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 sus datos para sacarles todo su potencial

1. La calidad de los datos: un recordatorio

La calidad de los datos es un elemento fundamental que hay que tener en cuenta antes de abordar técnicas de limpieza y procesamiento de datos. Para cualquier organización que quiera tomar decisiones con conocimiento de causa y sacar el máximo partido de su información, este aspecto no puede pasarse por alto. Todos los procedimientos que examinaremos en este capítulo tienen un único objetivo: proporcionar a los distintos equipos datos fiables.

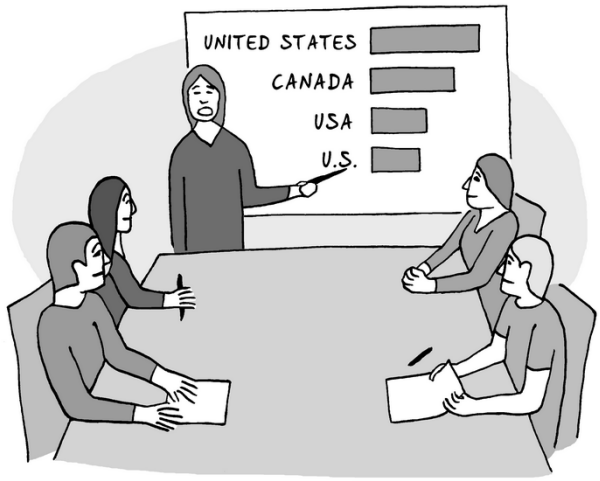
1.1 ¿Qué es la calidad de los datos?

La calidad de los datos refleja la capacidad de una organización para mantener la exactitud y sostenibilidad de su información a lo largo del tiempo. Como expertos en la materia, debemos ofrecer datos irreprochables, basados en indicadores claros y fácilmente interpretables. Empezaremos por examinar en detalle los seis criterios que definen las dimensiones de calidad de los datos (DQD).

Esta noción engloba tanto las características intrínsecas de los datos como los métodos aplicados para garantizarlas. En esencia, la calidad de los datos se define por su capacidad para servir a los fines previstos.

Una iniciativa de calidad de datos es un proceso a largo plazo, integrado en todo el ciclo de vida de los datos. Requiere un cambio cultural en la forma en que la organización gestiona sus datos. Es un enfoque global que afecta a toda la empresa y a sus prácticas cotidianas.

Es importante señalar que la introducción de datos erróneos en un proceso producirá inevitablemente datos inexactos en la salida. Por consiguiente, una estrategia basada en datos de mala calidad dará lugar a decisiones ineficaces, con consecuencias directas en el retorno de la inversión.



Créditos: https://dataedo.com/

1.2 ¿Por qué es importante la DQD?

La calidad de los datos suele verse comprometida por diversos factores. Entre ellos se encuentra el error humano en el momento de su introducción inicial. Errores tipográficos, diferentes convenciones de nomenclatura entre fuentes de datos o abreviaturas incorrectas son una fuente frecuente de problemas. Además, una información inicialmente exacta puede quedar obsoleta con el tiempo, al cambiar el contexto.

Si la calidad de los datos se ve comprometida, habrá consecuencias costosas para las empresas. He aquí algunos ejemplos concretos de problemas frecuentes y sus repercusiones:

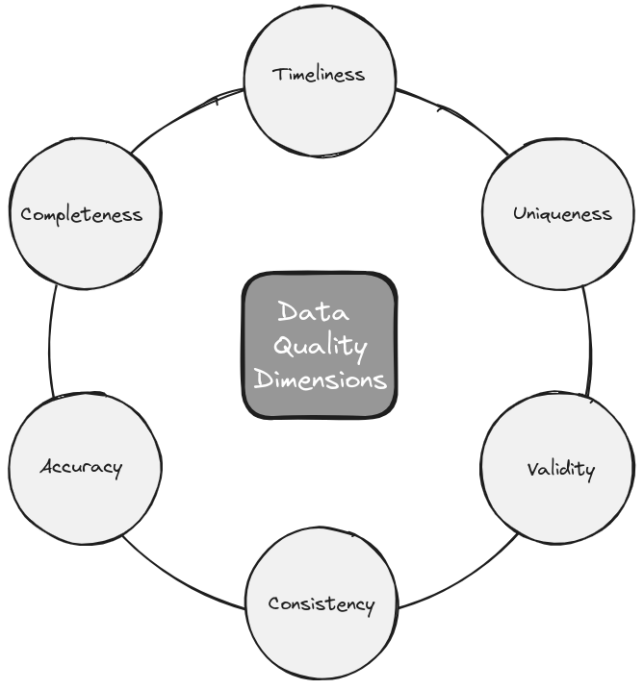
– Errores de introducción de datos: en 2018, un empleado de Samsung Securities cometió un monumental error de introducción de datos al distribuir dividendos, emitiendo inadvertidamente 2.800 millones de acciones «fantasmas», unas treinta veces el número total de acciones existentes. Este error provocó una perturbación masiva del mercado y una pérdida de confianza en la gestión de la empresa, lo que obligó a adoptar costosas medidas correctoras. [Fuente: https://www.sirfull.com/en/blog/poor-data-quality-impacts/]
– Incoherencias en los datos: en 2022, Unity Technologies se enfrentó a un grave problema con su herramienta de publicidad dirigida Pinpointer. Los datos inexactos de un cliente importante corrompieron los modelos de IA, lo que provocó una pérdida de ingresos de 110 millones de dólares y una caída del 37 % en el valor de las acciones de la empresa. [Fuente: https://zeenea.com/what-are-the-most-common-data-quality-issues-and-how-can-you-solve-them/]

