Talento Digital para Chile:

Módulo 3
Programación Avanzada en Python

nodovirtual.awakelab.cl

JELOU futurejob by Adalid Chile

3.2.- Contenido 2: Codificar un programa con clases, atributos y métodos utilizando colaboración y composición para resolver un problema de baja complejidad acorde al lenguaje Python

Objetivo de la jornada

1. Describe los conceptos de colaboración y composición distinguiendo sus diferencias de acuerdo al paradigma de orientación a objetos
2. Codifica una clase con métodos constructores, accesadores y mutadores para resolver un problema
3. Codifica una clase utilizando sobrecarga de métodos para resolver un problema

3.2.1.- Programación Orientada a Objetos en Python

En el módulo anterior, ya hemos visto conceptos de clase y orientación a objetos, ahora vamos a formalizar ciertas definiciones y conceptos.

3.2.1.1. Creación de una Clase en Python

Puede modelar casi cualquier cosa usando clases. Comencemos escribiendo una clase simple, Perro, que representa a un perro, no a un perro en particular, sino a cualquier perro. ¿Qué sabemos sobre la mayoría de los perros domésticos? Bueno, todos tienen nombre y edad. También sabemos que la mayoría de los perros se sientan y se dan vueltas. Esas dos piezas de información (nombre y edad) y esos dos comportamientos (sentarse y dar vueltas) irán a nuestra clase Perro porque son comunes a la mayoría de los perros. Esta clase le dirá a Python cómo hacer un objeto que represente a un perro. Después de escribir nuestra clase, la usaremos para crear instancias individuales, cada una de las cuales representa un perro específico.

Creando la clase Perro

Vamos a crear la clase Perro incluyendo de un par de métodos, además de una instancia.

```
class Perro():
    """Un simple intento de modelar un perro"""
    def __init__(self, nombre, edad):
        """Inicializar los atributos de nombre y edad."""
        self.nombre = nombre
        self.edad = edad

    def sentarse(self):
        """Simula un perro sentado en respuesta a una orden."""
        print(self.nombre.title() + " esta sentando.")

    def dar_vuelta(self):
        """Simular dar vueltas en respuesta a un comando."""
        print(self.nombre.title() + " se da vueltas")

willy = Perro('willy', 6)
willy.sentarse()
willy.dar_vuelta()
```

Resultado

```
Willy esta sentando.
Willy se da vueltas
```

Por convención, los nombres en mayúscula se refieren a clases en Python. Los paréntesis en la definición de la clase están vacíos porque estamos creando esta clase desde cero. Escribimos una cadena de documentación que describe lo que hace esta clase.

el método __init __()

Como hemos visto, una función que forma parte de una clase es un método. El método __init __() es un método especial que Python se ejecuta automáticamente cada vez que creamos una nueva instancia basada en la clase Perro. Este método tiene dos subrayados iniciales y dos subrayados finales, una convención que ayuda a evitar que los nombres de métodos predeterminados de Python entren en conflicto con los nombres de sus métodos.

Definimos el método __init __() para que tenga tres parámetros: self, nombre y edad. El parámetro self es obligatorio en la definición del método y debe ir primero antes que los demás parámetros. Debe incluirse en la definición porque cuando Python llama a este método __init __() más tarde (para crear una instancia de

Perro), la llamada al método pasará automáticamente el argumento self. Cada llamada de método asociada con una clase pasa automáticamente self, que es una referencia a la instancia misma; le da a la instancia individual acceso a los atributos y métodos de la clase. Cuando creamos una instancia de Perro, Python llamará al método __init__() de la clase Perro. Pasaremos a Perro() un nombre y una edad como argumentos; self se pasa automáticamente, por lo que no es necesario que lo pasemos. Siempre que queramos crear una instancia a partir de la clase Perro, proporcionaremos valores solo para los dos últimos parámetros, nombre y edad.

