Unidad 4

Clases y Herencia

Manual del Estudiante

CLOUD FORMACIÓN

Maria del Carmen Sierra Fernández

Contenido

Nivel I: Programación con Python (cubre Examen 98-381)

Unidad 4: Clases y Herencias

Clases

Programación orientada a objetos

Python es un lenguaje de programación orientado a objetos. Eso significa que proporciona funciones que admiten la programación orientada a objetos (OOP).

La programación orientada a objetos tiene sus raíces en la década de 1960, pero no fue hasta mediados de la década de 1980 que se convirtió en el principal paradigma de programación utilizado en la creación de nuevo software. Se desarrolló como una forma de manejar el tamaño y la complejidad cada vez mayores de los sistemas de software y para facilitar la modificación de estos sistemas grandes y complejos con el tiempo.

Hasta ahora, algunos de los programas que hemos estado escribiendo utilizan un paradigma de programación procedimental. En la programación de procedimientos, el enfoque está en la escritura de funciones o procedimientos que operan sobre datos. En la programación orientada a objetos, el foco está en la creación de objetos que contienen tanto datos como funcionalidad juntos. Por lo general, cada definición de objeto corresponde a algún objeto o concepto en el mundo real y las funciones que operan en ese objeto corresponden a las formas en que interactúan los objetos del mundo real.

Objetos

En Python, cada valor es en realidad un objeto. Ya sea un diccionario, una lista o incluso un número entero, todos son objetos. Los programas manipulan esos objetos realizando cálculos con ellos o ejecutando sus métodos. Para ser más específicos, decimos que un objeto tiene un estado y una colección de métodos que puede realizar.

Figura 1: Los objetos tienen un estado y una colección de métodos

El estado de un objeto representa aquellas cosas que el objeto sabe sobre sí mismo. El estado se almacena en variables de instancia. Ejemplo:

```
import turtle
wn = turtle.Screen()

alex = turtle.Turtle()
alex.shape("turtle")

alex.forward(120)
alex.left(90)
alex.forward(120)
alex.left(90)
alex.forward(120)

wn.exitonclick()
```

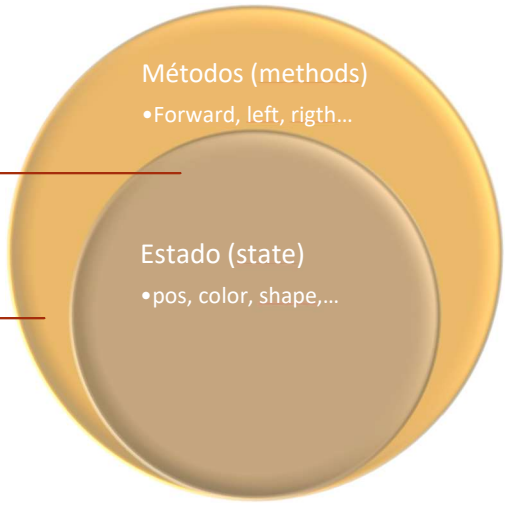


Figura 2: Ejemplo de los estados y métodos de un objeto turtle

Clases definidas por el usuario

Ya hemos visto clases como str, int y float. Estos fueron definidos por Python y están disponibles para que los usemos. 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, considere el concepto de punto matemático. En dos dimensiones, un punto son dos números (coordenadas) que se tratan colectivamente como un solo objeto. Los puntos a menudo se escriben 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 e y unidades hacia arriba desde el origen. Este (x, y) es el estado de un punto.

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

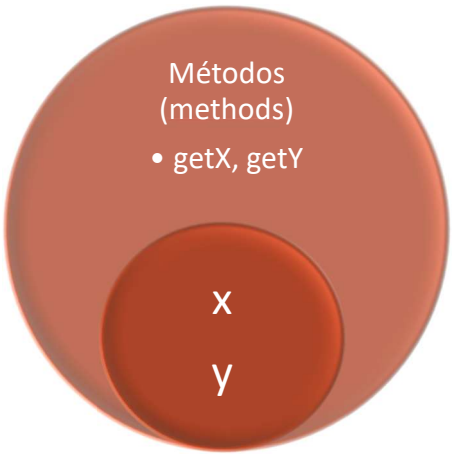


Figura 3: Ejemplo de un objeto creado por el usuario. x, y es el estado del punto y getX y getY los métodos.

Algunas de las operaciones típicas que se asocian con puntos podrían ser pedirle al punto su coordenada `x`, `getX`, o pedirle su coordenada `y`, `getY`.

También es posible que necesitemos calcular la distancia de un punto desde el origen, o la distancia de un punto a otro punto, o encontrar el punto medio entre dos puntos.

Ahora vamos a definir nuestra clase Point, donde cada punto tenga un atributo `x, y`:

```
class Point:
    """ Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self): # Este es el constructor
        """ Crea un nuevo punto en el origen 0,0 """
        self.x = 0
        self.y = 0
```

Las definiciones de clases pueden aparecer en cualquier lugar de un programa, pero generalmente están cerca del principio (después de las declaraciones de importación). Las reglas de sintaxis para una definición de clase son las mismas que para otras sentencias compuestas, se recomienda que los nombres de las clases comiencen con mayúscula. Hay un encabezado que comienza con la palabra clave, `class`, seguido del nombre de la clase y termina con dos puntos.

```
class Point:
```

Si la primera línea después del encabezado de la clase es una cadena (docstring), se convierte en la cadena de documentación de la clase y será reconocida por varias herramientas.

```
"""
Clase de punto para representar y manipular las coordenadas x, y.
"""
```

Cada clase debe tener un método con el nombre especial init. Este método de inicialización, a menudo denominado **constructor**, se llama automáticamente cada vez que se crea una nueva instancia de `Point`. Le da al programador la oportunidad de configurar los atributos requeridos dentro de la nueva instancia dándoles sus valores de estado iniciales. El parámetro `self` (puede elegir cualquier otro nombre, pero es común no cambiarlo) Se establece automáticamente para hacer referencia al objeto recién creado que debe inicializarse.

```
    def __init__(self): # Este es el constructor
        """ Crea un nuevo punto en el origen 0,0 """
        self.x = 0
        self.y = 0
```

Así que usemos nuestra nueva clase de puntos ahora.

```
p = Point()          # Instanciamos un nuevo objeto tipo Point
q = Point()          # instanciamos un segundo objeto
```

El proceso combinado de “crear un nuevo objeto" e "inicializarlo con la configuración predeterminada" se llama **instanciación**.

Durante la inicialización de los objetos, creamos dos atributos llamados `x` e `y` para cada objeto, y le asignamos a ambos el valor 0. Al ejecutar el programa, se han creado dos Puntos, cada uno con una coordenada `x` e `y` con valor 0. Sin embargo, debido a que no le hemos pedido al programa que haga nada con los puntos, no vemos ningún otro resultado.

Podemos observar que p y q son objetos tipos Point, que están en instancias diferentes.

```
p es de tipo <__main__.Point at 0x2014c4fc448>
q es de tipo <__main__.Point at 0x2014c4fc448>
print(p is q) False
```

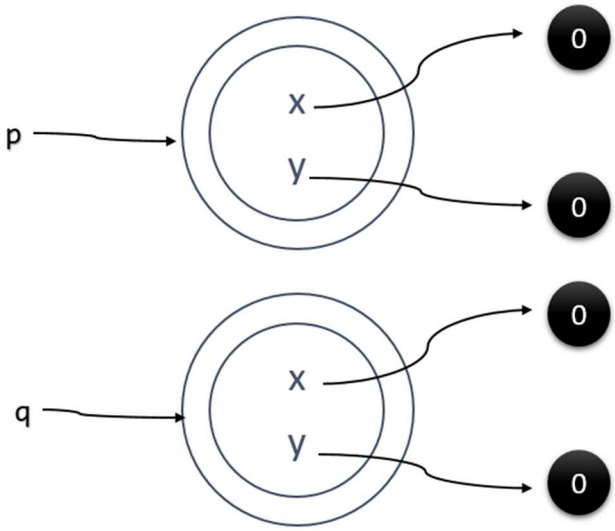


Figura 4: Creación e instanciación de los objetos p y q

La ejecución de `p = Point()` ha creado una instancia en blanco de la clase Point y se pasa como el primer y único parámetro, al método `init`. El código del método `init` se ejecuta, con la variable vinculada a esa instancia. Luego se inicializan las dos variables de instancia: `x, y` en `0`. Finalmente, `p` y `q` están vinculados a diferentes instancias de Point (Ver figura 4). Aunque ambos tienen variables de instancia `x, y` establecidas en `0`, son objetos diferentes. Por tanto, `p is q` se evalúa como `False`.

Crear una clase vacía

Una clase vacía, es una clase que inicialmente no contendrá nada, excepto la declaración de pass. Por ejemplo:

```
class Alumno:
    pass
```

Para crear una instancia de esta clase:

```
jose = Alumno()
print(jose)

<__main__.Alumno object at 0x000002C90E992088>
```

NOTA: Según la documentación de Python, la sentencia pass no hace nada. Puede ser utilizada cuando se requiere una sentencia sintácticamente, pero programa no requiere acción alguna.

