Capítulo 3-3
Modelo de objetos

1. Todo es un objeto

1.1 Principios

1.1.1 Qué sentido dar a «objeto»

Python es un lenguaje que utiliza varios paradigmas y, entre ellos, el paradigma orientado a objetos. Este se elaboró durante los años 1970 y es, ante todo, un concepto. Un objeto representa:

– un objeto físico:
  – parcela de terreno, bien inmueble, apartamento, propietario, inquilino...;
  – coche, piezas de un coche, conductor, pasajero...;
  – biblioteca, libro, página de un libro...;
  – dispositivo de hardware, robot...;
– un objeto informático:
  – archivo (imagen, documento de texto, sonido, vídeo...);
  – servicio (servidor, cliente, sitio de Internet, servicio web...);
  – un flujo de datos, pool de conexiones...;
– un concepto:
  – portador de alguna noción que pueda compartir;
  – secuenciador, ordenador, analizador de datos...

Observación

Aprovechamos para llamar la atención del lector sobre ciertos aspectos cognitivos: a partir del momento en que se modelizan personas como objetos, es fácil olvidar que detrás del código hay seres vivos que piensan, y cuya libertad puede ser limitada por las elecciones que se hacen sobre la manera de modelizar, la cantidad o la calidad de las informaciones que se decide tratar o conservar.

Aquí vemos un ejemplo muy simple y concreto: un gestor de horarios que solo permite crear franjas cada media hora puede tener una repercusión real sobre el funcionamiento de un servicio que necesita una granularidad más fina. Por lo tanto, influye de manera muy negativa en los usuarios que, antes de la llegada del software, tenían horarios que permitían más libertad.

Además, a partir del momento en que almacena datos de sus usuarios, es responsable de su conservación y seguridad; por eso tiene que asegurarse de que no se puedan filtrar, robar o incluso corromper.

Hay organismos que permiten dar una visión correcta de las reglas básicas que se deben seguir para tratar datos de este tipo, y también legislaciones, como el RGPD, que a su vez determinan algunos principios útiles.

Para modelizar como objeto, es necesario seguir algunos principios.
Uno de los principios es la encapsulación de datos. Esto significa que cada objeto posee en su seno no solo los datos que lo describen y que contiene (bajo la forma de atributos), sino también el conjunto de métodos necesarios para gestionar sus propios datos (modificación, actualización, compartición...).
El desarrollo orientado a objetos consiste, simplemente, en crear un conjunto de objetos que representa de la mejor forma posible aquello que modelan y en gestionar sus interacciones.
Cada funcionalidad se modela, de este modo, bajo la forma de interacciones entre objetos. De su correcto modelado y de la naturaleza de sus interacciones dependen la calidad del programa y también su estabilidad y mantenibilidad.
El paradigma orientado a objetos define, entonces, otros mecanismos para dar respuesta a las distintas problemáticas que se le plantean al desarrollador: polimorfismo, interfaces, herencia, sobrecarga de métodos, sobrecarga de operadores…
Es aquí donde se diferencian los lenguajes entre sí, pues cada uno propone soluciones que le son propias utilizando o no ciertos mecanismos del lenguaje orientado a objetos y de forma más o menos fiel a su espíritu.

1.1.2 Adaptación de la teoría de objetos en Python

En lenguajes como PHP, por ejemplo, se agrega una **semántica de objetos** que permite a los desarrolladores escribir de forma similar a un lenguaje orientado a objetos. Esto se realiza en dos etapas: la posibilidad de declarar clases (con interfaces y herencia simple) y la posibilidad de crear instancias de estas clases y acceder a los atributos de los métodos. Pero no es más que una semántica de objetos, puesto que detrás se trata en realidad de tablas (que contienen los atributos) que se asocian a una lista de métodos que pueden aplicarse al objeto.

La implementación está, por tanto, muy lejos de un paradigma orientado a objetos, aunque la semántica esté presente y sea suficiente para este lenguaje.