– Datos obsoletos: Tesco, el gigante británico de la distribución, tuvo problemas de inexactitud en sus datos de existencias, lo que provocó frecuentes desabastecimientos en sus tiendas. Estas imprecisiones provocaron pérdidas de ventas, insatisfacción de los clientes y un impacto negativo directo en el volumen de negocios de la empresa. [Fuente: https://www.sirfull.com/en/blog/poor-data-quality-impacts/]
– Errores críticos en los datos: entre marzo y julio de 2017, Equifax generó puntuaciones crediticias incorrectas para millones de consumidores debido a datos erróneos. La empresa se enfrentó a multas reglamentarias, demandas judiciales y una pérdida de credibilidad, lo que puso en peligro las decisiones de concesión de préstamos y erosionó la confianza del público. [Fuente en francés: https://www.datagalaxy.com/fr/blog/big-data-attention-aux-donnees-de-mauvaise-qualite/]
– Datos de localización inexactos: Royal Dutch Shell tuvo errores en los datos de localización de sus pozos petrolíferos, lo que provocó una perforación ineficaz. Estos errores supusieron millones de dólares en costes adicionales y una considerable pérdida de tiempo debido a una perforación incorrecta. [Fuente en francés: https://solutions-business-intelligence.fr/data-quality-enjeux-et-bonnes-pratiques/]

Cada uno de estos problemas de calidad de datos tuvo un impacto significativo en las operaciones, la reputación y los resultados financieros de las empresas implicadas. Estos ejemplos ponen de relieve la importancia de invertir en procesos sólidos de gestión de la calidad de los datos para evitar estos costosos escollos.

1.3 Los principales criterios de la DQD

Veamos los principales criterios de la DQD, núcleo de cualquier estrategia eficaz de gestión de datos.



1.3.1 Precisión (accuracy)

La precisión es el grado en que los datos almacenados se ajustan a los valores reales que deben representar. Mide lo cerca que está el valor almacenado del valor real correcto o aceptado.

En el sector bancario, imaginemos un error en el cálculo de los intereses de un préstamo hipotecario. Si el tipo de interés se registra incorrectamente como 3,5 % en lugar de 3,05 %, esto podría dar lugar a que se cobrara de más a miles de clientes. No solo podría acarrear importantes pérdidas financieras para el banco en caso de reembolso, sino que también podría dañar gravemente su reputación y dar lugar a posibles sanciones.

Para garantizar la exactitud, las empresas pueden poner en marcha procesos de validación de datos, comprobaciones cruzadas automatizadas y auditorías periódicas. El uso de inteligencia artificial para detectar anomalías también puede ser eficaz.

1.3.2 Integridad (completeness)

La integridad es el grado de presencia de todos los elementos de datos necesarios en un conjunto de datos concreto. Evalúa el grado en que se incluyen todos los valores requeridos y están presentes todos los registros que deberían estarlo.

En el campo de la investigación médica, imaginemos un estudio sobre la eficacia de un nuevo tratamiento contra el cáncer. Si los datos de seguimiento posteriores al tratamiento están incompletos para un grupo significativo de pacientes, las conclusiones del estudio podrían estar distorsionadas. Las decisiones cruciales sobre si aprobar o rechazar el tratamiento podrían basarse en información parcial, lo que podría afectar a la vida de muchos pacientes.

El uso de campos obligatorios en los formularios de introducción de datos, la creación de alertas para los datos que faltan y la aplicación de procesos sistemáticos de recopilación de datos pueden mejorar la exhaustividad.

1.3.3 Coherencia (consistency)

La coherencia se refiere a la ausencia de contradicciones en los datos dentro de un conjunto de datos o entre distintos conjuntos de datos. Garantiza que los datos sean uniformes y lógicamente compatibles en todos los sistemas, aplicaciones y procesos de la organización.

En una empresa multinacional, imagine que el departamento de recursos humanos y el de finanzas utilizan sistemas diferentes para gestionar la información de los empleados. Si un empleado cambia de trabajo y esta información solo se actualiza en el sistema de recursos humanos, podrían producirse errores en las nóminas, las prestaciones e incluso en la planificación estratégica de recursos humanos.

El uso de un sistema de información integrado, la aplicación de procesos automáticos de sincronización entre distintos sistemas y el establecimiento de normas coherentes de gestión de datos en toda la empresa pueden mejorar la coherencia.