Ahora, podemos crear los atributos a nuestro objeto `jose`:

```
jose.nombres = 'José Manuel'
jose.apellidos = 'López Fernández'
jose.curso = "Programación Python"
```

y podemos invocar sus atributos con la notación de puntos:

```
jose.curso
'Programación Python'
```

Agregar parámetros al constructor

Nuestro constructor hasta ahora solo puede crear puntos en la ubicación (0,0). Para crear un punto en la posición (7, 6) es necesario que proporcionemos alguna capacidad adicional para que el usuario pase información al constructor. Dado que los constructores son simplemente funciones con nombres especiales, podemos usar parámetros para proporcionar la información específica.

Podemos hacer que nuestro constructor de clases se pueda usar de manera más general poniendo parámetros adicionales en el método `init`, como se muestra en este ejemplo.

```
class Point:
""" Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self, initX, initY ):  # Este es el constructor
        """ Crea un nuevo punto en el origen initX, initY """
        self.x = initX
        self.y = initY

p = Point(7,6)
```

Ahora, cuando creamos nuevos puntos, proporcionamos las coordenadas `x,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`.

El método **init** para una clase, se utiliza para tomar algunos parámetros y almacenarlos como variables de instancia. Recordemos, que los parámetros desaparecerán cuando el método termine de ejecutarse. Sin embargo, las variables de instancia seguirán estando accesibles en cualquier lugar. Esta es una forma de guardar los valores iniciales que se proporcionan cuando se invoca el constructor de la clase. El método **init**, podría utilizarse para analizar el contenido de las variables y hacer algunos cálculos sobre ellas, almacenando los resultados en variables de instancia. Incluso podría establecer una conexión a Internet, descargar contenido y almacenarlo en variables de instancia.

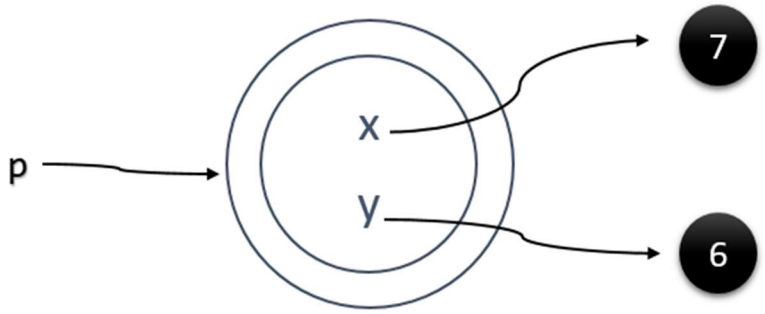


Figura 5: Ejemplo de instanciación de p

Agregar otros métodos a una clase

La ventaja clave de usar una clase como Point en lugar de algo como una simple tupla (7, 6), es que podemos agregar métodos a la clase Point, es decir, operaciones con puntos. La creación de una clase aporta organización a nuestros programas. Podemos agrupar las operaciones y los tipos de datos a los que se aplican, y cada instancia de una clase puede tener su propio estado.

Un método se comporta como una función, pero se invoca en una instancia específica. Por ejemplo: Un string vinculado a la variable `mi_frase`, llama a la función `split()`, la invoca con su propio nombre, como primer parámetro `mi_frase.split`(“,”) y cómo segundo parámetro en este caso, le pasa “,”.

Ahora vamos a incorporar dos métodos, que nos permitan obtener información sobre el estado del punto. El método `getX`, cuando se invoca, retornará el valor de la coordenada x y el método `getY`, retornará el valor de la coordina y. Estos métodos no necesitan ninguna otra información de parámetros para hacer su trabajo, sólo utilizaremos un parámetro formal, `self`. Como sabemos, todos los métodos definidos en una clase, que operan dentro de la misma clase tendrán `self` como su primer parámetro. Es decir, esto sirve como referencia al objeto en sí, que a su vez da acceso a los datos de estado dentro del objeto.

```
class Point:
      """ Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self, initX, initY ):
        """ Crea un nuevo punto en el origen 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())
7
6
```

Luego de invocar a los métodos getX y getY del objeto p, observamos que ambos métodos retornan el valor de la variable de instancia x o y respectivamente del objeto self. En otras palabras, la implementación del método `get`, es ir al estado del propio objeto y obtener el valor de x o y.

Agreguemos otro método, `distanciaDOrigen`:

```
class Point:
    """ Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self, initX, initY ):
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x

    def getY(self):
        return self.y

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

p = Point(7,6)
print(p.distanciaDOrigen())
9.219544457292887
```

Métodos que utilizan objetos como argumentos y parámetros

Ahora crearemos un método que calcule la distancia entre dos (2) puntos, este método necesitará dos parámetros:

```
class Point:
    """ Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self, initX, initY ):
        """ Crea un nuevo punto en el origen initX,initY """
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x
```

```
    def getY(self):
        return self.y

    def distanciaDOrigen(self):
        return math.sqrt((self.x ** 2) + (self.y ** 2))

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

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

p = Point(7,6)
q = Point(4,3)
print(p.distancia(q))  # Calculamos la distancia entre p y q
```

Observamos que el método `distancia` tiene dos parámetros: `self` y `point2`. Cuando invocamos el método para el punto `p`, lo hacemos utilizando la notación de puntos: `p.distancia(q)`, donde `p` es el primer parámetro `self` y `q` es el segundo parámetro `point2`. El método realiza los cálculos y retorna la distancia (`dist`) entre los dos puntos `p` y `q`.

Método especiales underescore (__)

__str__

El método `str` es responsable de devolver una representación de cadena según lo definido por el creador de la clase. Es decir, nosotros como programadores, podemos elegir el aspecto que debe tener un objeto cuando se imprime.

```
def __str__()
```

Este método utiliza la misma convención de nomenclatura que el constructor, es decir, dos guiones bajos antes y después del nombre. Es común en Python utilizar esta técnica de nomenclatura para métodos especiales.

Ahora vamos a crear un método __str__ para representar un punto, la cadena incluirá los valores de x e y, así como algún texto de identificación. El método str creará y retornará esta cadena.

```
class Point:
    """ Clase de punto para representar y manipular las coordenadas x, y. """

    def __init__(self, initX, initY
        self.x = initX
        self.y = initY

    def getX(self):
        return self.x
```

```
    def getY(self):
        return self.y

    def distanciaDOrigen(self):
        return math.sqrt((self.x ** 2) + (self.y ** 2))

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

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

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

p = Point(4,3)
q = Point(10,100)

print(p)
print(q)
x = 4, y = 3
x = 10, y = 100
```

Cualquiera que sea la cadena que devuelva el método `str` para una clase, esa será la cadena que se imprimirá cuando coloque cualquier instancia de esa clase en una declaración de impresión. Por esa razón, la cadena que devuelve el método `str` de una clase generalmente debe incluir valores de variables de instancia.

__repr__

Llamado por la función incorporada repr () para calcular la representación de cadena "oficial" de un objeto. El valor de retorno debe ser un objeto de cadena. Si una clase define __repr __ () pero no __str __ (), entonces __repr __ () también se usa cuando se requiere una representación de cadena “informal” de instancias de esa clase.

```
def __repr__(self):
            return self.__str__()

p = Point(4,3)
p
Las coordenadas del punto son: x = 4, y = 3
```

Otros métodos especiales

`__unicode__(self)`, define el comportamiento para la invocación de unicode(objeto).

`__format__(self, formatstr)`, define el comportamiento para cuando una instancia de la clase se utiliza como argumento del método cadena.format(objeto).

`__hash__(self)`, define el comportamiento para la invocación de hash(objeto). Tiene que devolver un número entero, y su resultado se utiliza para la comparación rápida de claves en los diccionarios.

Llamada por la función integrada hash()para las operaciones en los miembros de colecciones hash, incluidos set, frozenset y dict.

`__nonzero__(self)`, define el comportamiento para la invocación de bool(objeto). Debe retornar True o False, dependiendo de si desea considerar la instancia como True o False.

`__new__(clase,...)`, este método es el primero en ser invocado al crear un objeto, incluso antes del método __init__(). Recibe como argumento la clase, seguida de los argumentos que se pasarán al método __init__().

`__del__(self)`, este método es invocado antes de destruir un objeto. Se utiliza para definir lo que tiene que pasar antes de destruir un objeto. Se utilizar cuando trabajamos con archivos, en este método podemos cerrar el fichero o archivo.

Puedes consultar otros métodos en:
https://docs.python.org/3/reference/datamodel.html#special-method-names

Métodos especiales para realizar operaciones entre objetos

Los siguientes métodos se pueden definir para emular objetos numéricos. Es decir que podemos realizar operaciones entre los objetos de una clase.

