho que distance fuera un método de la clase Point. Entonces, habríamos llamado al primer parámetro self, y lo habríamos invocado usando la notación de puntos, como en el siguiente código. La forma de implementarlo es una cuestión de estilo de codificación. Ambas funcionan correctamente. La mayoría de los programadores eligen si hacer que las funciones sean independientes o métodos de una clase basándose en si la función parece semánticamente una operación que se realiza sobre instancias de la clase. En este caso, como la distancia es realmente una propiedad de un par de puntos y es simétrica (la distancia de a a b es la misma que la de b a a) tiene más sentido que sea una función independiente y no un método. Se han producido muchas discusiones acaloradas entre programadores sobre este tipo de decisiones de estilo.

```
    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def distance(self, point2):
        xdiff = point2.getX()-self.getX()
        ydiff = point2.getY()-self.getY()

        dist = math.sqrt(xdiff**2 + ydiff**2)
        return dist

p = Point(4,3)
q = Point(0,0)
print(p.distance(q))
```

20.7. Convertir un Objeto en String

Cuando trabajamos con clases y objetos, a menudo es necesario imprimir un objeto (es decir, imprimir el estado de un objeto). Considere el siguiente ejemplo.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. """

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5


p = Point(7,6)
print(p)
```

La función print mostrada arriba produce una representación de cadena del Punto p. La funcionalidad por defecto proporcionada por Python le dice que p es un objeto de tipo Punto. Sin embargo, no te dice nada sobre el estado específico del punto.

Podemos mejorar esta representación si incluimos un método especial llamado __str__. Observa que este método utiliza la misma convención de nombres que el constructor, es decir, dos guiones bajos antes y después del nombre. Es común que Python utilice esta técnica de nomenclatura para los métodos especiales.

El método __str__ es el responsable de devolver una representación de la cadena definida por el creador de la clase. En otras palabras, tú, como programador, puedes elegir el aspecto de un punto cuando se imprime. En este caso, hemos decidido que la representación de la cadena incluirá los valores de x e y, así como algún texto de identificación. Se requiere que el método __str__ cree y devuelva una cadena.

Cualquiera que sea la cadena que devuelva el método __str__ para una clase, esa es la cadena que se imprimirá cuando se ponga cualquier instancia de esa clase en una sentencia print. Por esta razón, la cadena que devuelve el método __str__ de una clase debe incluir normalmente los valores de las variables de instancia. Si un punto tiene un valor de x 3 y un valor de y 4, pero otro punto tiene un valor de x 5 y un valor de y 9, esos dos objetos Point probablemente se verán diferentes cuando los imprimas, ¿verdad?

Echa un vistazo al código siguiente.

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. """

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x = {}, y = {}".format(self.x, self.y)

p = Point(7,6)
print(p)
```

Cuando ejecutamos el programa anterior puedes ver que la función print ahora muestra la cadena que elegimos.

Ahora, te preguntarás, ¿no tenemos ya un convertidor de tipo str que puede convertir nuestro objeto en una cadena? Sí, lo tenemos.

¿Y no usa print automáticamente esto cuando imprime cosas? Sí, otra vez.

Sin embargo, como vimos antes, estos mecanismos automáticos no hacen exactamente lo que queremos. Python proporciona muchas implementaciones por defecto para métodos que, como programadores, probablemente querremos cambiar. Cuando un programador cambia el significado de un método decimos que anulamos el método. Obsérvese también que la función de conversión del tipo str utiliza cualquier método __str__ que proporcionemos.

