Seminario 14

* **Patrón Decorator.**
* **Decoradores en Python.**

# Equipo 5:

* Luis Ernesto Ibarra
* Luis Enrique Dalmau
* Laura Tamayo Blanco
* Alberto Helquera
* Jessy Gigato

¿Qué es el patrón Decorador (Decorator Pattern)?

**Patrón de diseño Decorator**

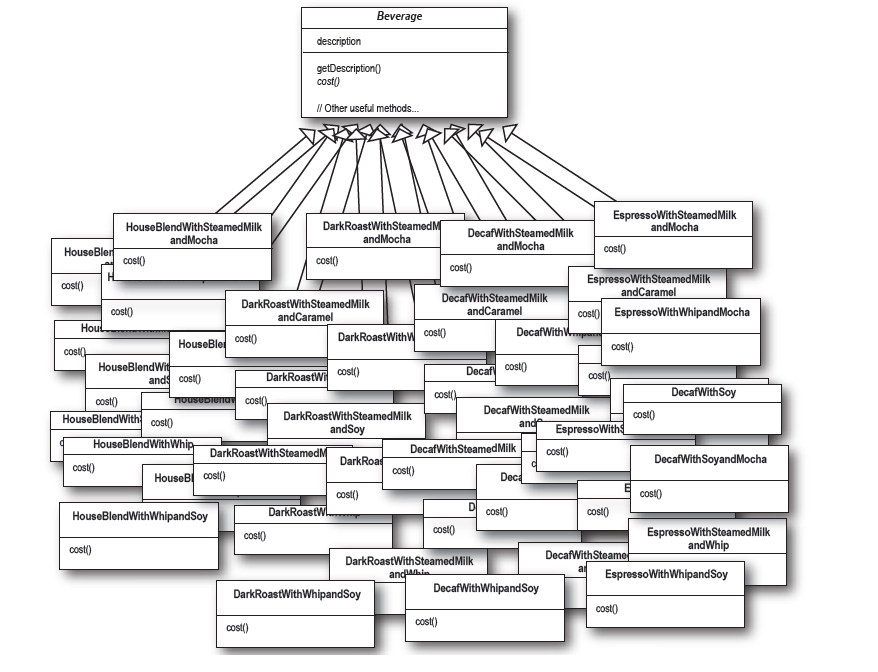
Un patrón de diseño es una descripción de la relaciones entre objetos y clases que son personalizadas con el objetivo de resolver un problema de diseño general en un contexto particular. Se encarga de identificar las clases e instancias que participan, sus roles y colaboraciones, y la distribución de sus responsabilidades.

Según Christopher Alexander( reconocido arquitecto que desarrolló muchos de los conceptos de los patrones que trabajamos) … **“cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y luego describe la esencia de la solución a dicho problema, en una forma que se pueda utilizar la solución un millón de veces más, sin tener que repetir la misma vía dos veces”**.

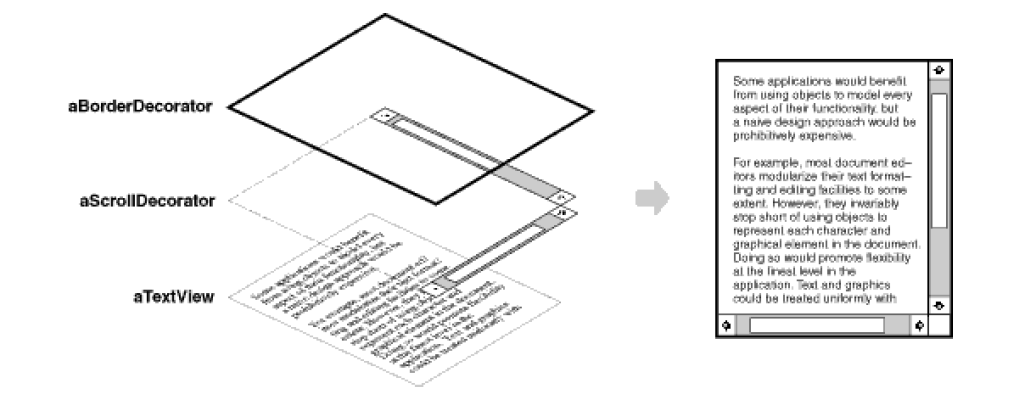
En general, se puede afirmar que los patrones están orientados a facilitar la reutilización de diseños y arquitecturas de software que ya han tenido un éxito confirmado. El patrón de diseño Decorador, también conocido como wrapper, es uno de los patrones más utilizados. Tiene como propósito asignar nuevas responsabilidades a un objeto dinámicamente, independientemente de otras instancias de su misma clase. Es una alternativa flexible a la creación de subclases por herencia para añadir funcionalidades.