Método	Operación	Operador
__add__(self, otro)	Suma	+
__sub__(self, otro)	Resta	-
__mul__(self, otro)	Multiplicación	*
__floordiv__(self, otro)	División entera	//
__div__(self, otro)	División real	/
__mod__(self, otro)	Módulo o Residuo	%
__pow__(self, otro[, modulo])	Potencia	**
__lshift__(self, otro)	Desplazamiento a izquierda	<<
__rshift__(self, otro)	Desplazamiento a derecha	>>
__and__(self, otro)	'AND' de bits	&
__xor__(self, otro)	'XOR' de bits	^
__or__(self, otro)	'OR' de bits	|
__iadd__(self, otro)	Suma abreviada	+=
__isub__(self, otro)	Resta abreviada	-=
__imul__(self, otro)	Multiplicación abreviada	*=
__ifloordiv__(self, otro)	División entera abreviada	//=

__idiv__(self, otro)	División real abreviada	/=
__imod__(self, otro)	Módulo o Residuo abreviado	%=
__ipow__(self, otro[, modulo])	Potencia abreviada	**=
__ilshift__(self, otro)	Desplazamiento a izquierda de bits abreviada	<<=
__irshift__(self, otro)	Desplazamiento a derecha de bits abreviada	>>=
__iand__(self, otro)	'AND' de bits abreviado	&=
__ixor__(self, otro)	'XOR' de bits abreviado	^=
__ior__(self, otro)	'OR' de bits abreviado	|=

Ahora vamos a crear dos nuevos métodos especiales, para sumar o restar dos puntos en nuestra clase Point:

```
class Point:
      …
    def __add__(self, otherPoint):
        return Point(self.x + otherPoint.x, self.y + otherPoint.y)

    def __sub__(self, otherPoint):
        return Point(self.x - otherPoint.x, self.y - otherPoint.y)

p = Point(4,3)
q = Point(10,100)

print(p + q)
print(p - q)
```

```
x = 14, y = 103
x = -6, y = -97
```

También podemos escribir métodos para operadores unarios.

Método	Operación	Operador
__neg__(self)	Negar	-
__pos__(self)	Positivar	+
__abs__(self)	Valor absoluto	abs()
__invert__(self)	Invertir valor	~
__complex__(self)	Convertir a complejo	complex()
__int__(self)	Convertir a entero corto	int()
__long__(self)	Convertir a entero largo	long()
__float__(self)	Convertir a punto flotante	float()
__oct__(self)	Convertir a octal	oct()
__hex__(self)	Convertir a hexadecimal	hex()

Ejemplo:

```
class Point:
```

```
      ....
      def __invert__(self):
         return Point(self.y, self.x)
p = Point(4,3)
p = ~p
print(p)
Las coordenadas del punto son: x = 3, y = 4
```

Los métodos para operaciones de comparación son:

Método	Operación	Operador
__lt__(self, other)	Menor que	<
__le__(self, other)	Menor o igual que	<=
__eq__(self, other)	Igual que	==
__ne__(self, other)	Diferente que	!=
__ge__(self, other)	Mayor o igual que	>=
__gt__(self, other)	Mayor que	>

Ejemplo:

```
class Point:
      ....
      def __eq__(self, otherPoint):
          return self.x == otherPoint.x and self.y == otherPoint.y

a = Point(14,13)
b = Point(14,13)
a == b
True
```

Retornar una Instancia

Las funciones y los métodos pueden devolver objetos. En realidad, esto no es nada nuevo, ya que todo en Python es un objeto y hemos estado retornando valores. La diferencia es que en una clase el método puede crear un objeto usando el constructor y luego lo retorne como el valor del método.

Ahora vamos a crear un nuevo método en la clase Point, para crear un punto medio entre dos puntos:

```
      class Point:
            …

          def puntomedio(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)
z = p.puntomedio(q)

print(z)
Las coordenadas del punto son: x = 4.0, y = 8.0

print(type(z))
<class '__main__.Point'>
```

En este caso z es una nueva instancia de Point, es decir un nuevo punto, cuyas coordenadas son 4 y 8. Comprobemos si podemos acceder a la información del punto z:

```
print(z.getX())
print(z.getY())
4.0
8.0
```

Agregar comportamiento de iteración a una clase creada por el usuario

Como vimos en el tema 1, un **iterable** es un objeto que puede devolver un **iterador. Iterable** es un objeto donde sus elementos se obtienen uno por uno. Las colecciones Python incorporadas que son iterables son:

```
[1, 2, 3]     # Listas
(1, 2, 3)     # Tuplas
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict o diccionarios
```

Los **iteradores** son objetos que hacen referencia a un elemento, y que tienen un método **next** que permite hacer referencia al siguiente elemento.

La función **iter()**, nos retorna un iterador.

```
cadena = "Saber gobernar es rectificar"
iterador = iter(cadena)
print(iterador)
print(type(iterador))

<str_iterator object at 0x000002394EF11FC8>
<class 'str_iterator'>
```

La función iter(), nos retorna un iterado str, porque la variable cadena es una string. Esta función nos retorna el iterador de acuerdo con la clase de la variable.

- str_iterator para cadenas
- list_iterator para sets.
- tuple_iterator para tuplas.

- set_iterator para sets.
- dict_keyiterator para diccionarios.

Con la función next(), podemos acceder a sus elementos: `next(iterador)`

```
cadena = "Saber gobernar es rectificar"
iterador = iter(cadena)
print(next(iterador))
S
```

Una forma de iterar un objeto iterable es utilizando *in*

```
text = "Saber gobernar es rectificar"
for letra in text:
    print(letra)

for i in text[::2]: # toma un elemento sí y otro no
    print(i)
```

Una forma de iterar un objeto iterable, es utilizando *in*

```
text = "Saber gobernar es rectificar"
for letra in text:
    print(letra)

for i in text[::2]: # toma un elemento sí y otro no
    print(i)
```

Sabemos que 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__()` levanta una excepción `StopIteration` que le avisa al bucle del `for` que hay que terminar. Podemos llamar al método `__next__()` usando la función integrada `next()`.

Ahora que conocemos la mecánica del protocolo de iteración, podemos agregar comportamiento de iterador a nuestras clases. Definir un método `__iter__()` que retorne un objeto con un método `__next__()`. Si la clase define `__next__()`, entonces logra que `__iter__()` retorne `self`:

```
class OrdenInverso:
    """Muestra un string en Orden Inverso."""
    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]

    inverso = OrdenInverso('spam')
    for char in inverso: # inverso es un objeto iterable
        print(char)
    m
    a
    p
    s
```

Variables de clase y variables de instancia

Cada instancia de una clase tiene su propio espacio de nombres con sus propias variables de instancia. Es decir, las variables de instancia son datos únicos de cada instancia.

Una clase también puede tener variables de clase. Una variable de clase se establece como parte de la definición de clase. Es decir, las variables de clase son atributos y métodos compartidos por todas las instancias de la clase

En el siguiente ejemplo, podemos observar que las variables attack, defense, health y p_type son variables de clase. Mientras que name y level son variables de instancia.

```
class Pokemon():

    attack = 12
    defense = 10
    health = 15
    p_type = "Normal"

    def __init__(self, name, level = 5):
        self.name = name
        self.level = level

    def __str__(self):
        return "Pokemon: {}, Tipo: {}, Nivel: {}, Ataque: {},
                Defensa: {}, Salud: {}".format(self.name,
                                               self.p_type,
                                               self.level,
                                               self.attack,
                                               self.defense,
                                               self.health)

    def cambiar_ataque(self):
```

```
        self.attack = self.attack + 10

    def gano_combate(self):
        self.level = self.level + 1
        self.defense = self.defense + 1

pikachu = Pokemon("Pikachu")
print(pikachu)
```

```
Pokemon: Pikachu, Tipo: Normal, Nivel: 5, Ataque: 12, Defensa: 10,
Salud: 15
```

```
bulby = Pokemon("Bulby")
print(bulby)
```

```
Pokemon: Bulby, Tipo: Normal, Nivel: 5, Ataque: 12, Defensa: 10,
Salud: 15
```

En nuestro ejemplo, cuando instanciamos los objetos `pikachu` y `bulby`, se crea una instancia para cada objeto, sus variables de instancia ahora son `name` y `level` de instancia definidas en el método `__init__`. Observamos que al invocar el método `__str__` nos muestra los valores definidos en las variables de clase de `attack`, `defense`, `health` y `p_type`.

Para entender las variables de clase y variables de instancia, es útil conocer las reglas que usa el intérprete de Python.

1. El intérprete primero comprueba si el objeto tiene un conjunto de variables de instancia. Si es así, usa ese valor.
2. Si no encuentra una variable de instancia, verifica si la clase tiene una variable de clase. Si es así, usa ese valor.
3. Si no encuentra una instancia o una variable de clase, crea un error de tiempo de ejecución.