Las dos variables definidas en __init__() tienen el prefijo self.

Definición de Atributos

Cualquier variable con el prefijo self está disponible para todos los métodos de la clase, y también podemos acceder a estas variables a través de cualquier instancia creada a partir de la clase. self.nombre = nombre toma el valor almacenado en el nombre del parámetro y lo almacena en el nombre de la variable, que luego se adjunta a la instancia que se está creando. El mismo proceso ocurre con self.edad = edad. Las variables a las que se puede acceder a través de instancias como ésta se denominan **atributos**.

Definición de Métodos

La clase Perro tiene dos métodos definidos sentarse y dar_vuelta. Debido a que estos métodos no necesitan información adicional como un nombre o una edad, simplemente los definimos para que tengan un parámetro, self. Las instancias que creemos más adelante tendrán acceso a estos métodos. En otras palabras, podrán sentarse y darse vuelta. Por ahora, sentarse() y dar_vuelta no hacen mucho. Simplemente imprimen un mensaje que dice que el perro está sentado o dando vueltas. Pero el concepto puede extenderse a situaciones realistas: si esta clase fuera parte de un juego de computadora real, estos métodos contendrían código para hacer que un perro animado se siente y se dé la vuelta. Si esta clase se escribió para controlar un robot, estos métodos dirigirían los movimientos que hacen que un perro robot se siente y se dé la vuelta.

Hacer una instancia a partir de una clase

Como se ve en el ejemplo de la clase Perro. Piense en una clase como un conjunto de instrucciones sobre cómo crear una instancia. La clase Perro es un conjunto de instrucciones que le dice a Python cómo crear instancias individuales que representen perros específicos. Como hemos visto podemos crear dos instancias Perro de la siguiente forma:

```
willy = Perro('willy', 6)
bobby = Perro('bobby', 2)
```

y a la vez podemos tener acceso a sus atributos usando la notación punto

```
print("mi perro se llama " + willy.nombre.title() + " y tiene " +
str(willy.edad) + " años")
```

Resultado

```
mi perro se llama Willy y tiene 6 años
```

Como hemos visto anteriormente (Métodos Accesadores y Mutadores - getters and setters) la notación punto también nos sirve para tener acceso a los métodos:

```
willy = Perro('willy', 6)
willy.sentarse()
willy.dar_vuelta()
```

Resultado:

```
Willy esta sentando.
Willy se da vueltas
```

3.2.1.2. La sobrecarga de métodos

Qué es la sobrecarga de métodos

Una característica destacada de muchos lenguajes de programación orientados a objetos es una herramienta llamada sobrecarga de métodos. La sobrecarga de métodos simplemente se refiere a tener varios métodos con el mismo nombre que aceptan diferentes conjuntos de argumentos.

Para qué sirve

En lenguajes de tipado estático, esto es útil si queremos tener un método que acepte un número entero o una cadena, por ejemplo. En lenguajes no orientados a

objetos, podríamos necesitar dos funciones llamadas agregar_cadena y agregar_entero para adaptarse a tales situaciones. En lenguajes orientados a objetos de tipo estático, necesitaríamos dos métodos, ambos llamados agregar, uno que acepta cadenas y otro que acepta enteros.

En Python, solo necesitamos un método, que acepta cualquier tipo de objeto. Puede que tenga que hacer algunas pruebas en el tipo de objeto (por ejemplo, si es una cadena, conviértalo en un número entero), pero solo se requiere un método.

Sin embargo, la sobrecarga de métodos también es útil cuando queremos que un método con el mismo nombre acepte diferentes números o conjuntos de argumentos. Por ejemplo, un método de mensaje de correo electrónico puede venir en dos versiones, una de las cuales acepta un argumento para la dirección de correo electrónico de origen. El otro método podría buscar una dirección de remitente predeterminada. Python no permite varios métodos con el mismo nombre, pero proporciona una interfaz diferente e igualmente flexible.