Es utilizado con el objetivo de eliminar la necesidad de una cantidad excesiva de subclases. Al mencionar cantidad excesiva de subclases nos referimos a modelos complejos donde es necesario representar todos los posibles comportamientos de un objeto, y dichos comportamientos son difíciles de predecir o es exagerado pretender expresarlos todos.(Fig 1.1)

El objeto decorador funciona como una envoltura del objeto original. Su presencia es transparente al cliente que lo utiliza pues todas las solicitudes sobre él son transferidas al objeto original. Los comportamientos adicionales pueden ser añadidos antes o después de haber transferido una solicitud.

*Figura 1.1: Representación de la clase beverage (bebida) con todas sus posibles combinaciones de agregos.*

El patrón Decorador es utilizado usualmente en aplicaciones GUI (Graphical User Inteface) donde se decoran botones, cajas de diálogo y otros con bordes adicionales u otros componentes GUI. (Ver figura 1.2)

*Figura 1.2: Ejemplo de uso del patrón de diseño Decorador*

*Entre los efectos positivos de su uso se encuentran su mayor flexibilidad en comparación con la herencia estática, pues permite adherir y eliminar responsabilidades y comportamientos de forma sencilla en tiempo de ejecución, mientras que la herencia requiere una nueva clase para cada responsabilidad adicional.*

*Además, facilita en gran medida la adición de una propiedad más de una vez a un mismo objeto (por ejemplo, añadir dos bordes a un botón) y evita que las clases más altas en la jerarquía estén demasiado cargadas de funcionalidad y sean complejas. Por otro lado, un diseño que haga uso de este patrón puede resultar en un sistema compuesto por una gran cantidad de objetos similares, cuya única diferencia radicará en la forma en que ellos están interconectados y no en los valores de sus variables.*

*A pesar que estos sistemas son fáciles de personalizar para alguien que los entiende, pueden resultar difíciles de aprender y depurar. Es válido señalar que se debe ser cuidadoso a la hora de referirse hacia alguno de los componentes decorados, pues un componente original no es igual a un componente decorado, y esto puede generar ciertos conflictos de tipado.*

*En conclusión, en el paradigma de programación orientada a objetos se busca modelar los objetos de la vida real y Decorador es un patrón muy versátil para este propósito, de ahí su popularidad.*

Aplicabilidad:

* Añadir responsabilidades a objetos individuales de forma dinámica y transparente
* Responsabilidades de un objeto pueden ser retiradas
* Hay una necesidad de extender la funcionalidad de una clase, pero no hay razones para extenderlo a través de la herencia.
* Existe la necesidad de extender dinámicamente la funcionalidad de un objeto y quizás quitar la funcionalidad extendida.

¿Qué es un decorador en Python?

Un decorador en Python es una función que toma como primer parámetro una función o una clase y devuelve una función o una clase respectivamente.

Sintaxis:

@Expr\_1

@Expr\_2

…

@Expr\_n

definition # function or class definition

Todas las expresiones Expr\_i deben tienen que recibir el parámetro de la función o clase a decorar.

La sintaxis anterior solamente es azúcar sintáctica para:

definition = Expr\_1(Expr\_2(…(Expr\_n(definition))…)

Lo anterior es esencial para comprender como funcionan los decoradores.

