Antes que nada primero veamos un resumen de lo que es el chequeo de tipos y los diferentes tipados que existen y diferentes perspectivas de estos

##### Chequeo de tipos ( chequeo de tipificación )

El proceso de verificar e imponer los límites impuestos por los tipos de datos –comprobación (chequeo) de tipificación– puede ocurrir tanto en la compilación (una comprobación estática) o en la ejecución (una comprobación dinámica). Si un lenguaje impone fuertemente las reglas de tipificación (es decir, generalmente permitiendo solo las conversiones de tipo de dato automáticas que no hagan perder información), uno se puede referir al proceso como fuertemente tipado; si no, débilmente tipado.

##### Tipado estático

Se dice de un lenguaje de programación que usa un tipado estático cuando la comprobación de tipificación se realiza durante la compilación, y no durante la ejecución. Ejemplos de lenguajes que usan tipado estático son C, C++, Java y Haskell. Comparado con el tipado dinámico, el estático permite que los errores de tipificación sean detectados antes, y que la ejecución del programa sea más eficiente y segura.

##### Tipado dinámico

Se dice de un lenguaje de programación que usa un tipado dinámico cuando la comprobación de tipificación se realiza durante su ejecución en vez de durante la compilación(si es que compilan). Ejemplos de lenguajes que usan tipado dinámico son Perl, Python y Lisp. Comparado con el tipado estático, o sistema de enlazado temprano, el tipado dinámico es más flexible (debido a las limitaciones teóricas de la decidibilidad de ciertos problemas de análisis de programas estáticos, que impiden el mismo nivel de flexibilidad que se consigue con el tipado dinámico ), a pesar de ejecutarse más lentamente y ser más propensos a contener errores de programación.

##### Tipado estático y dinámico combinados

Algunos lenguajes estáticamente tipados tienen una "puerta trasera" en el lenguaje que permite a los programadores escribir código que no es comprobado estáticamente. Por ejemplo, los lenguajes como Java y los parecidos al C tienen una "conversión de tipos de datos forzada (cast)"; estas operaciones pueden ser inseguras durante la ejecución, porque pueden causar comportamientos indeseados cuando el programa se ejecuta. Otros lenguajes de programación como C# utilizan declaraciones de tipos de datos dinámicos en un ambiente estático, permitiendo la flexibilidad casi completa de un lenguaje dinámico en un lenguaje estático.

La presencia de un tipado estático en un lenguaje de programación no implica necesariamente la ausencia de mecanismos de tipado dinámico. Por ejemplo, Java usa tipado estático, pero ciertas operaciones requieren el soporte de test de tipos de datos durante la ejecución, que es una forma de tipado dinámico.

##### **Chequeo de tipificación estático y dinámico en la práctica**

La elección entre sistemas de tipificación dinámico y estático requiere algunas contraprestaciones.

El tipado estático busca errores en los tipos de datos durante la compilación. Esto debería incrementar la fiabilidad de los programas procesados. Sin embargo, los programadores, normalmente, están en desacuerdo en cómo los errores de tipos de datos más comunes ocurren, y en qué proporción de estos errores que se han escrito podrían haberse cazado con un tipado estático. El tipado estático aboga por la creencia de que los programas son más fiables cuando son chequeados sus tipos de datos, mientras que el tipado dinámico apunta al código distribuido que se ha probado que es fiable y con un conjunto pequeño de errores. El valor del tipado estático, entonces, se incrementa a la par que se endurece el sistema de tipificación. Los defensores de los lenguajes fuertemente tipados como ML y Haskell han sugerido que casi todos los errores pueden ser considerados errores de los tipos de datos, si los tipos de datos usados en un programa están suficientemente bien declarados por el programador o inferidos por el compilador.

El tipado estático resulta, normalmente, en un código compilado que se ejecuta más rápidamente. Cuando el compilador conoce los tipos de datos exactos que están en uso, puede producir código máquina optimizado. Además, los compiladores en los lenguajes de tipado estático pueden encontrar atajos más fácilmente. Algunos lenguajes de tipificación dinámica como el Lisp permiten declaraciones de tipos de datos opcionales para la optimización por esta misma razón. El tipado estático generaliza este uso.

En contraste, el tipado dinámico permite a los compiladores e intérpretes ejecutarse más rápidamente, debido a que los cambios en el código fuente en los lenguajes dinámicamente tipados puede resultar en menores comprobaciones y menos código que revisar. Esto también reduce el ciclo editar-compilar-comprobar-depurar.

Lenguajes estáticamente tipados que no dispongan de inferencia (como Java), requieren que el programador declare los tipos de datos que un método o función puede procesar. Esto puede servir a veces de documentación adicional del programa, que es activo y dinámico,en lugar de estático. Esto permite al compilador prevenir salirse de sincronía, y de ser ignorado por el programador. Sin embargo, un lenguaje puede ser de tipificación estática sin requerir la declaración del tipo de dato (ejemplos incluyen Scala y C#3.0), así que esto no es una consecuencia de tipificación estática.

El tipado dinámico permite construcciones que algunos sistemas de tipado estático rechazarían al considerarlas ilegales. Por ejemplo, la función eval de Python, la cual ejecuta datos arbitrarios como si fueran código. Además, el tipado dinámico es más adecuado para código de transición o para el prototipado. Desarrollos recientes en lenguajes como Haskell (por ejemplo, los tipos algebraicos generalizados), permiten a lenguajes de tipado estático, ejecutar código a partir de estructuras de datos de una forma segura.

El tipado dinámico típicamente hace que la metaprogramación sea menos verbosa. Por ejemplo, los genéricos de C++ requiere una escritura más descriptiva que su equivalente en Ruby o Python, precisamente porque es necesario especificar los tipos implicados.

### Ejercicio 2

##### Valore la utilidad de este decorador. ¿Sirve para algo hacer chequeo de tipos en un lenguaje dinámico?

Este decorador **typeCheck**, como ya vimos al resolver el ejercicio anterior, simplemente es un **wrapper** de una función, el cual, durante tiempo de ejecución, verifica los tipos de los parámetros de la misma. Dicho decorador a pesar de no reportar mejoras al rendimiento y optimización del código resultante en tiempo de ejecución, si vuelve mas robusto y seguro de usar el lenguaje garantizando un mejor **feedback** de errores en los tipos de datos utilizados, obviamente reportando dichos errores en tiempo de ejecución. También se puede utilizar como medio de seguridad para crear módulos para usos de terceros que necesiten de una entrada específica

Ahora en respuesta a la pregunta de si sirve o no hacer chequeo de tipos en un lenguaje dinámico pues queremos recalcar que si un lenguaje es dinámico es porque se diseñó así, ya que por el contexto en que se desarrolla, es usado y se creó, para resolver determinados problemas que los lenguajes de tipado estático no podían resolver o que dicha solución era muy engorrosa o difícil de realizar.

Hay muchas partes negativas de usar chequeo de tipos en un lenguaje dinámico ya que complejizan el lenguaje agregándole el tener que revisar los tipos en tiempo de ejecución además de ejecutar el código (en caso de no ser lenguajes compilados) lo cual ralentiza su ejecución. Además este chequeo implica un tema de importantes repercusiones en dicho lenguaje como lo es la modificación de las varianzas del mismo, ya que al tener chequeo de tipos todo el dinamismo que aportaba estos lenguajes se ve forzado a seguir reglas de tipado durante la herencia entre clases.

Pero no todo es tan malo… garantizar este chequeo de tipos vuelve mas seguro y robusto el lenguaje permitiendo a los programadores tener una mayor confianza en el código y en la captura de errores correspondientes a problemas en el uso de los tipos de datos