Veamos ahora que pasa, si invocamos el método `cambiar_ataque()` para el objeto `pikachu`.

```
pikachu.cambiar_ataque()
print(pikachu)
```

```
Pokemon: Pikachu, Tipo: Normal, Nivel: 5, Ataque: 22, Defensa: 10,
Salud: 15
```

Observamos que el nivel de ataque ha cambiado de 12 a 22, si consultamos el atributo `attack` del objeto `pikachu`, vemos que retorna 22.

```
pikachu.attack
22
```

Cuando el intérprete ve una declaración de asignación, por ejemplo:

```
self.attack = self.attack + 10
```

evalúa la expresión del lado derecho; establece la variable de instancia para que se vincule a ese objeto de Python. Tenga en cuenta que una declaración de asignación de este formulario nunca establece la variable de clase; solo establece la variable de instancia.

Como identificar clases e instancias

Antes de decidir definir una nueva clase, hay algunas cosas que debemos tener en cuenta y algunas preguntas que debe hacernos:

1. ¿Cuáles son los datos con los que quiere tratar? (¿Datos sobre canciones de iTunes? ¿Datos sobre de tweets de Twitter? ¿Búsquedas de hashtags en Twitter? ¿Dos números que representan las coordenadas de un punto en un plano bidimensional?)
2. ¿Qué representará una instancia de la clase? En otras palabras, ¿qué tipo de cosa nueva en nuestro programa debería tener una funcionalidad elegante? ¿Una canción? ¿Un hashtag? ¿Un tweet? ¿Un punto? La respuesta a esta pregunta debería ayudarnos a decidir cómo llamar a la clase que defina.
3. ¿Qué información debería tener cada instancia como variables de instancia? Esto está relacionado con lo que representa una instancia. Una forma práctica es convertirlo en una oración. "Cada instancia representa una <canción> y cada <canción> tiene un <artista> y un <título> como variables de instancia". O, "Cada instancia representa un <Tweet> y cada <Tweet> tiene un <usuario (que lo publicó)> y <una cadena de contenido de mensaje> como variables de instancia".
4. ¿Qué métodos de instancia debe tener cada instancia? ¿Qué debería poder hacer cada instancia? Por ejemplo: Quizás cada canción tenga un método que devuelva una cadena con el nombre de su artista. O para un tweet, tal vez cada tweet tenga un método que devuelva la longitud del mensaje del tweet.
5. ¿Cómo debería verse la versión impresa de una instancia? De esta forma podemos determinar cómo escribir el método __str__. Por ejemplo: cada canción mostrará el título de la canción y el nombre del artista". o "Cada Tweet mostrará el nombre de usuario de la persona que lo publicó y el contenido del mensaje del tweet".

Después de considerar estas preguntas, podemos tomar decisiones sobre cómo comenzar con la definición de una clase.

Herencia de clases

La herencia es una característica poderosa en la programación orientada a objetos. Se refiere a definir una nueva clase con poca o ninguna modificación de una clase existente. La nueva clase se denomina clase derivada o secundaria y la clase de la cual se hereda se denomina clase base o principal.

Sintaxis de herencia de Python

```
class BaseClass:
  Cuerpo de la clase Base

class DerivedClass(BaseClass):
  Cuerpo de la clase Derivada
```

La clase derivada hereda características de la clase base, agregándole nuevas características. Esto resulta en la reutilización del código.

Veamos un ejemplo: tenemos la clase `Persona`, con los atributos `nombre` y `año de nacimiento`. La clase `Persona` tiene un método `obtener_edad()`, que al invocarlo nos retorna la edad de la persona.

```
from datetime import date

AÑO_ACTUAL = date.today().year

class Persona:

    def __init__(self, nombre, año_nac):
        self.nombre = nombre
        self.año_nac = año_nac

    def obtener_edad(self):
        return AÑO_ACTUAL- self.año_nac

    def __str__(self):
        return '{} nació en {}'.format(self.nombre,
self.año_nac)


alicia = Persona("Alicia Mendoza", 1992)
print(alicia.obtener_edad())
print(alicia)
28
Alicia Mendoza nació en 1992
```

Ahora queremos crear una clase `Estudiante`, cada estudiante debe tener un `nombre`, `año de nacimiento` y el `curso` en el cual se matriculó. También crearemos un método

`registrar asistencia`, donde marcaremos las veces que ha asistido a sus clases. Para crear la clase estudiante podemos hacerlo de dos formas: crear una nueva clase estudiante o utilizar las ventajas de la herencia y crear la nueva clase `Estudiante` tomando como clase base `Persona`. Veamos la segunda opción:

```
from datetime import date

AÑO_ACTUAL = date.today().year

class Persona:

    def __init__(self, nombre, año_nac):
        self.nombre = nombre
        self.año_nac = año_nac

    def obtener_edad(self):
        return AÑO_ACTUAL- self.año_nac

    def __str__(self):
        return '{} nació en {}'.format(self.nombre, self.año_nac)

# Crearemos la clase Estudiante utilizando la herencia

class Estudiante(Persona):

    def __init__(self, nombre, año_nac, curso):
        Persona.__init__(self, nombre, año_nac)
        self.curso = curso
        self.n_asistencias = 0

    def registrar_asistencia(self):
        self.n_asistencias += 1

    def __str__(self):
        return '{} es estudiante del curso {}'.format(self.nombre,
self.curso)

alicia = Estudiante("Alicia Mendoza", 1992, "Programación Python")
print(alicia)
alicia.registrar_asistencia()
print(alicia.n_asistencias)
print(alicia.obtener_edad())
Alicia Mendoza es estudiante del curso Programación Python
1
28
```

Observe que en el método de iniciación `__init__` de la clase `Estudiante`, estamos heredando los atributos nombre y año de nacimiento definidos en la clase base Persona, utilizando la sentencias:

```
Persona.__init__(self, nombre, año_nac) o

super().__init__(nombre, año_nac).
```

La clase derivada `Estudiante`, ha heredado el método `obtener_edad`. Vemos además que la clase Estudiante tiene sus propios métodos `__str__` y `registrar_asistencias()`.

La herencia básicamente funciona definiendo una nueva clase y usando una sintaxis especial para mostrar lo que la nueva clase derivada o subclase hereda de una base o superclase.

Métodos anuladores o overrriding.

Si un método se define para una clase y también se define para su clase base o principal, se llama al método de la clase derivada o subclase y no al método de la clase base o principal. A estos métodos se les conoce como **métodos anuladores o overrriding**.

Funciones integradas para herencia.

Python ofrece dos funciones integradas que funcionan con herencia:

`isinstance(objeto, claseinfo)`: permite verificar el tipo de una instancia. Retorna True si `objeto` es una instancia del argumento `claseinfo`. Ejemplos:

```
isinstance(alicia, Persona)
True

isinstance(alicia, Estudiante)
False

isinstance(maria, Estudiante)
True
```

Podemos observar, que `alicia` es una instancia de la clase `Persona`, pero no es una instancia de la clase Estudiante. Mientras `maria` es una estancia de la clase `Estudiante`.

`issubclass(clase, claseinfo)`: permite verificar la herencia de clases. Retorna True si `clase`, es una clase derivada o subclase de `claseinfo`. Ejemplo:

```
issubclass(Estudiante, Persona)
True

issubclass(Persona, Estudiante)
False
```

Podemos observar, que `Estudiante` es una clase derivada o subclase de `Persona`.

Herencia múltiple

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

Sintaxis:

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

Para la mayoría de los propósitos, podemos pensar en la búsqueda de los atributos heredados de clases padres primero busca de izquierda a derecha, sin repetir la misma clase. Por lo tanto, si un atributo no se encuentra en `DerivedClassName`, 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.

Veamos un ejemplo:

```
class Estudiantes(object):  #Creamos la clase padre

    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

class Universidad(object):

    def presentar_facultad (self):
        print("""Estudio en la Facultad de Psicologia""")

class Psicologia (Estudiantes, Universidad): # Estudiante Clase
principal y Universidad en secundaria

    def presentarse(self):
        self.clase = "Psicología"
        print('Soy {}, tengo {} años y soy estudiante de
                  {}'.format(self.nombre, self.edad, self.clase))

gus = Psicologia('Gustavo Ortiz', 18)
gus.presentarse()
gus.presentar_facultad()
gus.nombre
Soy Gustavo Ortiz, tengo 18 años y soy estudiante de Psicología
Estudio en la Facultad de Psicología
'Gustavo Ortiz'
```

Observamos que la clase `Psicologia`, hereda de la clase principal `Estudiante` y `Universidad`. Y aunque el objeto `gus` pertenece a la clase `Psicologia`, podemos utilizar el método `presentar_universidad()`, que corresponde a la clase Universidad y también podemos acceder a los atributos nombre y edad.

Enumeraciones

Una enumeración es un conjunto de nombres simbólicos (miembros) vinculados a valores únicos y constantes. Dentro de una enumeración, los miembros se pueden comparar por identidad, y la enumeración en sí se puede iterar.

El módulo enum

Este módulo define cuatro clases de enumeración que se pueden usar para definir conjuntos únicos de nombres y valores: `Enum, IntEnum, Flag, and IntFlag`. También define un decorador, unique(), y un ayudante, auto.

Enum: Clase base para crear constantes enumeradas.