Los decoradores en Python se dividen en dos familias, están los decoradores de funciones y los decoradores de clases. Estos, sin importar el tipo que sean, se pueden construir utilizando funciones y/o clases.

Ejemplos:

Function Decorators con Funciones (sin parámetros)

def decorator(func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return value

return inner\_func

@decorator

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se pudo observar cómo definir un decorador simple. En el llamado de la función decorada el código ejecutado es el de inner\_func en vez del target\_func. Como se puede ver inner\_func es un wrapper de la función a decorar.

Function Decorators con Funciones (con parámetros)

def param\_decorator(param0,param1,param2): # Params Function

def wrap\_func(func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return value

return inner\_func

return wrap\_func

@param\_decorator(1,2,3)

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se pudo observar cómo definir un decorador que tome argumentos. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = param\_decorator(1,2,3)(target\_function)

Así se puede apreciar que la función devuelta en la primera llamada tomará como argumento target\_function retornando a inner\_func como target\_function. Pudiendo utilizar dentro de inner\_func los parámetros capturados en el llamado del decorador.

Function Decorators con Clases (sin parámetros)

class FunctionDecoratorClass:

def \_\_init\_\_(self,func): # Wrap Function

self.func = func

def \_\_call\_\_(self, \*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = self.func(\*args,\*\*kwargs)

# Something to do after the func call

@FunctionDecoratorClass

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se pudo observar cómo definir un decorador con una clase. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = Decorator(target\_function)

Así se puede apreciar que la instancia de la clase devuelta se almacena en target\_function, y cada vez que se llame a esta función (instancia de clase), llamará a \_\_call\_\_, haciendo parecer que se está llamando a una función.

Function Decorators con Funciones (con parámetros)

class ArgumentFunctionDecoratorClass:

def \_\_init\_\_(self,\*args): # Params Function

self.args = args

def \_\_call\_\_(self, func): # Wrap Function

def inner\_func(\*args,\*\*kwargs): # Inner Function

# Something to do before the func call

value = func(\*args,\*\*kwargs)

# Something to do after the func call

return inner\_func

@ArgumentFunctionDecoratorClass(1,2,3,4)

def target\_function(\*args,\*\*kwargs):

# Function body

pass

target\_function()

En este ejemplo se pudo observar cómo definir un decorador con una clase que tome argumentos. Veamos a que se traduce esto para ver con más claridad lo que sucede:

target\_function = ArgumentDecorator(1,2,3)(target\_function)

Así se puede apreciar que la instancia de la clase devuelta en la primera llamada, llamará a \_\_call\_\_ tomando como argumento target\_function retornando a inner\_func como target\_function.

Class Decorators con Funciones (sin parámetros)

El método \_\_getattribute\_\_ es llamado cada vez que se quiere coger un atributo de la clase. Al sobreescribirlo de la forma en que se muestra forma un enlace entre la clase decoradora y la clase decorada.

def class\_decorator(Cls): # Wrap Function

class InnerClass(Cls):

# Additional Attributes

def \_\_init\_\_(self, \*args, \*\*kwargs):

print('Creating Decorated Instance')

super().\_\_init\_\_(\*args,\*\*kwargs)

self.attr = 'Value changed in Decorated Class'

# Adding extra content

def \_\_getattribute\_\_(self,name):

# Do something before get the attributes

print('Getting attribute',name,'in InnerClass')

x = super().\_\_getattribute\_\_(name)

print('Attribute',name,'receive in InnerClass')

# Do something after get the attributes

return x

return InnerClass

@class\_decorator

class DummyClass: …  
a = DummyClass()

a.test()

En este ejemplo se pudo observar cómo definir un decorador de clases con una funci’on. Veamos a que se traduce esto para ver con más claridad lo que sucede:

a = class\_decorator(DummyClass)()

Se puede observar que a es una instancia de InnerClass que intermedia con la clase DummyClass

Class Decorators con Funciones (con parámetros)

def class\_param\_decorator(\*pargs): # Params Function

def wrap\_func(Cls): # Wrap Function

class InnerClass(Cls):