En los lenguajes orientados a objetos, como Java, el paradigma orientado a objetos está en el núcleo del lenguaje y, por tanto, de la gramática. Se han realizado adaptaciones del concepto para amoldarse a distintos escenarios técnicos o a una filosofía propia del lenguaje. No se dispone de herencia múltiple, y el concepto de interfaz se ha transformado en su totalidad para ofrecer una alternativa. Como no existen más que objetos, es necesario pasar por el proceso de bootstrap y las arquitecturas se han vuelto difíciles o restrictivas debido a limitaciones técnicas que debían respetarse.

C++ también propone sus propias adaptaciones e innovaciones. El modelo orientado a objetos que ofrece es la referencia absoluta de un lenguaje de bajo nivel estáticamente tipado.

Estas son las características esenciales que diferencian estos lenguajes del lenguaje de programación Python y que hacen que el modelo de objetos de Python sea, necesariamente, muy diferente.

Pero, además de ser diferente, el lenguaje ha tratado de aprovechar sus cualidades básicas que lo diferencian de otros lenguajes para adaptar completamente la teoría de objetos a su filosofía y encontrar aplicaciones particularmente novedosas que permitan proponer un conjunto a la vez completo, preciso y con buen rendimiento.

Por este motivo se encuentran tantas diferencias. Por tanto, la forma de trabajar de Python está completamente adaptada al lenguaje, aunque no puede decirse que el modelo de objetos de Python sea mejor que el de C++, por ejemplo. El modelo de C++ está adaptado a C++ y el de Python lo está a Python. Si se hubieran retomado los conceptos de C++ en Python, estos no habrían encontrado lugar, y viceversa.

Al final, cuando se viene de trabajar en otro lenguaje, adquirir práctica puede resultar más o menos fácil en un primer lugar, aunque para comprender realmente las diferencias y sutilidades es necesario ir más allá en el modelo de objetos, en la teoría, y comprender las elecciones realizadas y su adaptación a las características del lenguaje.

Por ello, no se deje sorprende por el hecho de que no existan las palabras clave **`new`** o **`this`**, que la firma de los métodos sea diferente, sino comprenda la filosofía general y saque provecho de las posibilidades que se ofrecen.

Python se ha creado en un momento en el que los lenguajes de referencia ya existían y habían marcado su tiempo. Ha aprovechado su experiencia y sacado el mejor provecho. A día de hoy, el propio lenguaje Python es una fuente de inspiración.

1.1.3 Generalidades

El objeto es uno de los pilares esenciales de Python, que decide proporcionar un lenguaje donde todo es un objeto, con el objetivo de responder de manera sencilla y eficaz a problemáticas complejas, permitir una gran flexibilidad y ofrecer una gran libertad de acción a los desarrolladores, como veremos en este capítulo.

Python tiene un único principio, que es «todo es un objeto», lo cual no es simplemente un concepto genérico. En efecto, si es evidente que una instancia es un objeto, el hecho de que todo sea un objeto quiere decir que la propia clase es un objeto, que un método es un objeto y que una función es un objeto.

Esto significa que todas las clases, funciones y métodos disponen de atributos y de métodos particulares, y que pueden modificarse tras su creación.
De este modo, es posible declarar clases, métodos y funciones de manera imperativa, mediante el uso de las palabras clave **`class`** o **`def`**, aunque también pueden declararse por asignación, abriendo así posibilidades muy interesantes.
Pero Python no es un lenguaje doctrinal con una única visión y buscando imponerla. Si bien el objeto está en el núcleo de sus funcionalidades, los demás paradigmas no se han rechazado o dejado de lado. Son tan importantes los unos como los otros.
En efecto, en función de la tarea que quiera cumplir, habrá una única manera evidente de proceder y aun así se podrá recurrir a uno de los tres paradigmas: imperativo, orientado a objetos o funcional.
Python no preconiza la superioridad del objeto, ni busca impedir la programación imperativa para obligar a que se utilicen objetos simplemente porque el objeto sea un enfoque más moderno o más de moda.
Es, por otro lado, interesante, cuando se conocen varios lenguajes, ver cómo Python es capaz de vincular la experiencia imperativa con la orientación a objetos y hacer emerger lo mejor de cada una.
Los debutantes que ya conozcan alguno de los paradigmas podrán desarrollar utilizando preferentemente el paradigma que conozcan y, a continuación, descubrir los demás poco a poco, en función de su experiencia.