Comprueba tu comprensión

Cree una clase llamada Cereal que acepte tres entradas: 2 cadenas y 1 entero, y las asigne a 3 variables de instancia en el constructor: name (nombre), brand (marca) y fiber (fibra). Cuando se imprime una instancia de Cereal, el usuario debería ver lo siguiente "[nombre] cereal es producido por [marca] y tiene [entero de fibra] gramos de fibra en cada porción!" Al nombre de la variable c1, asigna una instancia de Cereal cuyo nombre es "Corn Flakes", la marca es "Kellogg's", y la fibra es 2. Al nombre de la variable c2, asigna una instancia de Cereal cuyo nombre es "Honey Nut Cheerios", la marca es "General Mills", y la fibra es 3. ¡Practica la impresión de ambos!

```
class Cereal:
    def __init__(self,name, brand, fiber):
        self.name = name
        self.brand = brand
        self.fiber = fiber

    def __str__(self):
        return "{} cereal is produced by {} and has {} grams of fiber i

c1 = Cereal("Corn Flakes", "Kellogg's", 2)
c2 = Cereal("Honey Nut Cheerios","General Mills",3)

print(c1)
print(c2)
```

20.8. Instancias como Valores Return

Las funciones y los métodos pueden devolver objetos. Esto no es nada nuevo, ya que todo en Python es un objeto y hemos estado devolviendo valores durante bastante tiempo. (También puedes tener listas o tuplas de instancias de objetos, etc.) La diferencia aquí es que queremos que el método cree un objeto usando el constructor y luego lo devuelva como el valor del método.

Supongamos que tenemos un objeto punto y deseamos encontrar el punto medio entre éste y algún otro punto objetivo. Nos gustaría escribir un método, llamémoslo halfway (mitad de camino) que toma otro Punto como parámetro y devuelve el Punto que está a mitad de camino entre el punto y el punto objetivo que acepta como entrada.

```
class Point:

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

    def distanceFromOrigin(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5

    def __str__(self):
        return "x = {}, y = {}".format(self.x, self.y)

    def halfway(self, target):
        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2
        return Point(mx, my)

p = Point(3,4)
q = Point(5,12)
mid = p.halfway(q)
# note that you would have exactly the same result if you instead wrote
# mid = q.halfway(p)
# because they are both Point objects, and the middle is the same no matt

print(mid)
print(mid.getX())
print(mid.getY())
```

El punto resultante, mid, tiene un valor x de 4 y un valor y de 8. También podemos utilizar cualquier otro método sobre mid ya que es un objeto Point.

20.9. Ordenando Listas de Instancias

Ya has aprendido a ordenar listas. Ordenar listas de instancias de una clase no es fundamentalmente diferente de ordenar listas de objetos de cualquier otro tipo. Hay una manera de definir un orden por defecto para las instancias, justo en la definición de la clase, pero requiere definir un montón de métodos o un método complicado, así que no nos molestaremos con eso. En su lugar, sólo debes proporcionar una función clave como parámetro a sorted (o sort).

Anteriormente, se ha visto cómo proporcionar una función de este tipo al ordenar listas de otros tipos de objetos. Por ejemplo, dada una lista de cadenas, puede ordenarlas en orden ascendente de sus longitudes pasando un parámetro clave. Ten en cuenta que si te refieres a una función por su nombre, das el nombre de la función sin paréntesis después, porque quieres el objeto de la función en sí. La función ordenada se encargará de llamar a la función, pasando el elemento actual de la lista. Así, en el ejemplo siguiente, escribimos key=len y no key=len().

```
L = ["Cherry", "Apple", "Blueberry"]

print(sorted(L, key=len))
#alternative form using lambda, if you find that easier to understand
print(sorted(L, key= lambda x: len(x)))
```

Cuando cada uno de los elementos de una lista es una instancia de una clase, es necesario proporcionar una función que tome una instancia como entrada y devuelva un número. Las instancias se ordenarán por su número.

```
class Fruit():
    def __init__(self, name, price):
        self.name = name
        self.price = price

L = [Fruit("Cherry", 10), Fruit("Apple", 5), Fruit("Blueberry", 20)]
for f in sorted(L, key=lambda x: x.price):
    print(f.name)
```