Hemos visto algunas de las posibles formas de enviar argumentos a métodos y funciones en ejemplos anteriores, pero ahora cubriremos todos los detalles. La función más simple no acepta argumentos. Probablemente no necesitemos un ejemplo, pero aquí hay un ejemplo para refrescar la memoria:

```
def no_args():
    pass
```

y así es como se llama:

```
no_args()
```

Una función que acepta argumentos proporcionará los nombres de esos argumentos en una lista separada por comas. Solo es necesario proporcionar el nombre de cada argumento. Al llamar a la función, estos argumentos posicionales deben especificarse en orden y no se puede omitir ninguno. Esta es la forma más común en que especificamos argumentos en nuestros ejemplos anteriores:

```
def argumentos_obligatorios(x, y, z):
    pass
```

y para llamarla:

```
argumentos_obligatorios("un string", una_variable, 3)
```

Cualquier tipo de objeto se puede pasar como argumento: un objeto, un contenedor, un tipo primitivo, incluso funciones y clases. La llamada anterior

muestra un string codificado, una variable desconocida y un número entero pasado a la función.

Lo que hace que Python sea realmente versátil es la capacidad de escribir métodos que aceptan un número arbitrario de argumentos posicionales o de palabras clave sin nombrarlos explícitamente. También podemos pasar listas arbitrarias y diccionarios a dichas funciones.

Por ejemplo, una función para aceptar un enlace o una lista de enlaces y descargar las páginas web podría utilizar tales argumentos variables o **varargs**. En lugar de aceptar un valor único que se espera que sea una lista de enlaces, podemos aceptar un número arbitrario de argumentos, donde cada argumento es un enlace diferente. Hacemos esto especificando el operador * en la definición de la función:

```
def obtener_paginas(*enlaces):
   for enlace in enlaces:
       # bajar el enlace con urllib
       print(enlace)
```

*enlaces dice "Aceptaré cualquier número de argumentos y los pondré todos en una lista de strings con el nombre de enlaces". Si proporcionamos solo un argumento, será una lista con un elemento, si no proporcionamos argumentos, será una lista vacía. Por tanto, todas estas llamadas a funciones son válidas:

```
obtener_paginas()
obtener_paginas('http://www.archlinux.org'))
obtener_paginas('http://www.archlinux.org', 'http://ccphillips.net/')
```

También podemos aceptar argumentos de palabras clave arbitrarios. Estos llegan a la función como un diccionario. Se especifican con dos asteriscos (como en **kwargs) en la declaración de la función. Esta herramienta se usa comúnmente en la definición de configuración. La siguiente clase nos permite especificar un conjunto de opciones con valores predeterminados:

```
class Opciones:
    opciones_predeterminadas = {
           'port': 21,
           'host': 'localhost',
           'username': None,
           'password': None,
           'debug': False,
           }

    def __init__(self, **kwargs):
        self.opciones = dict(Opciones.opciones_predeterminadas)
        self.opciones.update(kwargs)
```

```
    def __getitem__(self, key):
        return self.opciones[key]
```

Todo lo interesante de esta clase ocurre en el método __init__. Tenemos un diccionario de opciones y valores predeterminados a nivel de clase. Lo primero que hace el método __init__ es hacer una copia de este diccionario. Hacemos eso en lugar de modificar el diccionario directamente en caso de que instanciamos dos conjuntos separados de opciones. (Recuerde, las variables de nivel de clase se comparten entre instancias de la clase). Luego, __init__ usa el método de actualizar en el nuevo diccionario para cambiar cualquier valor no predeterminado a los proporcionados como argumentos de palabra clave. El método __getitem__ simplemente nos permite usar la nueva clase usando la sintaxis de indexación. Aquí hay una sesión que demuestra la clase en acción:

```
opciones = Opciones(username="dusty", password="drowssap",
debug=True)

opciones['debug']
True

opciones['port']
21

opciones['username']
'dusty'
```

Podemos acceder a nuestra instancia de opciones usando la sintaxis de indexación del diccionario, y el diccionario incluye tanto los valores predeterminados como los que configuramos específicamente usando argumentos de palabras clave.