IntEnum: Clase base para crear constantes enumeradas que también son subclases de int.

unique(): El decorador de clase Enum que garantiza que solo un nombre esté vinculado a cualquier valor.

auto: Las instancias se reemplazan con un valor apropiado para los miembros de Enum. El valor inicial comienza en 1.

Las enumeraciones se crean como cualquier otra clase, heredando desde `Enum` o `IntEnum`, lo que permite una rápida y fácil interpretación.

```
from enum import Enum

class Color(Enum):
    ROJO = 1
    VERDE = 2
    AZUL = 3
```

- La clase Color es una enumeración
- Los atributos Color.rojo, Color.verde, Color.azul, son miembros de la enumeración.
- Los miembros de la enumeración tienen nombres y valores. Por ejemplo: el nombre de Color.ROJO es ROJO, el valor de Color.ROJO es 1.

Los miembros de la enumeración tienen representaciones de cadenas legibles:

```
print(Color.ROJO)
Color.ROJO

print(repr(Color.ROJO))
```

```
<Color.ROJO: 1>

type(Color.ROJO)
<enum 'Color'>

print(Color.ROJO.name)
ROJO
```

Las enumeraciones soportan iteración, en orden de definición:

```
for n in Color:
    print(n)

Color.ROJO
Color.VERDE
Color.AZUL
```

El atributo especial __members__ es una asignación ordenada de solo lectura de nombres a miembros. Incluye todos los nombres definidos en la enumeración, incluidos los alias:

```
for name, member in Color.__members__.items():
    print(name, member)

ROJO Color.ROJO
VERDE Color.VERDE
AZUL Color.AZUL
```

Acceso programático a los miembros de la enumeración y sus atributos

A veces es útil acceder a los miembros en enumeraciones mediante programación (es decir, situaciones en las que Color.ROJO no funcionará porque no se conoce el color exacto al momento de escribir el programa). Enum permite dicho acceso:

```
Color(1)
<Color.ROJO: 1>
```

Si desea acceder a los miembros de enumeración por *nombre*, use el acceso a elementos:

```
Color['ROJO']
<Color.ROJO: 1>
```

Si tiene un miembro enum y necesita su name o value:

```
member = Color.ROJO
member.name
'ROJO'

member.value
1
```

Garantizando valores de enumeración únicos

Por defecto, las enumeraciones permiten múltiples nombres como alias para el mismo valor. Cuando no se desea este comportamiento, se puede usar el siguiente decorador para garantizar que cada valor se use solo una vez en la enumeración:

```
@enum.unique
```

Un decorador de clase específicamente para enumeraciones. Busca una enumeración `__members__` reuniendo cualquier alias que encuentre; si no se encuentra alguno se genera un ValueError con los detalles:

```
from enum import Enum, unique

@unique

class Mistake(Enum):
    ONE = 1
    TWO = 2
    THREE = 3
    FOUR = 3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-128-e584f085635f> in <module>
      1 from enum import Enum, unique
      2 @unique
----> 3 class Mistake(Enum):
      4     ONE = 1
      5     TWO = 2

~\Anaconda3\envs\AAUT1\lib\enum.py in unique(enumeration)
    867                 ["%s -> %s" % (alias, name) for (alias, name) in duplicates])
    868         raise ValueError('duplicate values found in %r: %s' %
--> 869                 (enumeration, alias_details))
    870     return enumeration
    871

ValueError: duplicate values found in <enum 'Mistake'>: FOUR -> THREE
```

Usando valores automáticos

Si el valor exacto no es importante, puede usar auto:

```
from enum import Enum, auto

class Colores(Enum):
    RED = auto()
    BLUE = auto()
    GREEN = auto()

list(Color)
[<Color.RED: 1>, <Color.BLUE: 2>, <Color.GREEN: 3>]
```

Comparaciones

Los miembros de la enumeración se comparan por identidad:

```
Colores.RED is Colores.RED
True
Color.RED is Color.BLUE
False
```

Decoradores

Antes de hablar de los decoradores es importante destacar que una función puede ser asignada a una variable, puede ser utilizada como argumento para otra función, o inclusive puede retornar una función. Veamos, algunos ejemplos:

Ejemplo 1:

```
def sumar(a, b):
    return a + b

def super_funcion(f, a, b):
    print("Valores recibidos", a)
    print("Valores recibidos", b)
    return f(a,b)

funcion = sumar  # Asignamos la función a una variable
super_funcion(funcion, 7, 6)  # pasamos la función como
argumento
Valores recibidos 7
Valores recibidos 6
El resultado es:  13
```

Ejemplo 2:

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def operacion(func, x):
    result = func(x)
    return result

operacion(inc,3)
4
operacion(dec,3)
2
```

Un decorador es el nombre de un patrón de diseño. Los decoradores alteran de manera dinámica la funcionalidad de una función, método o clase sin tener que hacer subclases o cambiar el código fuente de la clase decorada. Los **decoradores** son en sí mismos funciones, que toman como argumento una función y retornan otra función.

```
def f(...):
    ...
    f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

Veamos un ejemplo, tenemos la función.

```
def sumar(a, b):
    return a + b
```

Supongamos, que queremos imprimir un mensaje en pantalla cada vez que esta función es invocada. Para ello, podríamos usar un decorador.

```
def mostrar(f):
    def nueva_funcion(a, b):
        print("Esta función Llama a la función sumar ")
        print("El valor de a es:", a)
        print("El valor de b es:", b)
        print("El resultado es:", end=" ")
        return  f(a, b)
    return nueva_funcion

@mostrar
def sumar(a, b):
    return a + b

print(sumar(7, 5))
Esta función Llama a la función sumar
El valor de a es: 7
El valor de b es: 5
El resultado es: 12
```

Veamos otro ejemplo:

```
def mostrar_division(func):
    def funcion_interna(a, b):
        print("Vamos a dividir", a, "entre", b)
        if b == 0:
            print("No se puede dividir entre cero")
            return
        else:
```

```
                print("El resultados es: ", end="")
            return func(a, b)
        return funcion_interna

@mostrar_division
def dividir(a, b):
    print(a/b)

dividir(6,5)
Vamos a dividir 6 entre 5
El resultados es: 1.2

dividir(6,0)
Vamos a dividir 6 entre 0
No se puede dividir entre cero
```

En los dos ejemplos, vemos que la función decoradora, incorpora más funcionalidad a las funciones, imprime los valores de las variables e inclusive valida que el denominador no sea igual a 0.

También podemos hacer decoradores generales que funcionen con cualquier número de parámetros. En Python, esto se realiza como función `(* args, ** kwargs)`. De esta forma, `args` será la tupla de argumentos posicionales y `kwargs` será el diccionario de argumentos de palabras clave. Un ejemplo es:

```
def works_for_all(func):
    def inner(*args, **kwargs):
        print("Esta es la función decoradora")
        return func(*args, **kwargs)
    return inner
```

@wraps

Toma la función para ser decorada y añade la funcionalidad de copiar el nombre de la función, el *docstring*, los argumentos y otros parámetros asociados. Esto nos permite acceder a los elementos de la función a decorar una vez decorada.

Debemos importarla:

```
from functools import wraps

def works_for_all(func):
    @wraps(func)
    def inner(*args, **kwargs):
        print("**** Función Decoradora ****")
        print("La función " + func.__name__ + " fue llamada") #
nombre de la función
        print("Docstring: " + func.__doc__) #docstring de la
función
```

```
            print("El resultado es: ", end="")
            return func(*args, **kwargs)
        return inner

    @works_for_all
    def dividir(a, b):
        """ Función que divide 2 números"""
        print(a/b)

    dividir(6,5)
```

```
**** Función Decoradora ****
La función dividir fue llamada
Docstring:  Función que divide 2 números
El resultado es: 1.2
```

Encadenamiento de decoradores en Python

Se pueden encadenar múltiples decoradores en Python. Es decir, una función se puede decorar varias veces con decoradores diferentes (o iguales). Simplemente colocamos los decoradores encima de la función deseada.

```
    def star(func):
        def inner(*args, **kwargs):
            print("*" * 30)
            func(*args, **kwargs)
            print("*" * 30)
        return inner

    def guion(func):
        def inner(*args, **kwargs):
            print("-" * 30)
            func(*args, **kwargs)
            print("-" * 30)
        return inner

    @star
    @guion
    def printer(msg):
        print(msg)

    printer("  Bienvenidos  ")
```

```
******************************
------------------------------
  Bienvenidos
------------------------------
******************************
```

Clase Decorador

Esta sintaxis reemplaza la función por una instancia de la clase `decorator`. Si incorporamos el método `__call__` podemos ejecutar una función.

```
class Decorator(object):
    """Clase Decoradora Simple"""

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

    def __call__(self, *args, **kwargs):
        print('Antes de llamar a la función.')
        res = self.func(*args, **kwargs)
        print('Después de llamar a la función.')
        return res

@Decorator
def testfunc():
    print('Esta ejecutando la función.')

testfunc()
Antes de llamar a la función.
Esta ejecutando la función.
Después de llamar a la función.
```

Observamos que al invocar a la función `testfunc()`, primero se ejecuta la clase decoradora, además observamos que en la segunda línea del método `__call__` se ejecuta la función `testfunc`.