1.3.4 Vigencia (timeliness)

La vigencia mide el grado en que los datos están actualizados y disponibles en el plazo requerido para su uso previsto. Evalúa la frescura de los datos en relación con el momento en que se crearon o actualizaron por última vez, así como su disponibilidad en el momento en que se necesitan para los procesos empresariales.

En el trading de alta frecuencia, donde las decisiones de compra y venta se toman en milisegundos, la vigencia de los datos es fundamental. Un retraso de solo unos segundos en la actualización de los precios de las acciones puede acarrear considerables pérdidas financieras. Por ejemplo, si un acontecimiento importante afecta a la cotización de una acción, pero esta información no se refleja inmediatamente en los datos utilizados por los algoritmos de negociación, podrían tomarse decisiones de inversión desastrosas.

El uso de sistemas de procesamiento en tiempo real, la implantación de procesos automáticos de actualización de datos y la optimización de los flujos de datos pueden mejorar la puntualidad.

1.3.5 Validez (validity)

La validez es la medida en que los datos se ajustan a las normas empresariales definidas, los formatos especificados y las restricciones del dominio. Garantiza que los valores de los datos cumplen los criterios sintácticos y semánticos establecidos para el tipo de datos en cuestión.

En el sector de la aviación, imaginemos un sistema de reservas que acepte una fecha de vuelo anterior a la actual. Esto podría acarrear graves problemas en la planificación de vuelos, la gestión de tripulaciones y, potencialmente, comprometer la seguridad si estos datos no válidos se utilizan en otros sistemas críticos.

La aplicación de controles de validación estrictos en las interfaces de entrada, el uso de restricciones a nivel de base de datos y el establecimiento de procesos periódicos de limpieza de datos pueden mejorar la validez.

1.3.6 Singularidad (uniqueness)

La singularidad es la propiedad de que cada entidad distinta del mundo real se represente una y solo una vez en el conjunto de datos. Garantiza que no haya duplicados involuntarios ni redundancias en los registros de datos.

En el sector sanitario, imagine un paciente con varios historiales médicos debido a duplicados en la base de datos. Esto podría dar lugar a graves errores en el tratamiento si un médico no tiene acceso al historial médico completo del paciente. Por ejemplo, podrían pasarse por alto alergias o interacciones entre medicamentos, poniendo en riesgo la salud del paciente.

El uso de identificadores únicos, la aplicación de procesos de desduplicación y la implantación de controles estrictos a la hora de crear nuevos registros pueden mejorar la singularidad de los datos.

2. Depuración de datos

2.1 Primeros pasos con la biblioteca pandas

pandas es una potente y versátil biblioteca de Python diseñada para la manipulación y el análisis de datos. Fue desarrollada por Wes McKinney, un investigador que empezó a construir lo que se convertiría en pandas. El nombre «pandas» deriva de «Panel Data», un término econométrico para conjuntos de datos que incluyen observaciones a lo largo de varios periodos.

pandas es particularmente adecuada para trabajar con datos tabulares, similares a una hoja de cálculo de Excel o una tabla SQL. Las principales estructuras de datos gestionadas por esta biblioteca son series, que almacenan datos a lo largo de una dimensión, y DataFrames, que lo hacen a lo largo de dos dimensiones (filas y columnas). Estas estructuras de datos facilitan su manipulación, así como su limpieza, preprocesamiento, análisis y visualización.

pandas es muy utilizada en el análisis de datos. A menudo se presenta como la herramienta ideal para manipular datos que pueden organizarse en filas y columnas. Es más, dominar pandas es una habilidad muy buscada por los reclutadores, ya que muchas empresas de todos los sectores recurren cada vez más a la ciencia de datos.

Existen varias alternativas a pandas, entre ellas polars, dask y cudf. Cada una de estas soluciones tiene sus ventajas, en particular la velocidad de procesamiento en comparación con pandas. No las trataremos en este libro, ya que pandas sigue siendo la librería más utilizada para el análisis de datos en Python. Su riqueza y versatilidad, así como su amplia adopción en la comunidad de análisis de datos, la convierten en una herramienta esencial.

2.2 Presentación de nuestro conjunto de datos

En este capítulo, trabajaremos con un conjunto de datos que está disponible gratuitamente en la plataforma Kaggle.

El comercio electrónico se ha convertido en un nuevo canal de apoyo al desarrollo empresarial. A través del comercio electrónico, las empresas pueden acceder a un mercado más amplio y establecer una mayor presencia proporcionando canales de distribución más baratos y eficientes para sus productos o servicios. El comercio electrónico también ha cambiado la forma en que las personas compran y consumen productos y servicios. Muchas personas recurren a sus ordenadores o dispositivos inteligentes para encargar productos, que pueden recibir fácilmente en sus domicilios.

Se trata de un conjunto de datos de un año de transacciones de ventas en el Reino Unido para el comercio electrónico (venta minorista en línea). Esta tienda londinense vende regalos y artículos para el hogar para adultos y niños en su sitio web desde 2007. Sus clientes proceden de todo el mundo y suelen hacer compras directas para ellos mismos. También hay pequeñas empresas que compran al por mayor y venden a otros clientes a través de canales minoristas.