La sintaxis de argumento palabra clave puede ser peligrosa, ya que puede romper la regla "explícito es mejor que implícito". En el ejemplo anterior, es posible pasar argumentos arbitrarios de palabras clave al inicializador de Opciones para representar opciones que no existen en el diccionario predeterminado. Puede que esto no sea algo malo, dependiendo del propósito de la clase, pero hace que sea difícil para alguien que usa la clase descubrir qué opciones válidas están disponibles. También hace que sea fácil ingresar un error tipográfico confuso ("Debug" en lugar de "debug", por ejemplo) que agrega dos opciones donde solo debería haber existido una.

El ejemplo anterior no es tan malo si le indicamos al usuario de la clase que solo pase las opciones predeterminadas (incluso podríamos agregar algún código para hacer cumplir esta regla). Las opciones están documentadas en la definición de la clase, por lo que es fácil buscarlas.

Por supuesto, podemos combinar el argumento de variable y la sintaxis del argumento de palabra clave de variable en una llamada de función, y también podemos usar argumentos posicionales normales y por defecto. El siguiente ejemplo es algo artificial, pero demuestra los cuatro tipos en acción:

```
import shutil
import os.path

def mover(carpeta_destino, *archivos, verbose=False, **especifico):
    '''Mueva todos los nombres de archivo a la carpeta de destino, lo
    que permite tratamiento específico de determinados archivos.'''

    def imprima_verbose(mensaje, archivo):
        '''imprimir el mensaje solo si está habilitado verbose'''
        if verbose:
            print(mensaje.format(archivo))

    for archivo in archivos:
        ruta_destino = os.path.join(carpeta_destino, archivo)
        if archivo in especifico:
            if especifico[archivo] == 'ignore':
                imprima_verbose("Ignorando {0}", archivo)
            elif especifico[archivo] == 'copy':
                imprima_verbose("Copiando {0}", archivo)
                shutil.copyfile(archivo, carpeta_destino)
        else:
            imprima_verbose("Moviendo {0}", archivo)
            shutil.move(archivo, carpeta_destino)
```

Este ejemplo procesará una lista arbitraria de archivos. El primer argumento es una carpeta de destino y el comportamiento predeterminado es mover todos los archivos restantes de argumentos que no son palabras clave a esa carpeta. Luego hay un argumento de solo palabras clave, verbose, que nos dice si imprimimos información sobre cada archivo procesado. Finalmente, podemos proporcionar un diccionario que contiene acciones para realizar en nombres de archivos específicos; el comportamiento predeterminado es mover el archivo, pero si se ha especificado una acción de cadena válida en los argumentos de la palabra clave, se puede ignorar o copiar. Observe el orden de los parámetros en la función; primero se especifica el argumento que indica destino, luego la lista *archivos, luego un argumento específico de solo palabras clave y, finalmente, un diccionario **específico para contener los argumentos de palabras clave restantes.

Creamos una función auxiliar interna, imprima_verbose, que imprimirá mensajes solo si se ha establecido este parámetro (verbose). Esta función mantiene el código legible al encapsular esta funcionalidad en una única ubicación.

```
mover("mover_aca", "uno", "dos")
```

Este comando movería los archivos uno y dos al directorio mover_aca, asumiendo que existen (no hay verificación de errores ni manejo de excepciones en la función, por lo que fallaría espectacularmente si los archivos o el directorio de destino no existieran). El movimiento se produciría sin ningún resultado, ya que verbose es Falso de forma predeterminada.

```
mover("mover_aca", "tres", verbose=True)
Moviendo tres
```

Esto mueve un archivo, llamado tres, y nos dice qué está haciendo. Observe que es imposible especificar verbose como argumento posicional en este ejemplo; debemos pasar un argumento de palabra clave. De lo contrario, Python pensaría que es otro nombre de archivo en la lista *archivos.

