



Python Intermedio

Profesor: Cañizares Diego

Profesor: Mostovoi Alejandro

Año: 2021

Indice

Introduction	9
¿Por qué elegir Python?	10
Tipos de datos nativos	12
Numéricos	12
Enteros	12
Reales	14
Lógicos	15
Cadenas (Strings)	15
Accediendo una posición específica de una cadena:	16
Modificando una cadena	17
Recorriendo una cadena	17
Operadores	18
Longitud	18
Convertir a minúsculas/mayúscula (Case Converting)	18
Subcadena (slice)	19
Concatenar	20
Reemplazar texto	20
Separar (split)	20
Unir (join)	21
Verificando si una cadena existe en otra cadena	21
Formateando cadenas	21
Caracteres de escape	21
Raw (pura) Strings	22
String formatting	22
Controlando el flujo del programa	23
Estructuras de control	23
If	23
Operadores relacionales	24
If...else	25
elif	25
Estructuras de repetición	26
While	26
For	27

Función enumerate	28
Unpacking estructuras en un ciclo for	28
Funciones	29
Definición	29
Invocando funciones	30
Argumentos	31
Posicionales	31
Nombrados (Keyword arguments)	31
Cantidad de parámetros variables (*args y **kwargs)	32
*args	32
**kwargs	33
Unidad 2	34
Diseño Orientado a Objetos	34
Clases y Objetos	35
Introducción	35
Definición de una clase	35
El método constructor	36
Atributos	37
Atributos de instancia	37
Ámbito de los atributos de instancia	38
Atributos de clase	40
Métodos	42
Métodos de instancia	42
Métodos de clase	43
Métodos estáticos	44
Métodos __eq__, __str__ y __hash__	44
Obteniendo la representación de un objeto (__str__)	44
Comparando objetos (__eq__ y __hash__)	46
Método __eq__	46
Método __hash__	48
Herencia	49
Agregando propiedades	50
Modificando el comportamiento de las clases derivadas	51
La función super()	52
Clase abstracta	52
Herencia Múltiple	53

Polimorfismo	54
Excepciones	56
Manejando excepciones	56
Cláusula try...except...finally	58
Cláusula try...except...else...finally	59
Excepciones definidas por el usuario	60
Unidad 3	62
Patrones de Diseño	62
Patrones de diseño	63
Introducción	63
Patrones creacionales	63
Abstract Factory	63
Propósito	63
Problema	63
Solución	64
Estructura	66
Aplicabilidad	66
Cómo implementarlo	67
Pros y contras	67
Builder	68
Propósito	68
Problema	68
Solución	68
Clase directora	69
Estructura	70
Aplicabilidad	71
Cómo implementarlo	72
Pros y contras	72
Singleton	73
Propósito	74
Problema	74
Solución	74
Analogía en el mundo real	75
Estructura	75
Aplicabilidad	76
Cómo implementarlo	76

Pros y contras	76
Patrones estructurales	78
Adapter	78
Propósito	78
Problema	78
Solución	78
Analogía en el mundo real	79
Estructura	79
Clase adaptadora	80
Aplicabilidad	80
Cómo implementarlo	81
Pros y contras	81
Decorator	82
Propósito	82
Problema	82
Solución	83
Analogía en el mundo real	86
Estructura	86
Aplicabilidad	87
Cómo implementarlo	88
Pros y contras	88
Composite	89
Propósito	89
Problema	89
Solución	89
Analogía en el mundo real	90
Estructura	90
Aplicabilidad	91
Cómo implementarlo	91
Pros y contras	92
Patrones de comportamiento	93
State	93
Propósito	93
Problema	93
Solución	94
Analogía en el mundo real	95
Estructura	96

Aplicabilidad	97
Cómo implementarlo	97
Pros y contras	98
Strategy	98
Propósito	98
Problema	98
Solución	99
Analogía en el mundo real	100
Estructura	100
Aplicabilidad	101
Cómo implementarlo	102
Pros y contras	102
Template Method	103
Propósito	103
Problema	103
Solución	104
Analogía en el mundo real	105
Estructura	105
Aplicabilidad	105
Cómo implementarlo	106
Pros y contra	107
Unidad 4	107
Generators	108
Definición	109
Propósito	109
Ejemplos de uso	109
Leyendo archivos	109
Generando secuencias infinitas	110
La instrucción yield	111
Creando generators a través de expresiones	112
Generators y Pipelines	112
Unidad 5	113
Threads	114
Introducción	115
Definición	115
Diferencias entre hilo y proceso	116

Aplicaciones con un único hilo (proceso)	116
Aplicaciones multi-hilo (multi-threading)	118
Un caso especial - daemons threads	120
Trabajando con múltiples hilos	120
Concurrencia	121
Condición de carrera	121
Sincronización de hilos	124
Deadlock	125
Unidad 6	126
Coroutines	127
Introducción	128
Definición	128
Usando yield como expresión	128
Send y close	129
Pipelines	130
Bonus track	133
Broadcasting	134
Unidad 7	134
Administradores de dependencias y Entornos virtuales	135
Entornos Virtuales	136
Definición	136
Beneficios	136
Usando entornos virtuales	136
Activar un entorno virtual	138
Desactivar un entorno virtual	138
Administradores de dependencias	139
pip	139
pipenv	139
poetry	141
Manejando dependencias	142
Generando un paquete	142

—



Unidad 0

Introducción

Introduction

Una breve historia de Python

Python fue desarrollado a finales de la década de 1980¹. Todos coinciden en que su creador es Guido van Rossum cuando creó *python* como sucesor del lenguaje de programación *ABC* que estaba usando. El nombre del lenguaje, se cree, fue tomado de una de las películas de comedia del grupo: "Monty Python"². El lenguaje no se publicó hasta 1991 y ha crecido mucho desde entonces, en términos de la cantidad de módulos y paquetes incluidos.

Durante años, la versión estable de python fue la 2.x, hasta diciembre de 2019 donde fue oficialmente deprecada. Actualmente, la versión 3.x es la versión estable. La misma no es compatible con versiones anteriores de 2.x.

Algunas personas piensan que Python es para escribir pequeños scripts que sólo sirven para unir código "real", como el que puede ser escrito con lenguajes como C++, Java o Haskell. Sin embargo, veremos que Python es útil en casi cualquier situación. Actualmente Python es utilizado en proyectos de muchas empresas de renombre como Google, NASA, LinkedIn, etc.

Python se usa no sólo en el backend, sino también en el frontend. En caso de que estos términos le resulten nuevos, podemos referirnos a la programación de backend como lo que se encuentra detrás de escena; cosas como procesamiento de bases de datos, generación de documentos, etc. El desarrollo del frontend se refiere a la construcción de interfaces de usuario, ya sea gráficas o de texto, podríamos decir que es la cara visible de la aplicación.