1.2 Clases

1.2.1 Introducción

Una clase se define, simplemente, así:

```
>>> class A:
...     pass
```

Esta definición es de naturaleza imperativa, en el sentido de que una clase es un bloque que contiene un conjunto de instrucciones imperativas que se recorren y ejecutan unas detrás de otras.
Estas instrucciones pueden ser un docstring, por ejemplo:

```
>>> class A:
...     """Descripción de mi clase"""
```

Existen, en realidad, dos formas de describir una clase: bien utilizando este modo imperativo, descriptivo, que pone de relieve la encapsulación (utilizada a menudo), o bien mediante un prototipo, que también permite Python, de manera similar a JavaScript, como veremos más adelante.

1.2.2 Declaración imperativa de una clase

Una clase puede contener instrucciones declarando una variable, que se convierte en un atributo de clase, o una función, que se convierte en un método.

```
>>> class A:
...     """Descripción de mi clase"""
...     atributo = "Esto es un atributo"
...     def método(self, *args, **kwargs):
...         return "Esto es un método"
```

Una clase encapsula, así, con claridad todos sus datos, que son accesibles:

```
>>> A.__doc__
'Descripción de mi clase'
>>> A.atributo
'Esto es un atributo'
>>> A.método
<function método at 0x257ea68>
```

De este modo, el método es un atributo como los demás, pues cuando se accede sin invocarlo se devuelve la instancia correspondiente al método.
La única complejidad en la creación de clases se desprende de la complejidad funcional, del correcto modelado de objetos y de sus relaciones. Se recomienda trabajar de modo que los datos de una clase o de una instancia le pertenezcan y estén gestionados por la propia instancia, y no desde el exterior.

1.2.3 Instancia

Creemos una instancia:

```
>>> a = A()
```

Y accedamos al contenido de la instancia, definido en la clase:

```
>>> a.__doc__
'Descripción de mi clase'
>>> a.atributo
'Esto es un atributo'
>>> a.método
<bound method A.método of <__main__.A object at 0x25dfa90>>
```

Los atributos y métodos de la clase están, ahora, disponibles para la instancia, puesto que incluye un vínculo hacia los elementos de la clase, tal y como sugiere el término «bound».
Como puede verse, el hecho de acceder al método devuelve, simplemente, un objeto, aunque no invoca al método. Para realizar dicha llamada es necesario agregar los paréntesis y pasar los eventuales argumentos.
Recordemos que la firma del método espera un parámetro. Este parámetro representa, en realidad, la instancia. El vínculo entre el primer argumento del método definido en la clase y la instancia se realiza de manera natural:

```
>>> a.método()
'Esto es un método'
```

Es la gramática del lenguaje la encargada de informar el primer argumento situando la instancia. En Python no se hace magia, no existe ninguna variable mágica que represente automáticamente la instancia en curso; esta última está realmente visible y presente en la firma. Por convención, se denomina **`self`**.

La noción de interconexión entre una instancia y su clase es un elemento importante que debe dominarse. En efecto, si de una u otra manera se modifican los elementos de la clase, entonces se modifican también los elementos de la instancia:

```
>>> class B:
...     a = 'Otro atributo'
...     def m(self, *args, **kwargs):
...         return 'Otro método'
...
>>> A.atributo = B.a
>>> A.método = B.m
>>> a.atributo
'Otro atributo'
>>> a.método()
'Otro método'
```

Igual que la instancia no tiene su propio atributo, su valor es el de la clase y, como el atributo de la clase es dinámico, cualquier cambio realizado sobre este afectará a la instancia. No tener claro este aspecto puede llevarnos a generar errores inesperados. No obstante, preste atención: no es así como declaramos los atributos únicos de cada instancia, sino que se pasan al constructor, como veremos más adelante.

Informar los atributos directamente a nivel de la clase sirve, también, para que se compartan entre todas las instancias. Son, de algún modo, atributos de clase, en la semántica de Python, lo que equivale a atributos estáticos en la mayoría de los lenguajes.

Si bien estos atributos son de la clase, nada impide que un atributo con el mismo nombre aparezca en una instancia.

En este momento, podemos considerar que el atributo de la clase contiene el valor por defecto y el atributo de la instancia contiene el valor asociado de manera durable a la instancia.