Si queremos copiar o ignorar algunos de los archivos de la lista, en lugar de moverlos, podemos pasar argumentos de palabras clave adicionales:

```
mover("mover_aca", "cuatro", "cinco", "seis", cuatro="copy",
cinco="ignore")
```

Esto moverá el sexto archivo y copiará el cuarto, pero no mostrará ningún resultado, ya que no especificamos verbose. Por supuesto, también podemos hacer eso, y los argumentos de palabras clave se pueden proporcionar en cualquier orden:

```
mover("mover_aca", "siete", "ocho", "nueve", siete="copy",
verbose=True, ocho="ignore")
Copiando siete
Ignorando ocho
Moviedo nueve
```

3.2.1.3. Colaboración y Composición

Como vimos en el capítulo de encapsulamiento. La programación orientada a objetos nos ayuda a encapsular datos y procesarlos en una definición de clase ordenada. Esta encapsulación nos asegura que nuestros datos se procesan correctamente. También nos ayuda a comprender lo que hace un programa al permitirnos ignorar los detalles de la implementación de un objeto.

Qué es la colaboración entre objetos

Cuando combinamos varios objetos en una colaboración, aprovechamos el poder de la encapsulación. Veremos un ejemplo simple de creación de un objeto compuesto, que tiene varios objetos detallados en su interior.

Definición de colaboración: Definir una colaboración significa que estamos creando una clase que depende de una o más clases. Aquí hay una nueva clase, Dados, que usa instancias de nuestra clase Dado. Ahora podemos trabajar con una colección de Dados y no preocuparnos por los detalles de los objetos Dados individuales.

Primero observemos la clase Dado

```
"""Define un Dado y simula 12 lanzamientos."""
import random
class Dado(object):
    """Simular un dado genérico."""
    def __init__( self ):
        self.lados = 6
        self.lanzar()

    def lanzar( self ):
        """lanzar() -> numero
        Actualiza el dado con un lanzamiento al azar."""
        self.valor = 1+random.randrange(self.lados)
        return self.valor

    def getValor( self ):
        """getValor() -> numero
        retorna el último valor definido por lanzar()."""
        return self.valor
```

Nota: si desea probar la clase podemos hacer una llamada simple utilizando dos instancias Dado:

```
def main():

    d1, d2 = Dado(), Dado()
    for n in range(12):
         print(d1.lanzar(), d2.lanzar())

main()
```

Ahora que tenemos nuestra clase Dado procedamos a explorar la colaboración utilizando una clase Dados: Vamos a mostrar las dos clases por simplicidad para obervar

```
class Dado(object):

    """Simular un dado genérico."""

    def __init__( self ):
        self.lados = 6
```

```
        self.lanzar()

    def lanzar( self ):
        """lanzar() -> numero
        Actualiza el dado con un lanzamiento al azar."""
        self.valor = 1 + random.randrange(self.lados)
        return self.valor

    def getValor( self ):
        """getValor() -> numero
        retorna el último valor definido por lanzar()."""
        return self.valor


class Dados( object ):
    """Simular un par de dados."""

def __init__( self ):
        "Crea los dos objetos Dado."
        self.misDados = ( Dado(), Dado() )

    def lanzar( self ):
        "Retorna un numero al azar al lanzar los dados."
        for d in self.misDados:
            d.lanzar()

    def getTotal( self ):
        "Retorna el total para dos dados."
        t= 0
        for d in self.misDados:
            t += d.getValor()
        return t

    def getTuple( self ):
        "Return una tupla con el valor de los dados."
        return tuple( [d.getValor() for d in self.misDados] )

    def examinar( self ):
        "Retorna True si examinar el valor de cada objeto Dado
         comprueba que son iguales"
        return self.misDados[0].getValor() ==
self.misDados[1].getValor()
```

hagamos un test para comprobar el funcionamiento del programa

```
def test():
    x = Dados()
    for i in range(12):
        x.lanzar()
        print(x.getTotal(), x.getTuple())
```