Métodos de decoración

Para los métodos de decoración debe definir un método `__get__` adicional:

```
from types import MethodType

class Decorator(object):
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print('Dentro del Decorador.')
        return self.func(*args, **kwargs)

    def __get__(self, instance, cls):
        # Retorna un método si es llamado por una instancia
        print("Pasa por get")
```

```
        return self if instance is None else MethodType(self,
instance)

class Test(object):
    @Decorator
    def __init__(self):
        pass
    def mostrar(self):
        print("Estas invocando el método mostrar de la clase
Text")

a = Test()
a.mostrar()
Pasa por get
Dentro del Decorador.
Estas invocando el método mostrar de la clase Text
```

Observamos que al instanciar `a` como un objeto de la clase `Test`, se ejecuta primero el método `get` y luego el método `call` de la clase decoradora. Cuando invocamos el método `a.mostrar()`, el orden de ejecución es el mismo: `get`, `call` y finalmente el método invocado.

También podemos obtener los datos de la función que estamos ejecutando, utilizando `_wrapped`:

```
class Decorator(object):
    def __init__(self, func):
        # Copia nombre, modulo, docstring para la instancia.
        self._wrapped = wraps(func)(self)

    def __call__(self, *args, **kwargs):
        return self._wrapped(*args, **kwargs)

@Decorator
def test():
    """Docstring de la función test."""
    pass

test.__doc__
'Docstring de la función test.'

test.__name__
'test'
```

El siguiente ejemplo calcula el tiempo que tarda una función en ejecutarse:

```
import time

def timer(func):
```

```
        def inner(*args, **kwargs):
            t1 = time.time()
            f = func(*args, **kwargs)
            t2 = time.time()
            print('La función tardó {0} segundos'.format(t2-t1))
            return f
        return inner

    @timer
    def example_function():
        print("Calculemos...")
        sumatoria = 0
        for i in range(100000):
            sumatoria += i
        print("El resultado es: ", round(sumatoria,2))

    example_function()
    Calculemos...
    El resultado es:  4999950000
    La función tardó 0.015974760055541992 segundos
```

Casos de uso:

Los decoradores son muy útiles inicio de sesión en un programa y autorización para utilizar una función, por ejemplo, en una aplicación web. Son muy usados en frameworks como Flask o Django. Veamos un ejemplo de cómo usar un decorador para verificar que se está autenticado.

```
    from functools import wraps

    def requires_auth(f):
        @wraps(f)
        def decorated(*args, **kwargs):
            auth = request.authorization
            if not auth or not check_auth(auth.username,
    auth.password):
                authenticate()
            return f(*args, **kwargs)
        return decorated
```

Decoradores predefinidos

`@staticmethod`: Transforma un método en un método estático. Un método estático no recibe un primer argumento implícito. Para declarar un método estático, use este modismo:

```
    class A(object):
        @staticmethod
        def metodo(*argv):
            return argv
```

`@classmethod:` Transforma un método en un método de clase. Un método de clase recibe la clase como primer argumento implícito, al igual que un método de instancia recibe la instancia.

```
class A(object):
    @classmethod
    def metodo(*argv):
        return argv
```

`@property:` es un decorador integrado para la función `property()` en Python. Se utiliza para dar una funcionalidad "especial" a ciertos métodos para que actúen como captadores, definidores o eliminadores cuando definimos propiedades en una clase. Se pueden definir tres métodos para una propiedad:

- `getter:` para acceder al valor del atributo.
- `setter:` para establecer el valor del atributo.
- `deleter:` para eliminar el atributo de instancia.

Veamos un ejemplo:

```
class House:

    def __init__(self, price):
        self._price = price

    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, new_price):
        if new_price > 0 and isinstance(new_price, float):
            self._price = new_price
        else:
            print("Introduzca un precio Válido")

    @price.deleter
    def price(self):
        del self._price
```

En el ejemplo, podemos observar que el atributo `price` ahora se considera "protegido" porque agregamos un guion bajo a su nombre en `self._price`:

```
self._price = price
```

En Python, por convención, cuando agregamos un guion bajo al inicio de un nombre, le estamos diciendo a otros desarrolladores que no se debe acceder a él ni modificarlo directamente fuera de la clase. Solo se debe acceder a él a través de intermediarios (getter o setter) si están disponibles.

```
house = House(50000.0) # Crear una instancia
house.price
50000.0

house.price = 45000.0   # Cambiamos el precio
house.price
45000.0

house.price = -50 # introducidos un valor no válido
Introduzca un precio Válido

del house.price  # eliminamos el precio
house.price
AttributeError: 'House' object has no attribute '_price'
```

Cuando definimos propiedades, podemos cambiar la implementación interna de una clase sin afectar el programa, por lo que puede agregar captadores, definidores y eliminadores que actúan como intermediarios "detrás de escena" para evitar acceder o modificar los datos directamente.

Pruebas Unitarias

Python provee el framework `unittest` que realiza pruebas unitarias de nuestro código y tiene varios métodos para realizar pruebas de código.

Admite la automatización de pruebas, el uso compartido del código de instalación y apagado para pruebas, la agregación de pruebas en colecciones y la independencia de las pruebas del marco de informes.

Para lograr esto, `unittest` admite algunos conceptos importantes de una manera orientada a objetos:

- `test fixture`: representa la preparación necesaria para realizar una o varias pruebas y las acciones de limpieza asociadas. Esto puede implicar, por ejemplo, crear bases de datos temporales o proxy, directorios o iniciar un proceso de servidor.
- `test case`: Un caso de prueba es la unidad de prueba individual. Comprueba si hay una respuesta específica a un conjunto determinado de entradas. `unittest` proporciona una clase base, `TestCase`, que se puede utilizar para crear nuevos casos de prueba.
- `test suite`: Un conjunto de pruebas es una colección de casos de prueba, conjuntos de pruebas o ambos. Se utiliza para agregar pruebas que se deben ejecutar juntas.

- `test runner`: Un ejecutor de pruebas es un componente que organiza la ejecución de pruebas y proporciona el resultado al usuario. El corredor puede utilizar una interfaz gráfica, una interfaz textual o devolver un valor especial para indicar los resultados de la ejecución de las pruebas.

Definir pruebas unitarias con unittest

Para realizar pruebas unitarias, se requiere escribir una clase que extienda de `unittest.TestCase`. Dentro de esa clase definimos funciones que serán las encargadas de probar otras funciones; una función de prueba debe estar presente por cada función.

```
# Librerías

import os
import unittest

# importamos nuestro modulo figuras geométricas

from CalcularArea import calcular_areas

class TestOperaciones(unittest.TestCase):
    def setUp(self):
        # Aquí, opcionalmente, ejecuta lo que deberías ejecutar
        # antes de comenzar cada test.
        pass

    def test_area_cuadrado(self):
        esperado = 225
        actual = calcular_areas.area_cuadrado(15)
        # Pásalo en el orden: actual, esperado
        self.assertEqual(actual, esperado)

    def test_area_rectangulo(self):
        esperado = -225
        actual = calcular_areas.area_rectangulo(-15, 15)
        # Pásalo en el orden: actual, esperado
        self.assertEqual(actual, esperado)
      …

    def tearDown(self):
        # Aquí lo contrario de setUp, cuando cada test ha terminado
        pass

if __name__ == '__main__':
    unittest.main()
```

En el código hemos utilizamos `assertEqual`, que recibe dos argumentos: el valor actual, y el valor esperado.

Los métodos `setUp` y `tearDown` son llamados antes de empezar cada test y después de terminar cada test, respectivamente.

Al final hacemos la comprobación de `if __name__ == '__main__'`, solo en caso de que el archivo sea ejecutado directamente, no importado.

La documentación provee una tabla con el resto de las funciones de `unittest` y su respectiva operación.