Cuando el atributo de una instancia se modifica y recibe otro valor diferente al de la clase, se encuentra desconectado del atributo de la clase.

```
>>> a.atributo = 'Atributo de instancia'
>>> A.atributo
'Otro atributo'
>>> a.atributo
'Atributo de instancia'
```

El atributo de la instancia está conectado al de la clase. Ahora:

```
>>> a.atributo = A.atributo
```

Nos contentamos con realizar una asignación, sin cambiar de valor:

```
>>> A.atributo = 'B'
>>> a.atributo
'A'
```

1.2.4 Objeto en curso

Se denomina objeto «en curso» a la instancia en curso de la clase.
En Python, dicha instancia se denomina **`self`**, aunque no es más que una convención. Lo que importa es que el objeto en curso es, sistemáticamente, el primer objeto que recibe como parámetro un método, y dicho vínculo se establece de forma automática.
En la mayoría de los lenguajes existe una palabra clave **`this`** que permite ejecutar un método como una función, un poco de forma mágica, pues **`this`** representa a la instancia en curso.
Como a Python no le gusta la magia y quiere preservar la legibilidad, se contenta con exigir un primer argumento que representa a la instancia y el vínculo se establece a bajo nivel, pero no hay ningún elemento mágico de por medio. Lo que se utiliza en la función es, simplemente, variables que se presentan en la firma del método.
Cabe destacar que no se utiliza la palabra clave **`this`** ni la palabra clave **`new`** para crear la instancia.

1.2.5 Declaración por prototipo de una clase

La programación orientada a objetos por prototipo consiste en crear una clase y, a continuación, asignarle atributos y métodos como se hace, por ejemplo, en JavaScript.
Esto es muy diferente a la programación orientada a objetos clásica, puesto que nos contentamos con declarar una clase que es un recipiente vacío con un nombre y, a continuación, se le agregan atributos y métodos.
Estos métodos pueden ser, para ciertos lenguajes, simples funciones que transforman un objeto que se pasa como parámetro o que reciben un objeto como parámetro para devolver otro objeto sin que exista ningún vínculo entre ellos, salvo el hecho de agregarse en la misma clase.
El recurso de una palabra clave permite, por tanto, crear un vínculo artificial pero suficiente entre los métodos de una misma clase y sus propiedades. Esto puede parecer una agregación de propiedades y de funciones similares a lo que serían atributos y métodos.
Semánticamente, el uso de una clase así es idéntico al de una clase declarada de manera clásica, aunque los mecanismos internos sean totalmente distintos.
Esto no entra, en absoluto, en el espíritu de la programación orientada a objetos, pues si bien la encapsulación se resuelve de una manera diferente, aunque comprensible, los demás mecanismos tales como la instanciación, la diferenciación de instancias o el polimorfismo, por ejemplo, no pueden resolverse, o bien se resuelven de manera poco satisfactoria. Además, ciertos lenguajes hacen todas las clases puramente estáticas.
Estos lenguajes son, entonces, una interpretación del paradigma orientado a objetos bastante reducida, aunque por el contrario representan una ventaja indiscutible, que es la capacidad evolutiva, dado que, en cualquier momento, es posible agregar o modificar funciones.
En efecto, en la mayoría de los lenguajes, una vez declarada la clase, es imposible agregar nuevos métodos o atributos. En ocasiones, una permisividad natural permite agregar atributos de manera lateral. No obstante, esto es una limitación importante que hace que la programación orientada a objetos por prototipo encuentre su verdadero lugar.

En lo relativo a Python, esto es muy distinto. Por un lado, su lectura extrema del paradigma orientado a objetos hace que las propias clases, funciones y métodos sean objetos sobre los que es posible actuar como con cualquier otro objeto. Por otro lado, el hecho de que sea dinámico implica que, en todo momento, sea posible realizar una asignación o una modificación.
De este modo, es posible declarar una clase y, a continuación, añadir más tarde un atributo, por agregación. Para comenzar, creemos una clase de manera declarativa, como hemos hecho hasta ahora:

```
>>> class Declarativa(object):
...     """Clase escrita de manera declarativa"""
...
...     atributo_de_clase = 42
...
...     def __init__(self, name):
...         self.name = name
...         self.subs = []
...
...     def __str__(self):
...         return "{} ({})".format(self.name, ", ".join(self.subs))
...
 ...    def mostrar(self):
...         print(self)
```