Esta función crea una instancia de Dados, llamada x. Luego ingresa a un ciclo para realizar un conjunto de declaraciones 12 veces. El conjunto de declaraciones primero

manipula el objeto Dados usando su método lanzar. Luego accede al objeto Dados usando los métodos getTotal y getTuple.

Aquí hay otra función que usa un objeto Dados. Esta función lanza los dados 1000 veces y cuenta el número de tiradas donde los números son iguales en comparación con el número de otras tiradas con números diferentes. La fracción de lanzadas que son iguales es idealmente 1/6, 16.6%.

```
def test2():
    x = Dados()
    iguales = 0
    independientes = 0
    for i in range(1000):
        x.lanzar()
        if x.examinar(): iguales += 1
        else: independientes += 1
    print(iguales/1000., independientes/1000.)
```

Independencia. Un punto de la colaboración de objetos es permitirnos modificar una definición de clase sin romper todo el programa. Siempre que hagamos cambios en Dado (que no cambien la interfaz que utiliza Dado), podemos modificar la implementación de Dado todo lo que queramos. De manera similar, podemos cambiar la implementación de Dados, siempre que el conjunto básico de métodos esté todavía presente, somos libres de proporcionar cualquier implementación alternativa que elijamos.

Podemos, por ejemplo, reelaborar la definición de Dado confiando en que no perturbaremos a Dado o las funciones que usan Dado (ver tests). Cambiemos la forma en que representa el valor obtenido en el dado. Aquí hay una implementación alternativa de Dado. En este caso, la variable de instancia 'privada' valor, tendrá un valor en el rango 0 <= valor <= 5. Cuando getValor suma 1, el valor está en el rango habitual para un solo dado, 1≤ n ≤6.

```
class Dado(object):
    """Simula un dado de 6 lados."""
    def __init__( self ):
        self.lanzar()

    def lanzar( self ):
        self.valor = random.randint(0,5)
        return self.valor

        def getValor( self ):
        return 1 + self.valor
```

Puede realizar los tests para comprobar los cambios.

Dado que esta versión de Dado tiene la misma interfaz que otras versiones de Dado en este apartado, es isomórfica para ellas. Puede haber diferencias de rendimiento, según el rendimiento de las funciones randint y randrange. Dado que randint tiene una definición un poco más simple, puede procesarse más rápidamente.

De manera similar, podemos reemplazar Dado con la siguiente alternativa. Dependiendo del rendimiento elegido, esto puede ser más rápido o más lento que otras versiones de Dado.

```
lass Dado(object):
    """Simula un dado de 6 lados."""
    def __init__( self ):
        self.domain = range(1,7)

        def lanzar( self ):
        self.valor = random.choice(self.domain)
        return self.valor

        def getValor( self ):
        return self.valor
```

Puede realizar comprobaciones y modificaciones para solidificar la comprensión del concepto.

Qué es la composición de objetos

La composición es un concepto de diseño orientado a objetos que modela una relación "tiene". En composición, una clase conocida como compuesta contiene un objeto de otra clase conocida como componente. En otras palabras, una clase compuesta "tiene" un componente de otra clase.

La composición permite que las clases compuestas reutilicen la implementación de los componentes que contiene. La clase compuesta no hereda la interfaz de la clase de componente, pero puede aprovechar su implementación.

La relación de composición entre dos clases se considera débilmente acoplada. Eso significa que los cambios en la clase de componente rara vez afectan a la clase compuesta, y los cambios en la clase compuesta nunca afectan a la clase de componente.