Por último cabe destacar que actualmente encontramos muchas herramientas o frameworks tanto para desarrollo de interfaces gráficas de escritorio como para desarrollo de aplicaciones web. Entre las más conocidas podemos citar:

- Escritorio
 - wxPython
 - tKinter
 - PyQt
- Web
 - Django
 - Flask
 - Web2py

¹<http://www.artima.com/intv/pythonP.html>

² https://en.wikipedia.org/wiki/Monty_Python



¿Por qué elegir Python?

Estas son algunas de las características que hacen que Python sea una opción a considerar a la hora de elegir un lenguaje de programación para desarrollar nuestra aplicación.

Popular: de acuerdo con la encuesta de desarrolladores de [Stack Overflow](#), Python es la cuarta tecnología más popular y la número uno más buscada. (consulta realizada en 2021)

Simple: Python tiene una sintaxis simple similar al idioma inglés.

Gratuito: Python está desarrollado bajo una licencia de código abierto, lo que lo hace libre de instalar, usar y distribuir, incluso con fines comerciales.

Independiente de la plataforma: Python funciona en diferentes plataformas (Windows, Mac, Linux, etc.).

Interpretado: lo que significa que el código se puede ejecutar tan pronto como se escribe. Esto significa ciclos de desarrollo más rápidos.



Unidad 1

Nivelación

Page 10 of 10

Numéricos

Python permite trabajar con 3 tipos de valores numéricos:

- Enteros
- Reales
- Complejos

Enteros

El conjunto de números enteros, en Python, abarca un rango muy amplio de valores. No hay modificadores como “entero largo” (long int), “entero corto” (short int) como sí los hay en otros lenguajes como C/C++. En Python 3, no hay límite para la longitud de un valor entero. Puede tener tantos dígitos como lo permita el espacio de direcciones de memoria de la computadora.

Por ejemplo:

```
# Definición de variables enteras  
>>> numero_1 = 10  
>>> numero_negativo = -20  
>>> entero_muy_grande = 9999999999999999999999999999999999999999999999999999999
```

Python también nos permite trabajar con números enteros en otras bases, como puede ser binario, octal o hexadecimal. Para ello debemos especificar al momento de definir el número en qué base se encuentra:

Prefijo	Base
‘0b’ ó ‘0B’	2 (binario)
‘0o’ ó ‘0O’	8 (Octal)
‘0x’ ó ‘0X’	16 (Hexadecimal)

También podemos usar las siguientes funciones de conversión entre los sistemas mencionados:

Decimal ↔ Binario

- `bin`: devuelve la representación binaria de un número decimal
- `int` : devuelve la representación decimal de un número binario

Ejemplo:

```
>>> bin(10)
'0b1010'

>>> int(0b10)
2
```

Decimal ↔ Octal

- `oct`: devuelve la representación octal de un número decimal
- `int` : devuelve la representación decimal de un número octal

Ejemplo:

```
>>> oct(10)
'0o12'

>>> int(0o10)
8
```

Decimal ↔ Hexadecimal

- `hex`: devuelve la representación hexadecimal de un número decimal
- `int` : devuelve la representación decimal de un número hexadecimal

Ejemplo:

```
>>> hex(10)
'0xa'

>>> int(0xFF)
```

```
255
```

Reales

Son números, positivos o negativos, que poseen parte decimal

Por ejemplo:

```
# Definición de variables reales
>>> numero_1 = 10.5
>>> numero_negativo = -20.03
>>> Otro_eal = 1.23456
```

Python también nos brinda la posibilidad de expresar números en notación científica

```
# reales en notación científica
>>> numero_1 = 1e2
>>> numero_1
100.0
```

Los valores máximos y mínimos para números reales son (aproximadamente):

- Mínimo: 5.0×10^{-324} . Cualquier valor inferior se considera 0 (cero)
- Máximo: 1.8×10^{308} . Cualquier valor superior se indica con la cadena “inf”

Operadores aritméticos

Operador	Operación	Ejemplo
**	Exponente	$2 ** 3 = 8$
%	Módulo (devuelve el resto de una división)	$22 \% 8 = 6$

//	División entera	22 // 8 = 2
/	División	22 / 8 = 2.75
*	Multiplicación	3 * 3 = 9
-	Resta	5 - 2 = 3
+	Suma	2 + 2 = 4

La tabla anterior, ordena los operadores según su precedencia de mayor a menor

Lógicos

El tipo de dato lógico o **bool**, sólo admite 2 valores **True** o **False**. Cabe destacar que el tipo de dato **bool** en Python, es una subclase del tipo de dato **int**.

Operadores que se pueden utilizar con el tipo de dato bool

Operador	Acción
AND	conjunción
OR	disyunción
NOT	negación

Cadenas (Strings)

Una cadena o “string”, es una secuencia de cero o más caracteres encerrada entre comillas simples o dobles. A diferencia de lenguajes como C, en Python, las cadenas vienen con un poderoso conjunto de funciones de procesamiento.

Ejemplo:

```
# Creando cadenas
nombre = "Juan"
```



```
# En varias líneas
oracion = '''Esta es una cadena
en varias
líneas.
'''

# Usando el constructor str
cadena = str(45)
```

El constructor `str` permite convertir (casi) cualquier objeto a string.

Accediendo una posición específica de una cadena:

Una cadena, por ser una secuencia de caracteres, puede ser accedida en una posición específica. Para ello debemos indicar la posición a la cuál queremos acceder . Recordar que las cadenas al igual que muchos objetos se indexan a partir de 0 (cero).

Cabe destacar que en Python, las cadenas pueden ser accedidas con índices negativos

0	1	2	3
H	O	L	A
-4	-3	-2	-1

Los números sobre la palabra indican los índices de cada caracter, mientras que los números debajo de la palabra indican los índices negativos de la misma.

Ejemplo:

```
>>> cadena = 'abcde'
>>> cadena[0]
'A'
>>> cadena[2]
'c'
>>> cadena[-1]
'e'
```

Modificando una cadena

Las cadenas en Python son objetos inmutables. Esto quiere decir que no pueden ser modificadas luego de ser creadas.