# Additional Attributes

def \_\_init\_\_(self, \*args, \*\*kwargs):

print('Creating Decorated Instance')

super().\_\_init\_\_(\*args,\*\*kwargs)

self.attr = pargs

# Adding extra content

def \_\_getattribute\_\_(self,name):

# Do something before get the attributes

print('Getting attribute',name,'in InnerClass')

x = super().\_\_getattribute\_\_(name)

print('Attribute',name,'receive in InnerClass')

# Do something after get the attributes

return x

return InnerClass

return wrap\_func  
@class\_param\_decorator(1,2,3)

class DummyClass: …  
a = DummyClass()

a.test()

En este ejemplo se pudo observar cómo definir un decorador de clases con parámetros con una función. Veamos a que se traduce esto para ver con más claridad lo que sucede:

a = class\_param\_decorator(1,2,3)(DummyClass)()

Al observar podemos decir que la primera llamada devuelve wrap\_function, luego esta captura la clase a decorar y devuelve la InnerClass que luego es instanciada.

Class Decorators con Clases (sin parámetros)

class ClassDecoratorClass:

def \_\_init\_\_(self,Cls): # Wrap Function

class InnerClass(Cls):

# Additional Attributes

def \_\_init\_\_(self, \*args, \*\*kwargs):

print('Creating Decorated Instance')

super().\_\_init\_\_(\*args,\*\*kwargs)

self.attr = 'Value changed in Decorated Class'

# Adding extra content

def \_\_getattribute\_\_(self,name):

# Do something before get the attributes

print('Getting attribute',name,'in InnerClass')

x = super().\_\_getattribute\_\_(name)

print('Attribute',name,'receive in InnerClass')

# Do something after get the attributes

return x

self.Cls = InnerClass

def \_\_call\_\_(self, \*args,\*\*kwargs): # Inner Function

# Something to do before instance a class

Cls = self.Cls(\*args,\*\*kwargs)

# Something to do after instance a class

return Cls

@ClassDecoratorClass

class DummyClass: …  
a = DummyClass()

a.test()

En este ejemplo se pudo observar cómo definir un decorador de clases con una clase. Veamos a que se traduce esto para ver con más claridad lo que sucede:

a = ClassDecoratorClass(DummyClass)()

Como se pudo observar la clase ClassDecoratorClass se convierte en un creador de instancias de InnerClass, la cual es el wrapper alrededor de la clase a decorar.

Class Decorators con Clases (con parámetros)

class ArgumentClassDecoratorClass:

def \_\_init\_\_(self,\*args,\*\*kwargs): # Params Function

self.args = args

self.kwargs = kwargs

def \_\_call\_\_(self, Cls): # Wrap Function

# Something to do before instance a class

class InnerClass(Cls):

# Additional Attributes

def \_\_init\_\_(self2, \*args, \*\*kwargs):

print('Creating Decorated Instance')

super().\_\_init\_\_(\*args,\*\*kwargs)

self2.attr = self.args

# Adding extra content

def \_\_getattribute\_\_(self,name):

# Do something before get the attributes

print('Getting attribute',name,'in InnerClass')

x = super().\_\_getattribute\_\_(name)

print('Attribute',name,'receive in InnerClass')

# Do something after get the attributes

return x

return InnerClass

@ArgumentClassDecoratorClass(1,2,3,4)

class DummyClass: …  
a = DummyClass()

a.test()

En este ejemplo se pudo observar cómo definir un decorador de clases con parámetros con una clase. Veamos a que se traduce esto para ver con más claridad lo que sucede:

a = ArgumentClassDecoratorClass(1,2,3,4)(DummyClass)()

Como se pudo observar la clase ArgumentClassDecoratorClass se convierte en un creador de instancias de InnerClass, capturando como atributos los argumentos pasados en el decorador.