Ahora podemos utilizar este objeto:

```
>>> a = Declarativa("test")
>>> a.subs.append("cosa", "chisme")
>>> print(a)
test (cosa, chisme)
>> dir(a)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
>>> Declarativa.mostrar
<function Declarativa.mostrar__ at 0x7fb456895bf8>
```

Presentaremos, ahora, el código equivalente al anterior, escrito mediante prototipo. Veremos que primero se escriben los métodos:

```
>>> def proto__init__(self, name):
...     self.name = name
...     self.subs = []
...
>>> def proto__str__(self):
...     return "{} ({})".format(self.name, ", ".join(self.subs))
...
>>> Prototipo = type("Prototipo", (object,), {
```

```
    "__init__": proto__init__,
    "__str__": proto__str__,
    "atributo_de_clase": 42})
```

También es posible agregar funciones más tarde:

```
>>> def mostrar(self):
...     print(self)
...
>>> Prototipo.mostrar = mostrar
```

El resultado es completamente idéntico a nuestra clase declarada de manera clásica:

```
>>> dir(Prototipo)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', 'atributo_de_clase']
```

Esta forma de operar no es un error de programación o de diseño, es algo natural y que está previsto en Python.

Como un método no es más que una función encapsulada en una clase (si bien sigue algunas reglas particulares suplementarias que se presentan en la sección Métodos), Python no tiene ningún problema con esta forma de trabajar.

```
>>> def m(self):
...     return "Definido por prototipo"
...
>>> A.método = m
```

Esta forma de trabajar es bastante diferente a la de los lenguajes específicamente cualificados como «programación orientada a objetos por prototipo», como por ejemplo JavaScript, uno de los más conocidos y utilizados en este dominio.

Sin embargo, es bastante limitada en Python, a parte de las librerías que utilizan masivamente nociones complejas, tales como metaclases; por ejemplo, para resolver requisitos específicos de diseño.

La gran ventaja de esta técnica es que permite modificar las clases en cualquier momento, o extenderlas tanto como se quiera. Podemos perfectamente declarar una clase de la manera habitual, y después, en otro módulo importarla y agregarle métodos o atributos.

1.2.6 Tuplas con nombre

Existen muchos casos de uso en los que se necesita la flexibilidad de un objeto pero no se desea pasar demasiado tiempo escribiendo una clase. Para ello, existen las tuplas con nombre:

```
>>> from collections import namedtuple
>>> Punto = namedtuple('Punto', ['x', 'y'])
```

`Punto` es una clase particular que dispone de dos atributos **`x`** e **`y`**. Puede instanciarse:

```
>> p = Punto(4, 2)
```

Capítulo 3
Preparar los datos con Pandas y NumPy

1. Pandas, la librería de Python imprescindible para la manipulación de datos

Ahora estamos listos para explorar y analizar nuestros datos. Para ello, nos apoyaremos en uno de los módulos más imprescindibles de Python: Pandas.

Pandas es la librería de Python para manipular y analizar datos. Fue creada en 2008 por Wes McKinney, un estadístico y desarrollador de Python. Esta librería se ha ido consolidando poco a poco como un elemento esencial para la gestión de datos y ha contribuido a hacer de Python un referente en este campo.

1.1 Instalación

Para empezar, asegurémonos de que esté bien instalada. Si no es así, validemos la siguiente línea en un símbolo del sistema:

```
pip install pandas
```

Una vez instalada, basta con importarla:

```
import pandas as pd
```

El alias pd se usa comúnmente para simplificar los comandos relacionados con Pandas y sirve para asegurarse de que los comandos de Pandas que ejecutemos a continuación se refieran a ella.