Ejemplo:

```
>>> cadena = 'abcde'
>>> cadena[2] = '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

En el ejemplo anterior podemos observar el mensaje de error que arroja el intérprete de Python cuando intentamos modificar la tercer posición de la cadena.

En caso de necesitar modificar una cadena, lo que se puede hacer es generar una nueva cadena tomando porciones de la cadena original.

Ejemplo:

```
>>> cadena = 'abcde'
>>> cadena = cadena[0:2] + '1' + cadena[3:]
>>> cadena
'ab1de'
```

Recorriendo una cadena

Para iterar por los caracteres que conforman una cadena, podemos utilizar la estructura de repetición [for...in](#)

Ejemplo:

```
>>> cadena = 'abcde'
>>> for l in cadena:
```

```
... print(l, end= ' ')
```

```
A b c d e
```

Operadores

Longitud

Para encontrar la longitud de una cadena, Python proporciona el operador *len*

Ejemplo:

```
>>> cadena = 'abcde'
>>> len(cadena)
5
```

Convertir a minúsculas/mayúscula (Case Converting)

Python nos brinda 5 métodos para trabajar con la conversión de letras de mayúsculas a minúsculas y viceversa. Estos son:

- lower: convierte toda la cadena a minúsculas
- upper: convierte toda la cadena a mayúscula
- capitalize: convierte la primera letra de la cadena a mayúscula y el resto a minúscula.
- swapcase: invierte minúsculas y mayúsculas
- title: convierte la primer letra de cada palabra en la cadena a mayúscula

Ejemplos:

```
>>> cadena = 'Hola MUNDO!'

# lower
>>> cadena.lower()
```

```
Hola mundo

# upper
>>> cadena.upper()
HOLA MUNDO

# capitalize
cadena.capitalize()
Hola mundo!

# swapcase
cadena.swapcase()
hOLA mundo!

# title
cadena.title()
Hola Mundo!
```

Subcadena (slice)

En Python podemos obtener una porción de una cadena utilizando el operador “[n:m]”, que nos devuelve a partir de la cadena original una subcadena con los caracteres que se encuentran entre la posición [n, m), es decir, incluye el caracter que se encuentra en la primer posición especificada pero excluye el de la última posición especificada.

Ejemplo:

```
>>> cadena = 'Hola MUNDO!'
>>> cadena[0:2]
'Ho'

>>> cadena[3:6]
'a M'

# tomando índices negativos
>>> cadena[3: -2]
```

```
'a MUND'
```

Concatenar

Esta operación permite unir 2 o más cadenas. El operador que permite realizar esta operación es el '+'.
Ejemplo:

```
>>> cadena1 = 'Hola'
>>> cadena2 = 'Mundo!'
>>> cadena1 + ' ' + cadena2
'Hola Mundo!'
```

Reemplazar texto

Si necesitamos reemplazar una parte de una cadena, podemos utilizar el operador [replace](#)

Ejemplo:

```
>>> cadena = 'Hola Mundo!'
>>> cadena.replace('Mundo', 'World')
'Hola World!'
```

Separar (split)

El operador [split](#) nos permite separar una cadena en tokens y generar una lista con los tokens encontrados. Debemos especificar una cadena separadora.

Ejemplo:

```
>>> cadena = 'Hola Mundo!'
>>> cadena.split(' ')
['Hola', 'World!']
```

Unir (join)

En contraposición a la operación “separar” tenemos el operador *join* que permite generar una cadena a partir de los elementos de una lista.

Ejemplo:

```
>>> ' '.join(['Hola', 'World!'])  
'Hola Mundo!'
```

Verificando si una cadena existe en otra cadena

Para realizar ésta operación, Python provee 2 operadores, a saber:

- El operador *in*: devuelve True en caso de que una cadena sea parte de la otra
- El operador *find*: devuelve el menor índice donde encuentra la cadena buscada

Ejemplo:

```
>>> cadena = 'Hola Mundo!'  
>> "und" in cadena  
True  
  
>>> cadena.find("und")  
6
```

Formateando cadenas

Caracteres de escape

Caracter	Imprime
----------	---------

\'	Comilla simple
\"	Comilla doble
\t	tabulador
\n	Salto de línea
\\	Contrabarra

Raw (pura) Strings

Una cadena “raw” es una cadena donde de forma explícita se le indica al intérprete de Python que ignore todos los caracteres de escape y que imprima todas las contrabarras y demás secuencias de escape que encuentre.

Ejemplo:

```
>>> print('Hola\tMundo!')
'Hola    Mundo!'

# Si se genera una raw string
>>> print(r'Hola\tMundo!')
'Hola\tMundo!'
```

String formatting

En python existen 3 formas de reemplazar en un string el valor de una variable:

1. Utilizando %
2. Utilizando la función *format*
3. Utilizando f-string

Ejemplos

```
>>> nombre = "Juan"
```

```
# Utilizando operador %
"%s tiene %d años" % (nombre, 18)
"Juan tiene 18 años"

# Utilizando función format
"{1} tiene {0} años".format(18, nombre)
"Juan tiene 18 años"

# Utilizando f-string
>>> edad = 18
"{nombre} tiene {edad} años"
"Juan tiene 18 años"
```

Controlando el flujo del programa

Para controlar el flujo de ejecución de nuestra aplicación, en Python, como tantos otros lenguajes, disponemos de estructuras de control y de repetición.

Estructuras de control

En ciertas ocasiones necesitamos que cierta porción de código se ejecute sólo si se cumple una determinada condición. Es en estos casos donde la estructura de control *if...else* entra en acción.

Veamos cada una de sus partes:

If

Permite ejecutar un bloque de código cuando la condición evaluada es verdadera.

Analicemos su sintaxis:


```

if condicion:
    sentencia_1
    sentencia_2
    ....
    sentencia_N
otras_sentencias

```

} Bloque de código a ejecutar en caso de que se cumpla la condición

Como vemos en la imagen anterior, la estructura de control *if* posee una condición a evaluar y un bloque de código a ejecutar en caso de que la condición evaluada sea verdadera.

Cabe destacar que a diferencia de otros lenguajes, el indentado cobra real importancia ya que nos permite definir bloques de código.

Ejemplo:

```

>>> if 5 > 3:
...     print("5 es mayor a 3")
>>> "5 es mayor a 3"

```

Operadores relacionales

Operador	Significado	Ejemplo
==	igualdad	if x == y
!=	Distinto	if x != y
>	Mayor que	if x > y
>=	Mayor o igual que	if x >= y
<	Menor que	if x < y

<=	Menor igual que	if x <= y
----	-----------------	-----------

if...else

En la sección anterior mencionamos que la estructura de control *if* nos permite ejecutar un bloque de código en caso de que la condición evaluada sea verdadera. Ahora bien ¿Qué ocurre si necesitamos ejecutar otro bloque de código en caso de que la condición sea falsa? Es ahí donde es utilizada la estructura de control que vamos a describir a continuación:

```
if condicion:
    sentencia_1
    ....
    sentencia_N
else:
    sentencia_x
    ....
    sentencia_M
otras_sentencias
```

} Bloque de código a ejecutar en caso de que se cumpla la condición

} Bloque de código a ejecutar en caso de que no se cumpla la condición

Como vemos en la imagen anterior, la estructura de control *if...else* posee una condición a evaluar y un bloque de código a ejecutar en caso de que la condición evaluada sea verdadera y otro bloque de código a ejecutar en caso de que la condición sea falsa.

Ejemplo:

```
>>> if 2 > 3:
...     print("2 es mayor a 3")
...else:
...     print("2 es menor que 3")
>>> "2 es menor que 3"
```

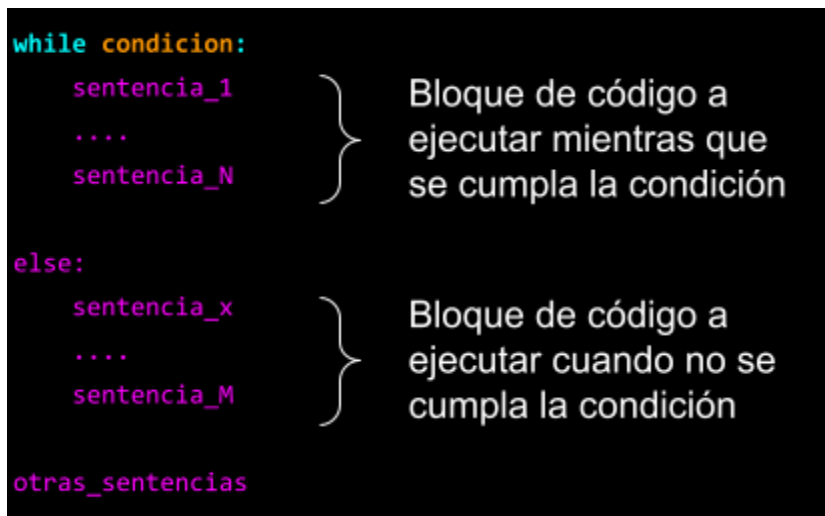
elif

Estructuras de repetición

While

Esta estructura nos permite repetir un bloque de código mientras que la condición evaluada sea verdadera.

Se dice que es un ciclo 0 - infinito ya que permite repetir instrucciones 0, 1, o N veces. Generalmente es utilizado cuando no se conoce a priori la cantidad de iteraciones a realizar.



Ejemplo:

```
>>> x = 4
>>> while x:
...     print(x)
...     x -= 1
... else:
...     print("fin del ciclo!")

4
3
2
```

```
1
fin del ciclo!
```

For

Esta estructura de repetición, permite iterar sobre los elementos de un objeto “iterable” (diccionario, lista, set, string, tupla), a diferencia de lo que ocurre en otros lenguajes donde nos permite repetir una secuencia de instrucciones una cantidad finita y predefinida de veces.

```
for e in elements:
    sentencia_1
    ....
    sentencia_N
else:
    sentencia_x
    ....
    sentencia_M
otras_sentencias
```

} Bloque de código a ejecutar por cada elemento del iterable

} Bloque de código a ejecutar si el ciclo finaliza correctamente

Ejemplo:

```
>>> lista = [1,2,3,4]
>>> for n in lista:
...     print(n)
... else:
...     print("fin del for!")

1
2
3
3
fin del for!
```

Función enumerate

Cuando necesitamos acceder al índice de los elementos del *iterable* que estamos recorriendo, podemos utilizar la función ***enumerate***

```
>>> lista = [1,2,3,4]
>>> for index, num in enumerate(lista):
...     print(f'Idx: {index}, Nro: {n}')
Idx: 0, Nro: 1
Idx: 1, Nro: 2
Idx: 2, Nro: 3
Idx: 3, Nro: 4
```

Como vemos, la función *enumerate*, agrega un índice al iterable que estamos recorriendo. Por defecto comienza en 0 (cero), pero si utilizamos el segundo parámetro de la función, podemos indicar el índice a partir de dónde comienza.

```
>>> lista = [1,2,3,4]
>>> for index, num in enumerate(lista, 4):
...     print(f'Idx: {index}, Nro: {n}')
Idx: 4, Nro: 1
Idx: 5, Nro: 2
Idx: 6, Nro: 3
Idx: 7, Nro: 4
```

Unpacking estructuras en un ciclo for

Al procesar un iterable con una estructura for, podemos obtener de forma simultánea los valores contenidos en los elementos del mismo.

Ejemplo:

```
>>> T = [(1, 2), (3, 4), (5, 6)]
>>> for (a, b) in T:
...     print(a, b)

1,2
3,4
5,6
```

Esta forma de procesar los elementos de un iterable, resulta muy útil cuando procesamos un diccionario:

```
>>> student = {'name': 'Juan', 'age': 30}
>>> for k, v in student.items():
...     print(k, v)

name Juan
age 30
```

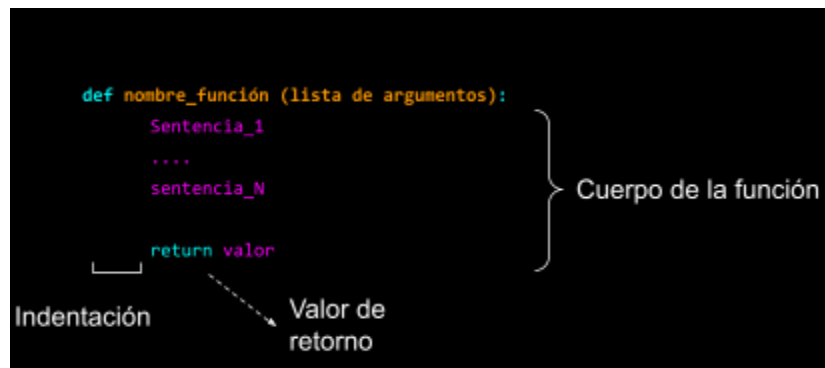
Funciones

Definición

Las funciones son el primer paso para la reutilización del código. Nos permiten definir un bloque de código reutilizable que, como tal, se puede usar repetidamente dentro de un programa.

Python, como la mayoría de los lenguajes, proporciona funciones integradas como `print`, `len` o `type`, pero también podemos definir funciones propias para usarlas dentro de los programas desarrollados. A estas últimas las denominaremos funciones de usuario.

La sintaxis para definir una función propia, o de usuario, es:



Vemos que existen algunos “elementos” particulares:

- `def`: palabra clave que nos permite definir una función
- `nombre_función`: es el nombre que le daremos a nuestra función. Debemos tratar de que sea lo más descriptivo posible
- `lista de argumentos`: lista opcional de argumentos que recibirá la función. Los argumentos nos permiten enviar datos/información a nuestra función.
- `Valor de retorno`: es el valor que devolverá nuestra función. También es opcional.

Ejemplo:

Una función simple que podemos definir es:

```
>>> def saludar():  
...     print("Hola mundo!")
```

Invocando funciones

Para invocar una función, una vez definida, lo haremos usando el nombre que dimos y agregando paréntesis

Ejemplo:

```
>>> def saludar():  
...     print("Hola mundo!")  
  