A veces será conveniente definir un método para la clase que haga algún cálculo sobre los datos de una instancia. En este caso, nuestra clase es demasiado simple para ilustrar eso. Pero para simularlo, he definido un método sort_priority que simplemente devuelve el precio que está almacenado en la instancia. Ahora, ese método, sort_priority toma una instancia como entrada y devuelve un número. Así que es exactamente el tipo de función que necesitamos proporcionar como parámetro clave para sorted. Aquí puede ser un poco confuso: para referirse a ese método, sin invocarlo realmente, puedes referirte a Fruit.sort_priority. Esto es análogo al código anterior que hacía referencia a len en lugar de invocar len().

```
class Fruit():
    def __init__(self, name, price):
        self.name = name
        self.price = price

    def sort_priority(self):
        return self.price

L = [Fruit("Cherry", 10), Fruit("Apple", 5), Fruit("Blueberry", 20)]
print("-----sorted by price, referencing a class method-----")
for f in sorted(L, key=Fruit.sort_priority):
    print(f.name)

print("---- one more way to do the same thing-----")
for f in sorted(L, key=lambda x: x.sort_priority()):
    print(f.name)
```

20.10. Variables de Clase y Variables de Instancia

Ya has visto que cada instancia de una clase tiene su propio espacio de nombres con sus propias variables de instancia. Dos instancias de la clase Point tienen cada una su propia variable de instancia x. Establecer x en una instancia no afecta a la otra.

Una clase también puede tener variables de clase. Una variable de clase se establece como parte de la definición de la clase.

Por ejemplo, considera la siguiente versión de la clase Point. Aquí hemos añadido un método graph que genera una cadena que representa un pequeño gráfico basado en texto con el Punto trazado en el gráfico. No es un gráfico muy bonito, en parte porque el eje Y está estirado como una banda elástica, pero puedes entender la idea a partir de esto.

Observa que hay una asignación a la variable printed_rep en la línea 4. No está dentro de ningún método. Eso la convierte en una variable de clase. Se accede a ella de la misma manera que a las variables de instancia. Por ejemplo, en la línea 16, hay una referencia a self.printed_rep. Si cambias la línea 4, haces que imprima un carácter diferente en las coordenadas x, y del Punto en la gráfica

```
class Point:
    """ Point class for representing and manipulating x,y coordinates. "'

    printed_rep = "*"

    def __init__(self, initX, initY):

        self.x = initX
        self.y = initY

    def graph(self):
        rows = []
        size = max(int(self.x), int(self.y)) + 2
        for j in range(size-1) :
            if (j+1) == int(self.y):
                special_row = str((j+1) % 10) + (" "*(int(self.x) -1)) +
                rows.append(special_row)
            else:
                rows.append(str((j+1) % 10))
        rows.reverse()  # put higher values of y first
        x_axis = ""
        for i in range(size):
            x_axis += str(i % 10)
        rows.append(x_axis)

        return "\n".join(rows)


p1 = Point(2, 3)
p2 = Point(3, 12)
print(p1.graph())
print()
print(p2.graph())

```

Para poder razonar sobre las variables de clase y de instancia, es útil conocer las reglas que utiliza el intérprete de python. De esta manera, puedes simular mentalmente lo que hace el intérprete.

Cuando el intérprete ve una expresión de la forma <obj>.<varname>, éste

Comprueba si el objeto tiene una variable de instancia establecida. Si es así, utiliza ese valor.

Si no encuentra una variable de instancia, comprueba si la clase tiene una variable de clase. Si es así, utiliza ese valor.

Si no encuentra una variable de instancia o de clase, crea un error en tiempo de ejecución (en realidad, primero hace otra comprobación, que conocerás en el próximo capítulo).

Cuando el intérprete ve una sentencia de asignación de la forma <obj>.<nombredevariable> = <expr>, lo hace:

Evalúa la expresión del lado derecho para obtener algún objeto python;

Establece la variable de instancia <varname> de <obj> para que esté ligada a ese objeto python. Tenga en cuenta que una sentencia de asignación de esta forma nunca establece la variable de clase; sólo establece la variable de instancia.

Para establecer la variable de clase, se utiliza una sentencia de asignación de la forma <nombre_de_la_clase> = <expr> en el nivel superior de una definición de clase, como en la línea 4 del código anterior para establecer la variable de clase printed_rep.