Esto proporciona una mejor adaptabilidad al cambio y permite que las aplicaciones introduzcan nuevos requisitos sin afectar el código existente.

Cuando se observan dos diseños de software competidores, uno basado en la herencia y otro basado en la composición, la solución de composición suele ser la más flexible. Ahora puede ver cómo funciona la composición.

```
class Empleado:
    def __init__(self, id, nombre):
        self.id = id
        self.nombre = nombre
```

Si observa la clase Empleado, verá que contiene dos atributos:

- id para identificar a un empleado.
- nombre para contener el nombre del empleado.

Estos dos atributos son objetos que "tiene" la clase Empleado. Por lo tanto, puede decir que un empleado "tiene" una id y "tiene" un nombre.

Otro atributo para un Empleado podría ser una Dirección:

```
class Direccion:
    def __init__(self, calle, ciudad, estado, codigo_postal,
            calle2=''):
        self.calle = calle
        self.calle2 = calle2
        self.ciudad = ciudad
        self.estado = estado
        self.codigo_postal = codigo_postal

    def __str__(self):
        lineas = [self.calle]
        if self.calle2:
            lineas.append(self.calle2)
        lineas.append(f'{self.ciudad}, {self.estado},
                          {self.codigo_postal}')
        return '\n'.join(lineas)

direccion = Direccion('Chile España 31', 'Santiago', 'CL', '03301')
print(direccion)

Chile España 31
Santiago, CL 03301
```

Implementó una clase de Dirección básica que contiene los componentes habituales de una dirección. Hizo que el atributo street2 sea opcional porque no todas las direcciones tendrán ese componente.

Implementó __str__() para proporcionar una bonita representación de una dirección. Puede ver esta implementación en el intérprete interactivo:

Cuando print() la variable de dirección, se invoca el método especial __str __(). Dado que sobrecargó el método (ver apartado anterior sobre sobrecarga de metodos) para devolver una cadena formateada como una dirección, obtiene una representación agradable y legible.

Ahora puede agregar la Dirección a la clase Empleado a través de la composición:

```
class Empleado:
    def __init__(self, id, nombre):
        self.id = id
        self.nombre = nombre
        self.direccion = None
```

Inicializamos el atributo de Dirección como None por ahora para que sea opcional, pero al hacerlo, ahora puede asignar una Dirección a un Empleado.

La composición es una relación débilmente acoplada que a menudo no requiere que la clase compuesta tenga conocimiento del componente.

```
direccion_empleado1 = Direccion('Chile España 31', 'Santiago', 'CL',
'03301')
empleado1 = Empleado(1, 'Roberto', direccion_empleado1)
# inspección rápida del objeto
empleado1.__dict__
```

```
Resultado:

{'id': 1,
 'nombre': 'Roberto',
 'direccion': <__main__.Direccion at 0x1073d3ad0>}
```

Ahora vemos que el objeto empleado "tiene" un objeto dirección asociado.

Diferencia entre colaboración y composición

Hasta acá hemos visto que en la colaboración existe una sinergia entre clases Dado colabora con Dados formando parte de la implementación. Tal sinergia puede tener muchas formas, pero de forma simple vemos que existe una dependencia de Dados respecto a Dado, ambas clases tienen un nivel de acoplamiento.

Composición es más flexible, ya que, un objeto "tiene" otro objeto, propiedad, atributo, etc. pero puede prescindir de tal relación en caso de ser necesario, o bien definir tal relación como un valor predeterminado o nulo.

3.2.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[3] Eric Matthes, Python Crash Course, 2016.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/

[6] https://www.netjstech.com/2019/06/abstraction-in-python.html

[7] https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/#:~:text=Encapsulation%20is%20one%20of%20the,an%20object%20from%20the%20outside.

[8] https://pythonprogramminglanguage.com/encapsulation/

[9] https://www.linuxtopia.org/online_books/programming_books/python_programming/python_ch21s06.html