Análisis general de los ejemplos

Como se pudo apreciar en los ejemplos anteriores, se dividieron las partes que componen un decorador en tres:

* Inner Function: Es la función que se ejecutara cada vez que se llame a la función decorada.
* Wrap Function: Es la función encargada de capturar la función a decorar. Debe de tomar solamente un argumento, el de la función, y de devolver una Inner Function.
* Params Functions: Son aquellas que se utilizan con el operador @ las cuales pueden tomar parámetros para personalizar el comportamiento de la función decorada. Viene acompañada de una Wrap Function y una Inner Function.

Aunque esto no es una metodología, es una buena forma de ver los decoradores. En el caso de los decoradores de clase las Inner Function son las InnerClass declaradas.

Python Function Annotations:

Las function annotations de Python, es una manera de añadir metadatos sobre los argumentos y el valor de retorno de las funciones para su posterior interpretación. Estas fueron incluidas a partir de Python 3.0.

Veamos un ejemplo:

def foo (param0,param1:type, param2:'string',param3:max(1,2,3)='default'\

,\*args:{1:'1',2:'2'},\*\*kwargs:[1,2,3,4])->str:

print(foo.\_\_annotations\_\_)

foo(1,2,3)

En el ejemplo anterior se puede observar que cualquier expresión puede estar en las annotations, lo que nos lleva a sacar la conclusión de que estas son solamente datos y que su uso varía según la interpretación que se le dé. Por ejemplo en VS Code si se anota el tipo de retorno de una función, luego de llamarla nos aparecerán los posibles métodos y atributos que contiene el tipo anotado, otro uso sería en la documentación de las variables, también en la creación de módulos que simulen la sobrecarga de funciones. Su uso es tan amplio como se quiera.

El acceso a estas es mediante el atributo \_\_annotations\_\_ de las funciones (ver ejemplo anterior), este devuelve un diccionario de nombre\_de\_atributo contra annotation. Veamos que imprime el ejemplo anterior.

{

'param1': <class 'type'>,

'param2': 'string',

'param3': 3,

'args': {1: '1', 2: '2'},

'kwargs': [1, 2, 3, 4],

'return': <class 'str'>

}

Como se puede observar imprime las annotations de los parámetros anotados (no imprime param0 el cual no está anotado) con el resultado de evaluar la expresión (note que en param3 imprime 3 que es el máximo entre 1, 2 y 3). La annotation del retorno de la función viene dada por la llave return, la cual como palabra clave de Python no puede formar parte de una expresión.

Ejercicio 1:

1. Implemente un decorador **typeCheck** que reciba los tipos de los parámetros de la función a decorar y veriﬁque en cada invocación que los objetos pasados como argumentos correspondan con dichos tipos.

En Python las variables y los parámetros de las funciones no tienen tipo estático, pero los objetos dentro de estas sí tienen tipos. Por lo tanto se hizo un decorador que en tiempo de ejecución, verificara que los parámetros anotados (function annotations) de la función se correspondieran con su tipo de entrada. Si se anota el tipo de salida, esta también es verificada.

Como la manera de pasar argumentos a una función en Python es muy flexible primero se hizo una función que ordenara estos de una manera más cómoda para su posterior análisis. La función clean\_func\_input recibe los datos de la función a limpiarle la entrada y la entrada de esa función y devuelve los argumentos en el mismo orden en que se declararon en el código.

def clean\_func\_input(arg\_count,arg\_list,defaults,\*args,\*\*kwargs):

args = list(args)

len\_args = len(args)

# Checking kwargs for double reference

explicit\_args = arg\_list[:min(len\_args,arg\_count)] # Anything beyond the arg\_count belongs to \*args

for name in explicit\_args:

if name in kwargs:

# Raise a TypeError

fn(\*args,\*\*kwargs)

# Inserting the defaults values

default\_offset = len\_args + len(defaults) - arg\_count

for index in range(len\_args,arg\_count):

default\_index = index - len\_args + default\_offset

default\_name = arg\_list[index]

if default\_name in kwargs:

args.insert(arg\_count-1,kwargs[default\_name])

kwargs.\_\_delitem\_\_(default\_name) # Deleting params kwargs from kwargs

else:

args.insert(arg\_count-1, defaults[default\_index])

return args,kwargs # funcion args and kwargs in order of declaration

Una vez explicado la función anterior, el código del decorador quedaría así:

def typeCheck(fn): # Wrap Function

"""