1.2 Estructura y tipo de datos

Antes de usarla, dediquemos unos momentos a explicar las diferentes estructuras que puede tomar Pandas. La que todo el mundo tiene en mente es el DataFrame de dos dimensiones que parece una hoja de cálculo de Excel, pero, en realidad, hay un tipo por dimensión:

Nombre de la estructura	Número de dimensiones	Principio
Series	1	Datos unidimensionales indexados
DataFrames	2	Hoja de cálculo con filas y columnas
Panels	3	Colección de DataFrames
DataFrames multi-index	4	DataFrames con multiíndice para filas o columnas

Observación

También hay una estructura `panel4D` de cuatro dimensiones, pero está obsoleta y no se recomienda su uso.

En la gran mayoría de los casos, solo se encontrará con series y DataFrames, pero es útil conocer la existencia de las otras estructuras.

La indexación está en el núcleo de estas estructuras, de una manera mucho más presente que en las hojas de cálculo. Proporciona un fácil acceso a filas, columnas o celdas. Pandas ofrece varias formas de hacerlo al permitir referirse a ellas ya sea por su índice o por su nombre.

Una pequeña aclaración útil: la indexación comienza en 0, y no en 1.

En la tabla siguiente se muestran las cuatro formas de hacerlo:

Acción	iloc	loc	at	query
Acceso a una celda	`df.iloc[2,1]`	`df.loc[2, 'Col2']`	`df.at[2, 'Col2']`	Inapropiado
Acceso a una fila	`df.iloc[2, :]`	`df.loc[2, :]`	Inapropi_ado	`df.query ("índice==2")`
Acceso a una columna	`df.iloc[ :,1]`	`df.loc[ :, 'Col2']`	Inapropiado	Inapropiado
Filtrado con condiciones	`mascara = df['A'] > 2 df.iloc[mascara .values, 1]`	`mascara = df['A'] > 2 df.loc[mascara.values, "B"]`	Inapropiado	`df.query ("A>2")`

Observación

La gran diferencia entre `iloc` y `loc`, que son las funciones más utilizadas, es que todo es numérico con `iloc`, mientras que `loc` requiere que especifique el nombre de la variable.

1.3 Posibilidades que ofrece

Más allá del aspecto estructural, veamos juntos lo que ha hecho que Pandas tuviera éxito: su capacidad para satisfacer todas las necesidades inherentes a la manipulación de datos.

Desde el inicio, Pandas garantiza la posibilidad de leer casi cualquier tipo de archivo. Además de los formatos más clásicos, como archivos de texto, CSV, Excel o JSON, también permite leer SQL, archivos propietarios como SAS o SPSS y otros formatos que nos encontraremos más adelante, como HDF5, Parquet o Pickle.

Una vez adquiridos los datos, podemos acceder fácilmente a diferente información, como las dimensiones, el tipo de variable, el número de observaciones distintas de cero por columna o estadísticas descriptivas básicas.

A continuación, Pandas nos ofrece toda la gama de funciones para preparar los datos: eliminar observaciones o columnas, imputar, reemplazar, fusionar otros archivos o crear nuevas variables.

A esto se suman sus capacidades de visualización gráfica, muy prácticas para comprender los datos. Sin embargo, para necesidades más avanzadas, es posible que prefiramos las funcionalidades que ofrecen los módulos dedicados, como Matplotlib o Seaborn.

Esta interoperabilidad con el resto de los módulos es otro de los pilares que hace fuerte a Pandas. Además de las librerías gráficas mencionadas con anterioridad, Pandas interactúa perfectamente con todos los módulos de Python dedicados a la ciencia de datos, como Scikit-Learn o NumPy. Vamos a presentar este último, que actúa a la sombra de Pandas.

2. NumPy, el pilar del cálculo numérico

NumPy es una librería fundamental para el cálculo científico en Python, que actúa como una infraestructura básica para muchas otras librerías. Tomémonos el tiempo para ver juntos qué lo hace fuerte y por qué es omnipresente en los módulos de Python.

2.1 La estructura ndarray

El ndarray, abreviatura de N-dimensionnal array, está realmente en el núcleo de la librería Nupy. También se proponen otras dos estructuras, la matriz y los escalares, pero centraremos nuestra atención en el ndarray, ya que su papel es central.

Observación

Preste atención: muchos nombres diferentes se pueden referir a un ndarray. El término tabla es la forma genérica para calificarlos, pero podemos encontrarnos con el término vector para ndarrays de una dimensión o matriz para los que tienen dos. Más allá de dos dimensiones, es común encontrarse con los nombres: array, NDArray o tensor.