Función	Operación equivalente
assertEqual(a, b)	a == b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Además de la lista anterior de funciones, también se puede comprobar si nuestro método lanza una excepción en un caso determinado. Por ejemplo, si pasamos como parámetro una cadena, en este caso debería generar un TypeError. Podemos añadir la siguiente comprobación:

```
def test_area_cuadrado(self):
    esperado = 225
    actual = calcular_areas.area_cuadrado(15)
    self.assertEqual(actual, esperado)
    self.assertRaises(TypeError, calcular_areas.area_cuadrado("15"))
```

El primer argumento de `asserRaises` indica la excepción que se espera y el segundo la función que queremos probar con sus argumentos. También puede expresarse de la siguiente manera:

```
def test_area_cuadrado(self):
    esperado = 225
    actual = calcular_areas.area_cuadrado(15)
    self.assertEqual(actual, esperado)
    with self.assertRaises(TypeError):
            calcular_areas.area_cuadrado("15")
```

Cómo realizar las pruebas de Test

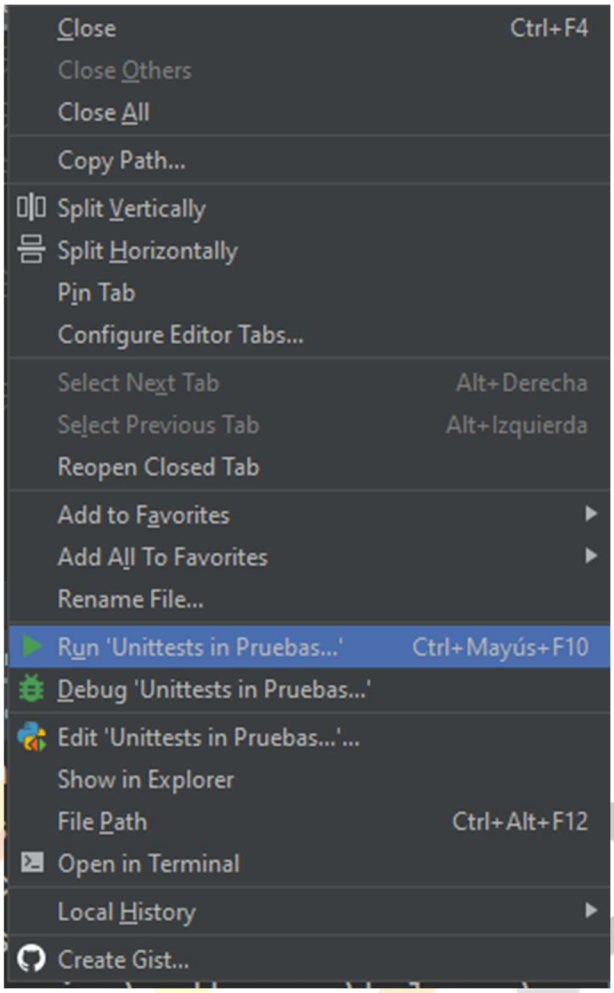
Para realizar las pruebas del test, ejecutamos nuestro archivo de pruebas desde el Terminal:

```
C:\…\> python PruebasUnitariasFG.py -v

      test_area_cuadrado (__main__.TestOperaciones) ...Paso 1:
      ejecuta lo que deberías ejecutar antes de comenzar cada test...
      Pasa cuando cada test ha terminado...
      Ok

      test_area_rectangulo (__main__.TestOperaciones) ... Paso 1:
      ejecuta lo que deberías ejecutar antes de comenzar cada test...
      Pasa cuando cada test ha terminado...
      ok

      ----------------------------------------------------------------
      -----
      Ran 2 tests in 0.001s

      OK
```

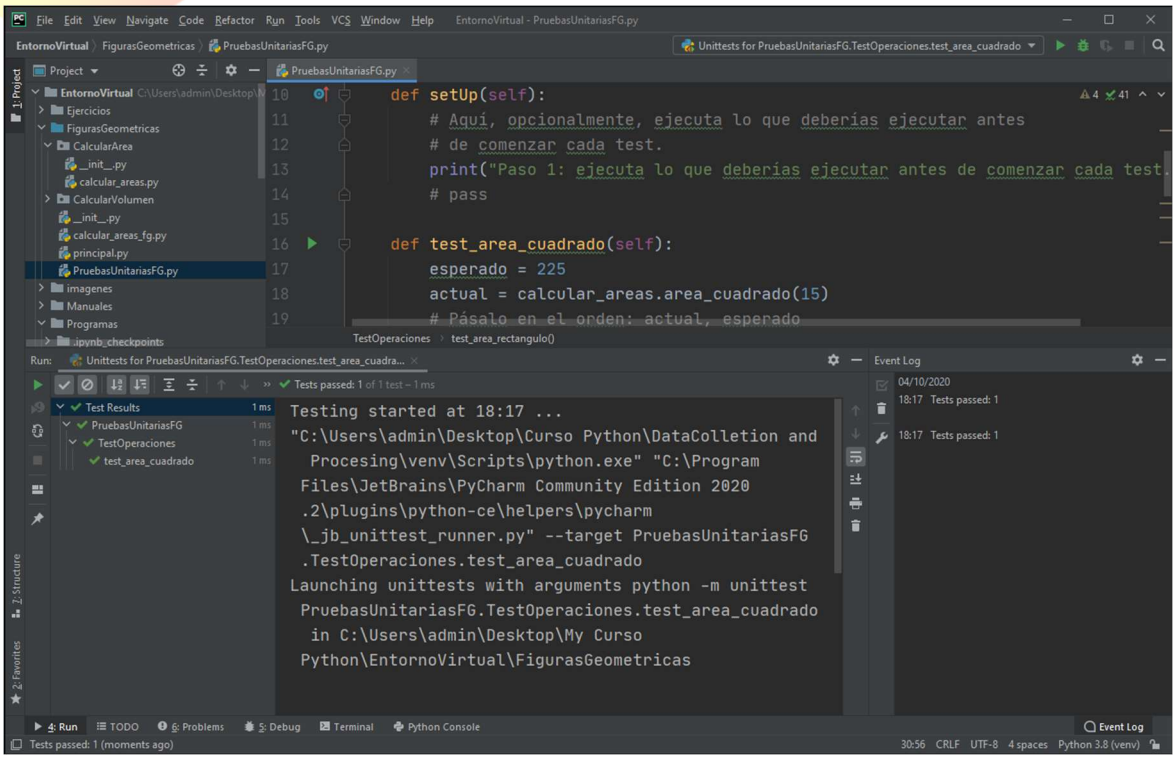
Desde los diferentes IDLE, también podemos ejecutar nuestras pruebas unitarias:

PyCharm

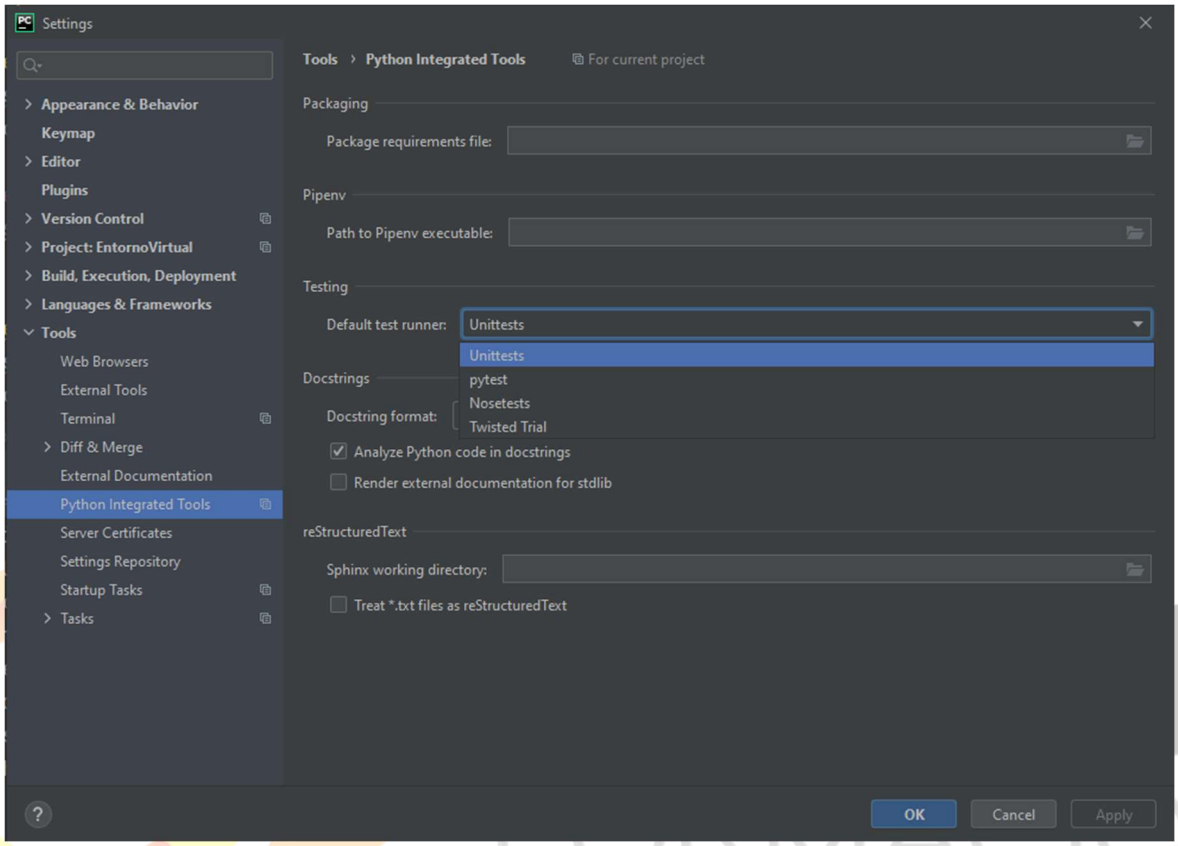
Seleccionando `Run 'Unisttests in …'`



En la parte inferior de la ventana podemos ver los resultados de los test unitarios.