Check if the function 'fn' behaves according its annotations

The annotations must be a 'type' instance

"""

# Gathering func info

types = typing.get\_type\_hints(fn) # types = fn.\_\_annotations\_\_ also work

arg\_count = fn.\_\_code\_\_.co\_argcount

arg\_list = fn.\_\_code\_\_.co\_varnames # tuple with the function params names

defaults = fn.\_\_defaults\_\_ # tuple with function default values

def inner\_typeCheck(\*args,\*\*kwargs): # Inner Function

args,kwargs = clean\_func\_input(arg\_count, \

arg\_list,defaults,\*args,\*\*kwargs)

# Checking arguments types

errors = []

for arg,arg\_type in types.items():

if arg == 'return': continue

index = arg\_list.index(arg)

if isinstance(arg\_type,type) and not \

isinstance(args[index],arg\_type):

errors.append(f"Wrong Type in argument \

'{arg}' of function '{fn.\_\_name\_\_}' \

\nExpected {arg\_type} receive {type(args[index])}")

if not errors:

# Calling the function

return\_value = fn(\*args,\*\*kwargs)

# Checking return type

if 'return' in types:

if isinstance(types['return'],type) and not \

isinstance(return\_value,types['return']):

errors.append(f"Wrong Type in return value of function \

'{fn.\_\_name\_\_}' \nExpected {types['return']}\

receive {type(return\_value)}")

raise TypeError(\*errors)

return return\_value

raise TypeError(\*errors)

return inner\_typeCheck

El decorador anterior en tiempo de ejecución verifica los tipos anotados, lanzando un error en tiempo de ejecución si los tipos no coinciden. Para ver ejemplos de uso diríjase a Code/annotate\_decorator.py.

Se hizo otro decorador cuya funcionalidad es similar al anterior pero esta vez se tiene que especificar en los argumentos del decorador las variables con sus tipos válidos.

def typeCheckParams(\*\*kwargs): # Params Function

errors = []

# Check if all values are a type or an iterable of types

for x,value in kwargs.items():

if not (hasattr(value,'\_\_iter\_\_') or isinstance(value,type)):

errors.append('The vales of the decorator must be an iterable \

of types or a type')

try: # Checking if is iterable (is not enough with

# checking if has \_\_iter\_\_)

for y in value:

if not isinstance(y,type):

errors.append('The vales of the decorator \

must be an iterable of types')

break

except TypeError: # Not an iterable, can be a type instance

if isinstance(value,type):

kwargs[x] = [value,] # Convert the type instance in a

# iterable of one for later code

# simplification

else:

errors.append('The vales of the decorator must be an \

iterable of types')

if errors:

raise TypeError(\*errors)

def wrap(fn): # Wrap Function

# Gathering func info

arg\_count = fn.\_\_code\_\_.co\_argcount

arg\_list = fn.\_\_code\_\_.co\_varnames # tuple with the function

# params names

defaults = fn.\_\_defaults\_\_ # tuple with function default values

def inner(\*args, \*\*i\_kwargs): # Inner Function

args,i\_kwargs = clean\_func\_input(arg\_count,arg\_list \

,defaults,\*args,\*\*i\_kwargs)

# Check Types

for i,arg in enumerate(arg\_list):

if arg in kwargs:

typex = kwargs[arg]

for possible\_type in typex:

if isinstance(args[i],possible\_type):

break

else:

errors.append(f"Wrong Type in argument '{arg}' of \

function '{fn.\_\_name\_\_}' \nExpected \

{typex} receive {type(args[i])}")

if errors:

raise TypeError(\*errors)

value = fn(\*args,\*\*i\_kwargs)

if 'returnx' in kwargs:

typex = kwargs['returnx']

for possible\_type in typex:

if isinstance(value,possible\_type):

break

else:

errors.append(f"Wrong Type in return value of \

function '{fn.\_\_name\_\_}' \nExpected \

{typex} receive {type(value)}")

raise TypeError(\*errors)

return value

return inner

return wrap

El decorador anterior tiene la misma funcionalidad que el primero, pero en este es necesario especificar que parámetro se quiere verificar y cuál o cuáles son los tipos válidos para este. Por la sintaxis de Python para especificar el tipo del retorno de la función es necesario utilizar la llave returnx. Para ver ejemplos de uso diríjase a Code/params\_decorator.py.