Empecemos por estudiar las características de esta estructura fundamental de NumPy.

2.1.1 Una estructura homogénea

En primer lugar, es una tabla multidimensional homogénea, es decir, puede tener tantas dimensiones como se desee, con la restricción de que todos los datos sean del mismo tipo. He aquí una manera fácil de crear un vector ndarray a partir de una simple lista de Python:

```
import numpy as np
a = [1, 2, 3, 4, 5]
array_numpy = np.array(a)
print(array_numpy.dtype)

# Output: int64
```

En cuanto a la obligación de homogeneidad mencionada anteriormente, se debe tener en cuenta que la introducción de un miembro de tipo coma flotante conduce de facto al cambio del tipo de todo el ndarray:

```
array_numpy_2 = np.array([1.0, 2, 3, 4, 5])
print(array_numpy_2.dtype)

# Output: float64
```

Este ejemplo solo tenía como objetivo crear un ndarray a partir de una estructura familiar. Ahora que ya tenemos confianza, veamos cómo crearlo directamente:

```
inicio = 0
fin = 10 #valor no incluido
incremento = 2
array_numpy = np.arange(inicio, fin, incremento)
# El incremento puede ser decimal

print(array_numpy)
# Output: [0 2 4 6 8]
```

Y el abanico de posibilidades no se detiene ahí. El comando `linspace`, por ejemplo, permite crear un vector con un número de valores de intervalo idénticos, lo cual es muy útil a la hora de elaborar gráficos:

```
array_linspace = np.linspace(0,100,15)
```

Esta simple línea de código generará un vector entre 0 y 100 inclusive, con 15 valores en total a intervalos iguales.

Ahora veamos cómo crear matrices bidimensionales.

Esto se puede hacer directamente desde una tabla de Pandas de la forma más sencilla posible:

```
# df es un DataFrame pandas
array_from_pandas = df.values
```

Aquí, hemos recuperado todo el DataFrame, pero solo podríamos haber obtenido una variable como vector de esta manera:

```
array_col_pandas = df['nombre_columna'].values
```

O incluso:

```
array_cols_pandas = df[['nombre_columa1', 'nombre_columna2',
'nombre_columna3']].values
```

También se nos ofrece la posibilidad de crear una matriz fijando las dimensiones y el contenido. Como una matriz nula, rellena de 0, usando el comando `zeros`:

```
num_rows = 10
num_columns = 10

shape = (num_rows, num_columns)
matriz_zeros = np.zeros(shape)
```

Los comandos `ones` y `full` hacen lo mismo, pero con 1 o un valor específico establecido para todas las casillas, respectivamente:

```
# Matriz rellena de 1
matriz_1 = np.ones(shape)

# Matriz rellena de 100 (Todos los números son posibles):
matriz_100 = np.full(shape, 100)
```

Por último, cabe destacar las diferentes posibilidades que ofrece para generar números aleatorios, ya sean enteros o decimales.

Para los números decimales, podemos solicitar una matriz de números uniformes entre 0 y 1 usando `random.rand`:

```
# Ejemplo de una matriz de 3 filas en 4 columnas:
matrice_rand = np.random.rand(3,4)
```

El uso nos permitirá obtener números gaussianos, es decir, números que tienen una media de 0 y una desviación típica de 1:

```
# Ejemplo de una matriz de 3 filas en 4 columnas:
matriz_rand = np.random.rand(3,4)
```

La creación de números enteros requerirá una operación adicional: el uso de la palabra clave `reshape` porque no es posible definir las dimensiones directamente. Indicaremos en `reshape` el número de filas y columnas que queremos:

```
# Crear una lista de enteros aleatorios
random.randint(low, high=None, size=None)
randint_numpy = np.random.randint(0,50,100)
# Transformar en una matriz de 10 filas por 10 columnas
randint_numpy = randint_numpy.reshape(10,10)
```

Esta matriz se podría haber creado directamente en una línea encadenando comandos:

```
randint_numpy = np.random.randint(0,50,100).reshape(10,10)
```