En PyCharm podemos definir Unitest u otro paquete como la opción por defecto: Desde el menú Setting – Tools – Python Integrated Tools, en la sección Testing - Default test runner: selecciones el paquete por defecto.



Prueba unitaria con doctest

La implementación de la prueba unitaria utilizando `doctest` se realiza junto con la documentación de una función o clase.

```
"""
Módulo para el cálculo del área de diferentes
figuras geométricas
"""

# Librerías

import math

# Funciones para cálculo del área de figuras geométricas

def area_cuadrado(lado):
    """
    Cálcula el área de un cuadrado
    >>> area_cuadrado(15)
    225
    """
```

```
        a_cuadrado = lado**2
        return a_cuadrado

    def area_triangulo(base, altura):
        """
        Cálcula el área de un triángulo
        >>> area_triangulo(10,15)
        75.0
        """
        a_triangulo = (base*altura)/2
        return a_triangulo

    …
```

`doctest` ejecuta todas las operaciones que se encuentren luego de `>>>` y las compara con el resultado inmediatamente siguiente hasta otra operación o bien un espacio en blanco.

Esto resulta de gran comodidad, porque la documentación de nuestra función cumple dos objetivos: ilustrar con un ejemplo y servir como prueba unitaria.

Para ejecutar `doctest`, añadiremos al final de nuestro módulo las siguientes sentencias:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod(verbose=True)
```

De esta forma, cuando el módulo sea ejecutado (no importado), `doctest` analizará la documentación de todas las funciones o clases y ejecutará las pruebas correspondientes. El resultado es el siguiente:

```
Trying:
    area_cuadrado(15)
Expecting:
    225
ok
Trying:
    area_triangulo(10,15)
Expecting:
    75.0
ok
15 items had no tests:
    __main__
    __main__.TestOperaciones
    __main__.TestOperaciones.setUp
    __main__.TestOperaciones.tearDown
    __main__.TestOperaciones.test_area_cuadrado
    __main__.___
    __main__.___.setUp
    __main__.___.tearDown
```

```
        __main__.___.test_area_cuadrado
        __main__.area_circulo
        __main__.area_paralelogramo
        __main__.area_poligono_regular
        __main__.area_rectangulo
        __main__.area_rombo
        __main__.area_trapecio
    2 items passed all tests:
       1 tests in __main__.area_cuadrado
       1 tests in __main__.area_triangulo
    2 tests in 17 items.
    2 passed and 0 failed.
    Test passed.
```

Utilizando `verbose=False`, la prueba unitaria imprimirá en pantalla únicamente los mensajes de error.

Esta prueba unitaria, también puede ser ejecutada desde la terminal, sin necesidad de incluir las últimas tres líneas de código que añadimos anteriormente.

```
python -m doctest -v calcular_areas_fg.py
```

A través de la documentación de nuestra función también podemos indicar que se espera una excepción cuando intentamos sumar un entero y una cadena. Dado que los mensajes de error contienen información que varía constantemente (nombre del archivo, número de línea, etc.), no es necesario indicarlo y en su lugar se colocan tres puntos.

```
def area_cuadrado(lado):
    """
    Cálcula el área de un cuadrado
    >>> area_cuadrado(15)
    225
    >>> area_cuadrado("15")
    Traceback (most recent call last):
        ...
    TypeError
    """
    a_cuadrado = lado**2
    return a_cuadrado
```

El módulo de registro (Logging)

El módulo `logging` en Python, es un módulo potente y listo para usar. Es utilizado por la mayoría de las librerías de terceros de Python, porque puede integrar los mensajes de registro con las librerías para producir un registro homogéneo para su aplicación. Para utilizar el módulo logging, debemos importarlo:

```
import logging
```

Con el módulo logging, podemos registrar los mensajes que deseamos ver. Existen cinco (5) niveles estándar que indican la gravedad o el tipo de evento. Cada uno tiene su método correspondiente que se puede utilizar para registrar eventos de ese nivel. Los niveles definidos, en orden de gravedad creciente, son los siguientes:

- DEBUG - Depuración
- INFO - Información
- WARNING - Advertencia
- ERROR - Error
- CRITICAL – Crítico

El módulo `logging` proporciona un registrador predeterminado que nos permite comenzar sin necesidad de hacer mucha configuración. Los métodos correspondientes para cada nivel se pueden llamar de la siguiente manera:

```
import logging

logging.debug('Este es un mensaje de depuración')
logging.info('Este es un mensaje de información')
logging.warning('Este es un mensaje de advertencia')
logging.error('Este es un mensaje de error')
logging.critical('Este es un mensaje Critico')
```

```
WARNING:root:Este es un mensaje de advertencia
ERROR:root:Este es un mensaje de error
CRITICAL:root:Este es un mensaje Critico
```

La salida muestra el nivel de gravedad o tipo de mensaje antes de cada mensaje junto con la raíz, que es el nombre que el módulo. Este formato, muestra el nivel, el nombre y el mensaje separados por dos puntos (:), es el formato de salida predeterminado que se puede configurar para incluir fecha, el número de línea y otros detalles.

Observamos, que los mensajes `debug()` e `info()` no se muestran. Esto es porque de forma predeterminada, el módulo `logging` registra los mensajes con un nivel de gravedad de ADVERTENCIA o superior.

Configuraciones básicas (Basic Configurations)

Podemos utilizar el método `basicConfig(**kwargs)` para configurar logging. Algunos de los parámetros más utilizados para `basicConfig()` son los siguientes:

- `level` o nivel: La raíz de logging se establecerá en el nivel de gravedad especificado.
- `filename` o nombre de archivo: especifica el archivo.

- `filemode` o modo de archivo: Si se proporciona un nombre de archivo, el archivo se abre en este modo. El valor predeterminado es a, lo que significa append() agregar.
- `format` o formato: Este es el formato del mensaje de registro.

Mediante el parámetro `level`, puede establecer el nivel de mensajes de registro que desea registrar. Esto se puede hacer pasando una de las constantes disponibles en la clase, y esto permitiría que se registren todas las llamadas de registro en o por encima de ese nivel. Por ejemplo:

```
import logging

logging.basicConfig(level=logging.DEBUG)
logging.debug('Esto se registrará...')
```

Todos los eventos por encima o en el mismo nivel DEBUG, ahora se registrarán.

Si necesitamos guardar estos registros en un archivo en lugar de mostrarlos en la consola, podemos utilizar el nombre de archivo y el modo de archivo y decidir el formato del mensaje utilizando el formato. Por ejemplo:

```
import logging

logging.basicConfig(filename='app.log', filemode='w',
                    format='%(name)s - %(levelname)s –
                            %(message)s')
logging.warning('Estos se registrará en el archivo app.log')
```

Los mensajes ahora, se escribirá en un archivo llamado app.log. El modo de archivo se establece en `w`, lo que significa que el archivo de registro se abre en "modo de escritura" cada vez que se llama a `basicConfig()`, y cada ejecución del programa reescribe el archivo. La configuración predeterminada para filemode es `a`, que es `append`.

debug(), info(), warning(), error() y critical() también llaman a basicConfig() sin argumentos automáticamente si no se ha llamado antes. Es decir, que después de llamarlo por primera vez, ya no se puede configurar el logging raíz, porque este llama a la función basicConfig() internamente.

Formato de la salida

Aunque podemos pasar cualquier variable como una cadena, como un mensaje a los registros, hay algunos elementos básicos que ya forman parte de `LogRecord` y se pueden agregar fácilmente al formato de salida:

- Registrar el ID de proceso junto con el nivel y el mensaje::

```
import logging

logging.basicConfig(format='%(process)d-%(levelname)
```

```
                                s-%(message)s')
logging.warning('Este es una advertencia')
```

`format` puede tomar una cadena con atributos `LogRecord` en cualquier arreglo que desee.

- Agregar la hora de creación de LogRecord (%(asctime)s).

```
import logging

logging.basicConfig(format='%(asctime)s - %(message)s',
                    level=logging.INFO)
logging.info('El administrador, ha iniciado sesisón')
```

El formato se puede cambiar utilizando el atributo datefmt, que utiliza el mismo lenguaje de formato que las funciones de formato en el módulo datetime, como time.strftime():

```
import logging

logging.basicConfig(format='%(asctime)s - %(message)s',
                    datefmt='%d-%b-%y %H:%M:%S')
logging.warning('El administrador, ha cerrado su sesisón')
```

- Incluir variables:

```
import logging

name = 'John'
logging.error(f'{name} generó un error')
root - ERROR - John generó un error
```

- Incluir excepciones

```
import logging

a = 5
b = 0

try:
  c = a / b
except Exception as e:
  logging.error("Ha ocurrido una excepción", exc_info=True)

root - ERROR - Ha ocurrido una excepción
Traceback (most recent call last):
  File "<ipython-input-9-2392de1ac056>", line 7, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

CLOUD
FORMACIÓN