Ejercicio 2:

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

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 casado 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++ requieren una escritura más descriptiva que su equivalente en Ruby o Python, precisamente porque es necesario especificar los tipos implicados.

Respuesta:

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 más 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 más 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.

Ejercicio 3:

3. ¿Tendría sentido modificar el lenguaje para tener variables y parámetros con tipos? ¿La definición del lenguaje obliga a la necesidad de variables y parámetros sin tipo?

Como todos sabemos en Python las variables y parámetros son dinámicos por la forma en que se construyó el lenguaje, esto se debe a como en runtime el intérprete se encarga de buscar en los diccionarios de los objetos las definiciones de sus métodos o atributos. Con ello se logra una mayor flexibilidad, libertad y reusabilidad del código creado, pero ocurre también que se produce una cierta inseguridad en el código debido a que no se puede saber hasta ejecutar el código si este es correcto y además se pierde en rendimiento debido al preprocesamiento que debe hacer el intérprete para poder ejecutar el programa.

Por ello muchos se preguntarán si tendría sentido modificar el lenguaje para tener variables y parámetros con tipo, pero antes de responder a esta interrogante ojeemos primero lo que dice el Zen de Python:

* Bello es mejor que feo.
* Explícito es mejor que implícito.
* Simple es mejor que complejo.
* Complejo es mejor que complicado.
* Plano es mejor que anidado.
* Disperso es mejor que denso.
* La legibilidad cuenta.
* Los casos especiales no son tan especiales como para quebrantar las reglas.
* Lo práctico gana a lo puro.
* Los errores nunca deberían dejarse pasar silenciosamente.
* A menos que hayan sido silenciados explícitamente.
* Frente a la ambigüedad, rechaza la tentación de adivinar.
* Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
* Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
* Ahora es mejor que nunca.
* Aunque nunca es a menudo mejor que ya mismo.
* Si la implementación es difícil de explicar, es una mala idea.
* Si la implementación es fácil de explicar, puede que sea una buena idea.
* Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Tim Peters, El Zen de Python

Se aprecia que algunos de los puntos que más se resaltan son la simplicidad, practicidad, facilidad de explicar y belleza del código, esto nos hace pensar que el dinamismo de tipado para los creadores de Python es considerado una muy buena idea y hace que nos cuestionemos de si realmente el tipado tiene lugar en un lenguaje así.

Veamos algunos de los puntos fuertes del tipado estático:

* Detección temprana de errores de programación (normalmente en tiempo de compilación).
* Un mayor rendimiento (por la posibilidad de compilador de hacer mayor optimización al código).
* Posibilidad de implementar mecanismos como el Intellisense.
* Mayor robustez del código.
* Facilita la documentación del código escrito.

En el caso de agregar tipado en Python, al no haber proceso de compilación, no se puede lograr el primer punto, ni el segundo. El tercero no tendría mucho sentido ya que como el lenguaje es predominantemente dinámicamente tipado este no funcionaría en la mayoría de los casos por lo que ciertamente se pierden varias de las ventajas que viene con el tipado estático al usar tipos en Python.

Pero, aun así, es cierto que escribir código en Python con tipos le ofrecería una mayor robustez al código escrito, facilitaría la documentación del mismo y haría más fácil el trabajo con las IDEs, por lo cual en grandes módulos en que sea necesario testear el código este podría ser bastante útil.

Por tanto, consideramos que el uso de tipos en Python tendría sentido; aunque siempre intentando no abusar de él y perder las ventajas que tiene el propio lenguaje, así como la ideología con que este fue concebido.