Observación

Asegúrese de que el producto de las dimensiones de la nueva forma producida con reshape corresponda exactamente a la longitud de la lista de números aleatorios; de lo contrario, tendremos un error.

```
array([[ 0, 39, 24, 36, 35,  5,  6,  3, 34, 40],
       [33, 28,  4, 26, 32, 45,  9,  5, 33,  7],
       [30,  8, 20,  7,  3, 21, 27, 44,  3, 38],
       [20,  7, 19, 31,  0,  5, 27, 43, 30,  9],
       [19,  7, 21, 37, 28,  8, 43, 46,  0, 40],
       [38, 25, 10, 34, 23, 32, 19, 26, 14, 32],
       [ 6, 33, 44, 45, 41,  4, 29, 27, 17, 35],
       [ 2, 20, 45, 15, 36, 41,  4, 49, 13, 30],
       [45, 23, 34, 35,  9, 26, 26, 22, 12, 15],
       [34, 26, 38, 46, 16, 47, 40,  0, 10, 11]])
```

Antes de cerrar este punto, señalemos una posibilidad común a todas las estructuras de NumPy: la de definir el dtype que impactará en el rendimiento. Así, cuanto mayor sea el espacio asignado en bits, mayor será la precisión, pero a expensas del tiempo de cálculo y viceversa.

Veamos las consecuencias de cambiar los tipos en una matriz:

```
# Definición de la semilla de aleatoriedad para obtener el mismo
resultado
np.random.seed(0)

m1 = np.random.randn(3,4)
m2 = m1.astype(np.float16) # Modificación del tipo de 64 a 16 bits

print("m1:\n",m1)
print("m2:\n",m2)
```

```
m1:
 [[ 1.76405235  0.40015721  0.97873798  2.2408932 ]
 [ 1.86755799 -0.97727788  0.95008842 -0.15135721]
 [-0.10321885  0.4105985   0.14404357  1.45427351]]
m2:
 [[ 1.764   0.4001  0.9785  2.24  ]
 [ 1.867  -0.977   0.95   -0.1514]
 [-0.1032  0.4106  0.144   1.454 ]]
```

Después de esta exploración de las diferentes estructuras, veamos cómo acceder a los datos.

2.1.2 Indexación

Al igual que en las listas de Python, los elementos de las matrices se indexan, lo que facilita el acceso. Pero la comparación con las listas simples de Python termina ahí, ya que los ndarrays ofrecen mucha más velocidad y opciones.

Empecemos por averiguar cómo acceder a nuestros datos.

El acceso a un elemento específico, como parte de una matriz bidimensional, consiste simplemente en proporcionar el número de fila y de columna:

```
print(m2[0,1]) # Mostrar el elemento fila 0 / columna 1
       # Output: 0.4001
```

Si queremos acceder a una fila o una columna en particular, basta con usar los «:» para incluir todos los elementos en cuestión:

```
print(m2[0, :]) # Mostrar la primera fila (de índice 0)
# Output: array([1.764 , 0.4001, 0.9785, 2.24 ], dtype=float16)
```

```
print(m2[ :, 1]) # Mostrar la segunda columna (de índice 1)
# Output: array([ 0.4001, -0.977 , 0.4106], dtype=float16)
```

También podemos usar los «:» como en el contexto de las listas, para mostrar varias filas o columnas, como aquí:

```
print(m2[ :, 1:3]) # Mostrar las columnas de índice 1 y 2


# Output:
array([[ 0.4001,  0.9785],
       [-0.977 ,  0.95  ],
       [ 0.4106,  0.144 ]], dtype=float16)
```

Por último, el acceso a las columnas o filas discontinuas se realiza a través de una lista como la siguiente:

```
print(m2[:, [0,3]] # Mostrar las columnas de índice 0 y 3

# Output:
array([[ 1.764 ,  2.24  ],
       [ 1.867 , -0.1514],
       [-0.1032,  1.454 ]], dtype=float16)
```

Invitamos a aquellos que aún no están muy familiarizados con este tipo de indexación a que se tomen el tiempo para practicarla porque NumPy la empleará constantemente durante las diferentes etapas.

Aprovechemos este punto sobre la indexación para abordar el tema de las operaciones con tablas, que encontraremos con frecuencia.