Por si tienes curiosidad, las definiciones de métodos también crean variables de clase. Así, en el código anterior, graph se convierte en una variable de clase que está ligada a un objeto función/método. p1.graph() es evaluado por:

buscando p1 y encontrando que es una instancia de Point

buscando una variable de instancia llamada graph en p1, pero sin encontrarla

buscando una variable de clase llamada graph en la clase de p1, la clase Point; encuentra un objeto función/método

Debido al () después de la palabra graph, invoca el objeto función/método, con el parámetro self ligado al objeto al que apunta p1.

20.11. Pensando en clases e instancias

Ahora puedes imaginar algunas razones por las que puedes querer definir una clase. Has visto ejemplos de creación de tipos más complicados o específicos que los incorporados en Python (como listas o cadenas). Turtle, con todas las variables de instancia y los métodos que aprendiste a usar antes en el semestre, es una clase que los programadores definieron y que ahora está incluida en el lenguaje Python. En este capítulo, definimos Punto con algunas funcionalidades que pueden facilitar la escritura de programas que involucren instancias de Punto en coordenadas x,y. Y en breve, verás cómo puedes definir clases para representar objetos en un juego.

También puedes utilizar clases definidas por ti mismo para contener datos - por ejemplo, los datos que obtienes al hacer una petición a una API REST.

Antes de que decidas definir una nueva clase, hay algunas cosas que debes tener en cuenta, y preguntas que debes hacerte:

¿Cuáles son los datos con los que quieres tratar? (¿Datos sobre un montón de canciones de iTunes? ¿Datos sobre un montón de tweets de Twitter? ¿Datos sobre un grupo de búsquedas de hashtags en Twitter? ¿Dos números que representan las coordenadas de un punto en un plano de dos dimensiones?)

¿Qué representará una instancia de tu clase? En otras palabras, ¿qué tipo de cosa nueva en tu programa debería tener una funcionalidad elegante? ¿Una canción? ¿Un hashtag? ¿Un tuit? ¿Un punto? La respuesta a esta pregunta debería ayudarte a decidir cómo llamar a la clase que definas.

¿Qué información debe tener cada instancia como variables de instancia? Esto está relacionado con lo que representa una instancia. Mira si puedes convertirlo en una frase. "Cada instancia representa una < canción > y cada < canción > tiene un < artista > y un < título > como variables de instancia". O bien, "Cada instancia representa un < Tweet > y cada < Tweet > tiene un < usuario (que lo ha publicado) > y < una cadena de contenido del mensaje > como variables de instancia."

¿Qué métodos de instancia debería tener cada instancia? Qué debería poder hacer cada instancia? Para seguir usando los mismos ejemplos: Tal vez cada canción tiene un método que utiliza una API de letras para obtener una cadena larga de su letra. Quizá cada canción tenga un método que devuelva una cadena con el nombre de su artista. O en el caso de un tuit, tal vez cada tuit tenga un método que devuelva la longitud del mensaje del tuit. (¡Anímate!)

¿Qué aspecto debería tener la versión impresa de una instancia? (Esta pregunta te ayudará a determinar cómo escribir el método __str__). Tal vez, "Cada canción impresa mostrará el título de la canción y el nombre del artista" o "Cada Tweet impreso mostrará el nombre de usuario de la persona que lo publicó y el contenido del mensaje del tweet".

Después de considerar estas preguntas y tomar decisiones sobre cómo vas a empezar con la definición de una clase, puedes empezar a definir tu clase.

Recuerda que una definición de clase, al igual que una definición de función, es una descripción general de lo que debe tener cada instancia de la clase. (Las instancias de la clase son específicas: por ejemplo, el Punto con una x y una y específicas >. Podrías tener un Punto con un valor x de 3 y un valor y de 2, así que para esa instancia particular de la clase Punto, pasarías 3 y 2 al constructor, el método __init__, así: nuevo_punto = Punto(3,2), como viste en las últimas secciones.