>>> saludar()
```

Argumentos

Como mencionamos anteriormente, los argumentos de una función permiten que la misma reciba información

Cuando invocamos a una función y pasamos parámetros, los mismos pasan por copia o valor. Esto significa que se realiza una copia del valor de los datos enviados a la función, por lo que si modificamos los mismos dentro de la función, al retornar, los datos no reflejarán los cambios realizados.

Existen 3 tipos de argumento que podemos utilizar al momento de definir una función.

Posicionales

Son los argumentos más comunes. Al invocar la función los valores son copiados según la posición de los mismos en los argumentos de la función.

Ejemplo:

```
>>> def mi_funcion(nombre, edad):  
...     print(f"{nombre} tiene {edad} años")  
  
>>> mi_funcion('Juan', 30)  
Juan tiene 30 años
```

Nombrados (Keyword arguments)

Los argumentos posicionales, si bien son los más comunes, tienen la “desventaja” de que debo conocer y respetar el orden en el que fueron definidos en la función al momento de invocar a la misma.

Para solucionar este problema podemos utilizar los argumentos nombrados o *keyword arguments*. Estos permiten invocar una función sin respetar el orden en el que fueron definidos, sólo debo hacer referencia al nombre dado al argumento en la definición de la función.

Ejemplo


```
>>> def mi_funcion(nombre, edad):
...     print(f"{nombre} tiene {edad} años")

>>> mi_funcion(edad=30, nombre='Juan')
Juan tiene 30 años
```

Nota: podemos combinar argumentos posicionales y nombrados en la misma invocación. Sólo hay que tener en cuenta que debemos utilizar los posicionales antes que los nombrados.

Cantidad de parámetros variables (*args y **kwargs)

Como mencionamos en secciones anteriores, el uso de parámetros nos permite enviar información a una función. Ahora bien, en las situaciones ejemplificadas, la cantidad de argumentos que puede recibir una función queda definida al momento de implementar la función.

¿Qué ocurre si necesitamos definir una función que a priori no conoce cuantos parámetros va a recibir?

Para resolver este tipo de situaciones, Python provee 2 mecanismos para trabajar con una cantidad variable de argumentos: **args* y ***kwargs*.

Una nota antes de comenzar:

- Tanto *args* como *kwargs* son nombres tomados por convención, es decir que no son palabras claves que debemos respetar sino que podemos cambiarlos por un nombre más descriptivo

*args

Cuando definimos una función utilizando el argumento **args* todos los parámetros que recibe la función (que no pueden ser asociados con un parámetro posicional) son contenidos en una tupla, por lo tanto, para accederlos podemos utilizar todas las operaciones definidas para dicho tipo de dato.

Ejemplo

```
>>> def saludar(*args):
...     print(f"Hola args[0]")
```

```
>>> mi_funcion('Juan')
Hola Juan

# También podemos definir la función como
>>> def saludar(*args):
...     print(f"Hola args[0], hola args[1]")

>>> mi_funcion('Juan', 'Pepe')
Hola Juan, hola Pepe
```

**kwargs

Similar al punto anterior, cuando utilizamos el doble asterisco como prefijo de un argumento todos los parámetros que recibe la función (que no pueden ser asociados con un parámetro posicional) son contenidos en un diccionario, por lo tanto, para accederlos podemos utilizar todas las operaciones definidas para dicho tipo de dato.

Ejemplo

```
>>> def mi_funcion(**kwargs):
...     print(kwargs)

>>> mi_funcion(nombre='Juan', edad=30)
{'nombre': 'Juan', 'edad': 30}

# También podemos invocar a la función utilizando un diccionario
mi_diccionario = {'nombre': 'Pepe', 'edad': 20}
>>> mi_funcion(**mi_diccionario)
{'nombre': 'Pepe', 'edad': 20}
```



Unidad 2

Diseño Orientado a Objetos

Clases y Objetos

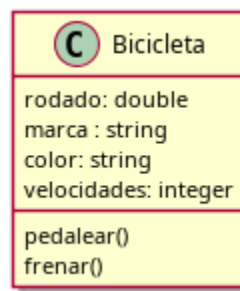
Introducción

Las clases y objetos son dos aspectos principales de la programación orientada a objetos.

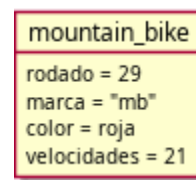
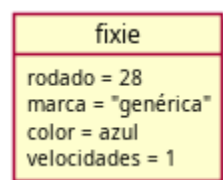
Podríamos decir que una clase es el plano, plantilla o molde a partir del cual se crean los objetos individuales. Los objetos son la “materialización” de dichas clases.

Por ejemplo, podríamos tener una clase “bicicleta” que nos permita modelar objetos que representan una bicicleta y luego instanciar varios objetos de tipo “bicicleta”. Cada uno de estos objetos tendrá las propiedades de una bicicleta, pero con distintos valores.

Siguiendo con el ejemplo, si nuestra clase *bicicleta* fuese definida de la siguiente manera:



Podríamos tener objetos como:



Ambos objetos, son instancias de la clase Bicicleta.

Antes de comenzar con lo específico del lenguaje, es necesario recordar que en Python, todo es un objeto: números, cadenas, listas, funciones, incluso las clases mismas.

Por ejemplo, cuando en Python escribimos `num = 42`, se crea un nuevo objeto de tipo integer con el valor 42, y asigna su referencia al nombre num.

Definición de una clase

Para crear nuestros propios objetos, antes debemos definir una clase. La sintaxis para definir una clase es la siguiente:



A diagram on a black background illustrating Python class syntax. On the left, the code `class nombre_de_la_clase:` is followed by indented lines `método_a:`, `...`, and `método_n`. A bracket on the right groups these indented lines under the label "Métodos de la clase". A bracket on the left under the class name is labeled "Indentación".

Por ejemplo, si quisiéramos crear nuestra clase bicicleta:

```
>>> class Bicicleta:
...     pass
>>>
```

La palabra reservada `pass` indica que la clase es una clase vacía.

El método constructor

En Python existe un método especial que se ejecuta de forma automática cada vez que se instancia un objeto de una clase. Este método se llama `__init__` y permite ejecutar acciones necesarias para inicializar un objeto.

Ejemplo:

```
class Bicicleta:

    def __init__(self):
        # acciones para inicializar el objeto
```

¿Qué es el parámetro `self`?

El parámetro `self` permite hacer referencia al objeto en sí mismo. Trazando una analogía con otros lenguajes como Java, sería el análogo a la palabra reservada `this`

Dos puntos importantes a tener en cuenta sobre el parámetro `self`:

1. No necesariamente debe llamarse "self". Podría utilizar cualquier otra palabra.
2. Debe ser el primer parámetro de un método de clase.

Atributos

Toda clase está compuesta por atributos y métodos (que se verán más adelante). Los atributos son las propiedades que hacen que 2 objetos de una misma clase sean diferentes uno del otro.

En el caso de nuestra clase Bicicleta, `rodado`, `marca`, `color` y `velocidades` son los atributos de la clase. Cada objeto de la clase Bicicleta tendrá los atributos mencionados con los valores correspondientes.

Existen dos tipos de atributos, los atributos de clase y los de instancia.

Atributos de instancia

Los atributos de instancia son variables únicas para cada objeto, es decir, para cada instancia de una clase. Cada objeto de esa clase tiene su propia copia de esa propiedad, por lo que a diferencia de los atributos de clase, cualquier cambio realizado en dichas propiedades no se refleja en otros objetos de esa clase.

En nuestro ejemplo de la clase Bicicleta, cada bicicleta (instancia) posee un valor propio para `tipo`, `rodado` y `velocidades`.

¿Cómo definimos los atributos de instancia en python?

A diferencia de otros lenguajes como Java, C++ o C#, los atributos de instancia son definidos dentro del método `__init__`.

Ejemplo:

```
class Bicicleta:

    def __init__(self, tipo, rodado, velocidades):
        self.tipo = tipo
        self.rodado = rodado
        self.velocidades = velocidades
```

Ámbito de los atributos de instancia

En Python todas las propiedades (y los métodos) de una clase son públicas, por lo que la abstracción y encapsulamiento no podemos a priori asegurarlo.

Ahora bien, dicho esto, existen algunos artilugios programáticos que nos permiten “ocultar” o al menos hacer que el acceso a las propiedades de una clase sea menos fácil.

Una primer alternativa consiste en nombrar las propiedades de la clase, anteponiendo un doble guión bajo al nombre: `__mi_propiedad`. Veamos un ejemplo:

```
class MiClase:
    def __init__(self, valor):
        self.__mi_prop_privada = valor

# en caso de instanciar un objeto de esta clase y tratar de acceder a la
# propiedad, obtendremos el siguiente error:

>>> m = MiClase(1)
>>> m.__mi_prop_privada
Traceback (most recent call last):
File "<input>", line 1, in <module>
m.__mi_prop_privada
AttributeError: 'MiClase' object has no attribute '__mi_prop_privada'
```

¿Qué ocurre internamente? ¿Logramos generar una propiedad privada?

La respuesta es no. Lo que ocurre internamente es que el intérprete de Python renombra dicha propiedad de la siguiente forma: `_MiClase__mi_prop_privada` por lo que sigue siendo pública, sólo que posee otro nombre.

Siguiendo nuestro ejemplo, si tratáramos de acceder escribiendo `m._MiClase__mi_prop_privada` podríamos acceder a la propiedad sin problemas.

Una segunda alternativa, que se acerca más a un lenguaje como Java o C++, es definir al atributo de la clase como una propiedad.

Esta segunda alternativa podemos lograrla de 2 formas distintas, la primera es utilizando la función `property()` y la segunda es utilizando `decorators`.

Veamos algunos ejemplos de ambas opciones:

```
class MiClase:
    def __init__(self, valor):
        self.__mi_prop_privada = valor

    def get_mi_propiedad(self):
        return self.__mi_prop_privada

    def set_mi_propiedad(self, valor):
        self.__mi_prop_privada = valor

    mi_prop = property(get_mi_propiedad, set_mi_propiedad)

# al instanciar un objeto de esta clase y acceder a la propiedad:
>>> m = MiClase(1)
# Podemos acceder como si fuese una propiedad pública
>>> m.mi_prop
1
>>> m.mi_prop = 3
>>> m.mi_prop
3
```

Lo que ocurre en este ejemplo, es que al ejecutar `m.mi_prop` o `m.mi_prop = 3` lo que se ejecuta internamente son las funciones `get_mi_propiedad` y `set_mi_propiedad`

La segunda opción para lograr el mismo resultado es utilizar `decorators`:

```
class MiClase:
    def __init__(self, valor):
        self.__mi_prop_privada = valor
```



```

@property
def mi_prop(self):
    return self.__mi_prop_privada

@mi_prop.setter
def mi_prop(self, valor):
    self.__mi_prop_privada = valor

# El ejemplo de uso es idéntico al ejemplo anterior

```

Aspectos a destacar sobre esta última opción:

- `@property` debe escribirse sobre el getter de la propiedad. También es el que define el nombre de la misma.
- `@<nombre_propiedad>.setter` debe escribirse sobre el setter de la propiedad. El nombre del método debe coincidir con el nombre de la propiedad.

Al igual que en la opción anterior, al ejecutar `m.mi_prop` o `m.mi_prop = 3` lo que se ejecuta internamente son las funciones `mi_prop` *getter* y *setter* respectivamente.

Una ventaja que obtenemos al definir propiedades, es que podemos hacer que las mismas sean propiedades de sólo lectura. Para lograrlo basta con definir el `getter` y no el `setter` de la propiedad.

Atributos de clase

Los atributos de clase, son propiedades comunes a todos los objetos de la misma. Esto significa que cada objeto creado a partir de dicha clase, tendrá el mismo valor en la/s propiedad/es que se hayan definido como 'atributos de la clase'.

En Python, un objeto puede modificar el valor de una propiedad de clase sin afectar al resto de los objetos ya que sólo modifica el valor correspondiente a su instancia.

Veamos un ejemplo:

```

class Bicicleta:
    color = 'rojo'

    def __init__(self, tipo, rodado, velocidades):

```

```
        self.tipo = tipo
        self.rodado = rodado
        self.velocidades = velocidades

b1 = Bicicleta('fixie', 28, 1)
b2 = Bicicleta('mb', 29, 21)

print(b1.color)
"rojo"

print(b2.color)
"rojo"

b1.color = 'azul'
print(b1.color)
"azul"

print(b2.color)
"rojo"
```

Ahora bien, al ser un atributo de clase, esto significa que sólo hay una copia de esa variable que se comparte con todos los objetos por lo que podemos acceder al mismo a través del nombre de la clase, ya sea para obtener como para asignar un valor.

Tener en cuenta que en caso de asignar un valor, esta acción modificará los valores de todos los objetos.

Ejemplo:

```
b1 = Bicicleta('fixie', 28, 1)
b2 = Bicicleta('mb', 29, 21)

print(b1.color)
"rojo"

print(b2.color)
```

```
"rojo"

# Cambio el valor de la propiedad de la clase
Bicicleta.color = 'azul'

print(b1.color)
"azul"

print(b2.color)
"azul"
```

Métodos

Como mencionamos previamente, un objeto posee datos y comportamiento, es decir, atributos y métodos respectivamente. Mientras que las propiedades nos permiten representar los datos de un objeto, los métodos nos permiten representar las acciones posibles que puede llevar a cabo un objeto, es decir, su comportamiento.

Métodos de instancia

Los métodos de instancia no son más que funciones definidas dentro de una clase. Los mismos sólo pueden ser invocados sobre una instancia de un objeto.

Ejemplo

```
class Bicicleta:

    def __init__(self, tipo, rodado, velocidades):
        self.tipo = tipo
        self.rodado = rodado
        self.velocidades = velocidades

    def avanzar(self, metros):
        for i in range(metros):
            print(f'avanzando {i+1} metros...')
```

```

    def detenerse(self):
        print('bicicleta detenida')

# Creamos una instancia
b = Bicicleta('mb', 29, 21)

# Luego, sobre la instancia invocamos al método avanzar
b.avanzar(3)

# obtenemos la siguiente salida
avanzando 1 metros...
avanzando 2 metros...
avanzando 3 metros...

```

Métodos de clase

Un método de clase es un método que se comparte entre todos los objetos de una misma clase. A diferencia de los métodos de instancia, la invocación se puede realizar tanto sobre la clase como sobre una instancia de la misma, es decir un objeto.

La particularidad que poseen los métodos de clase es que debe poseer un parámetro llamado `cls` el cuál hará referencia a la clase.

Ejemplo

```

class Bicicleta:

    @classmethod
    def mostrar_clase(cls):
        print(f'La clase es: {cls}')

# Podemos invocar al método mostrar_clase sin tener una instancia
Bicicleta.mostrar_clase()

```

```
# obtenemos la siguiente salida
La clase es: <class '__main__.Bicicleta'>
```

Los métodos de clase podemos utilizarlos con o sin una instancia de la misma.

Por lo general utilizaremos estos métodos para generar instancias de la clase en cuestión

Métodos estáticos

A diferencia de los métodos de clase o los de instancia, los métodos estáticos no están relacionados a una clase o a la instancia de la misma, por lo que no pueden acceder a las propiedades de la clase o al estado de un objeto.

Estos métodos tampoco reciben entre sus parámetros una referencia a la clase o su instancia.

Ejemplo:

```
class Bicicleta:

    @staticmethod
    def convertir_kmh_a_ms(km, hora):
        return (km*1000, hora*3600)

# Podemos invocar al método convertir_kmh_a_ms sin tener una instancia
ms = Bicicleta.conertir_kmh_a_ms(27, 1)
```

Métodos `__eq__`, `__str__` y `__hash__`

Obteniendo la representación de un objeto (`__str__`)

Supongamos que tenemos la clase *Alumno* con la siguiente estructura:

```
class Alumno:

    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

# instanciamos un objeto y lo imprimimos
alumno = Alumno("Juan", "Perez")
print(alumno)
<__main__.Alumno object at 0x7f7ac30777c0>
```

Vemos que la salida sólo nos brinda información acerca de la clase y la posición de memoria donde se ubica la instancia.

Ahora bien, si necesitamos obtener una representación del objeto legible por una persona, lo que debemos hacer es implementar el método `__str__`

```
class Alumno:

    def __init__(self, nombre, apellido):
        self.__nombre = nombre
        self.__apellido = apellido

    def __str__(self):
        return f'{self.__apellido}, {self.__nombre}'

# instanciamos un objeto y lo imprimimos
alumno = Alumno("Juan", "Perez")
print(alumno)
Perez, Juan
```

En este segundo ejemplo vemos que al utilizar la función `print` sobre la instancia `alumno`, el intérprete de Python invoca al método `__str__` implementado en la clase `Alumno`.

Comparando objetos (__eq__ y __hash__)

Metodo __eq__

Retomando el ejemplo de la clase Alumno, imaginemos que sólo disponemos del nombre y apellido para identificar a un alumno, por lo que tenemos el siguiente ejemplo:

```
class Alumno:

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return f'{self.apellido}, {self.nombre}'

alumno_1 = Alumno("Juan", "Perez")
alumno_2 = Alumno("Juan", "Perez")

Alumno_1 == alumno_2
False
```

Vemos que generamos 2 instancias de la misma clase, con los mismos valores, pero al comparar las instancias entre sí, obtenemos que no son iguales.

Ahora bien, en el dominio de nuestro ejemplo si 2 alumnos poseen el mismo nombre y apellido entonces deberíamos considerar que es el mismo alumno. Para lograrlo debemos implementar el método `__eq__` que es el utilizado por el intérprete de Python cuando comparamos objetos.

```
class Alumno:

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido
```

```

def __str__(self):
    return f'{self.apellido}, {self.nombre}'

def __eq__(self, otra_instancia):
    return self.nombre == otra_instancia.nombre and \
           self.apellido == otra_instancia.apellido

alumno_1 = Alumno("Juan", "Perez")
alumno_2 = Alumno("Juan", "Perez")

# Ahora al comparar las instancias vemos que el resultado es True
alumno_1 == alumno_2
True

# Si generamos otra instancia vemos que el resultado es False
alumno3 = Alumno("Maria", "Perez")
alumno == alumno3
False

```

Por último, podemos agregar una verificación al método `__eq__` para evitar realizar comparaciones con objetos de distintas clases:

```

class Alumno:

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return f'{self.apellido}, {self.nombre}'

    def __eq__(self, otra_instancia):
        result = False
        if isinstance(otra_instancia, Alumno):
            result = self.nombre == otra_instancia.nombre and \
                    self.apellido == otra_instancia.apellido

```



```
return result
```

En este último ejemplo, utilizamos la función `isinstance` para verificar que la variable `otra_instancia` es un objeto de la clase `Alumno`.

Metodo `__hash__`

El método `__hash__` al igual que el método `__eq__` es utilizado para comparar objetos, pero a diferencia de este último que devuelve un valor booleano, el método `__hash__` debe devolver un valor entero.

Este método es utilizado por diccionarios y sets al momento de agregar un objeto a los mismos y por tal motivo si implementamos el método `__hash__` sobre una clase propia, debemos tener en cuenta que durante todo el ciclo de vida del objeto, el hash debe ser siempre el mismo.

Veamos un ejemplo:

```
class Alumno:

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def __str__(self):
        return f'{self.apellido}, {self.nombre}'

    def __eq__(self, otra_instancia):
        ...

    def __hash__(self):
        return hash(self.nombre + self.apellido)

    return result
```

Algunas consideraciones más:

- Los objetos que al compararse son iguales, también deben tener el mismo valor de hash.
En otras palabras:
`Si a == b => hash(a) == hash(b)`
- La recíproca del postulado anterior no siempre se cumple.
- No es estrictamente necesario implementar el método `__hash__`

Herencia

En el paradigma de programación orientado a objetos, la herencia podemos definirla como una técnica que nos permite compartir y, como consecuencia de esto, no repetir código común entre clases que guardan cierta relación entre sí. La herencia nos ayuda a representar/implementar la relación “A es un B”.

Al implementar herencia entre clases, la clase que hereda, automáticamente recibe todos los atributos y métodos de la clase desde la cual hereda.

Si la clase A hereda de la clase B, se dice que la clase A es una clase derivada o subclase de B; mientras que la clase B se dice que es la clase base o “superclase”

Retomando el ejemplo de nuestra clase Bicicleta, si nuestra aplicación nos lleva a tener que implementar distintos tipos de bicicletas, sería conveniente utilizar la herencia como mecanismo para encapsular el comportamiento y las propiedades comunes a todas las bicicletas en una única clase (la clase base).

Veamos un ejemplo de cómo implementar la herencia en Python:

```
class Bicicleta:

    def __init__(self, rodado, velocidades):
        self.rodado = rodado
        self.velocidades = velocidades

    def avanzar(self, metros):
        ...

    def detenerse(self):
        ...
```

```
class Fixie(Bicicleta):
    pass

class MountainBike(Bicicleta):
    pass

# Creamos algunas instancias
fixie = Fixie(28, 1)
mountain = MountainBike(29, 21)
```

En este ejemplo, aunque sencillo, vemos cómo implementar herencia entre clases.

Tenemos la clase base “Bicicleta” y luego 2 clases que heredan de esta última que son “Fixie” y “MountainBike”. En este ejemplo (simple) no poseen código propio por eso utilizamos la instrucción `pass`. Pero como mencionamos anteriormente, poseen los atributos y métodos de la clase “Bicicleta”.

Agregando propiedades

Las clases derivadas pueden tener, además, sus propios atributos. Por ejemplo si una subclase fuese “BicicletaElectrica”, la misma podría tener un atributo “bateria” que ni su clase base ni el resto de las clases derivadas tendría sentido que lo tengan.

```
class Bicicleta:

    def __init__(self, rodado, velocidades):
        self.rodado = rodado
        self.velocidades = velocidades

    def avanzar(self, metros):
        ...

    def detenerse(self):
        ...
```

```
class Electrica(Bicicleta):

    def __init__(self, rodado, velocidades, bateria):
        self.rodado = rodado
        self.velocidades = velocidades
        self.bateria = bateria
```

Modificando el comportamiento de las clases derivadas

En ocasiones, la situación planteada, nos presentará comportamientos distintos entre los objetos que participan, a pesar de que los mismos se encuentren relacionados, lo que nos llevará a agregar o modificar el comportamiento de las clases derivadas.

Retomando el ejemplo anterior, una bicicleta eléctrica puede prenderse, mientras que el resto de las bicicletas no poseen dicho comportamiento. En este caso estaríamos agregando comportamiento a la clase derivada.

Por otra parte, cuando en una clase derivada, modificamos un método que fue definido en su clase base, hablamos de sobrescribir el método (override).

Veamos algunos ejemplos:

```
class Electrica(Bicicleta):

    # El método __init__ se está sobrescribiendo.
    def __init__(self, rodado, velocidades, bateria):
        self.rodado = rodado
        self.velocidades = velocidades
        self.bateria = bateria
        self.estado = "apagada"

    # El método prender se agrega a la clase Electrica
    def prender(self):
        self.estado = "prendida"
```

La función super()

En algunos casos, cuando sobreescribimos un método en una clase derivada, también necesitamos ejecutar el código de la clase base. Para esto disponemos de la función `super()`.

Esta función nos permite hacer referencia a la clase base.

El ejemplo anterior, podría reescribirse para reutilizar el código del método `__init__` de la clase “Bicicleta”:

```
class Electrica(Bicicleta):  
  
    # El método __init__ se está sobreescribiendo.  
    def __init__(self, rodado, velocidades, bateria):  
        super().__init__(rodado, velocidades)  
        self.bateria = bateria  
        self.estado = "apagada"
```

Clase abstracta

Una clase abstracta es una clase que no puede instanciarse, es decir, no vamos a poder crear objetos a partir de dicha clase.

Ahora bien, ¿Para qué me sirve una clase de la cual no puedo generar instancias?

Veremos en las próximas unidades que este tipo de clases son muy importantes para encapsular comportamiento y establecer una interfaz común en un grupo de subclases que sí podrán ser instanciadas. A estas últimas se las denomina clases concretas.

En Python, para indicar que una clase es abstracta debemos utilizar el módulo `abc` (abstract base class).

Ejemplo:

```
from abc import ABC

class Bicicleta(ABC):
    ...
```

En una clase abstracta, comúnmente tendremos métodos abstractos que deberán ser implementados en las subclases.

Para definir un método abstracto, el módulo `abc` nos proporciona el decorator `@abstractmethod`

Ejemplo

```
from abc import ABC, abstractmethod

class Bicicleta(ABC):
    ...

    @abstractmethod
    def frenar(self):
        pass
```

En este caso definimos la clase `Bicicleta` como abstracta, con un método abstracto “frenar”. Si intentamos instanciar un objeto de esta clase obtendremos un error.

El método `frenar` deberá ser implementado en las subclases “Fixie”, “Electrica”, etc.

De esta forma generamos una interfaz común para todas las subclases, que luego mediante el uso del polimorfismo podemos aprovechar.

Cabe destacar que una clase abstracta no estará conformada sólo por métodos abstractos, sino que también puede contener métodos concretos, es decir, con código. Las subclases podrán utilizar dichos métodos a través de la función `super()`

Herencia Múltiple

En Python, también podemos hacer que una clase herede de varias clases base. A esto se le denomina “herencia múltiple”.

Al igual que con la “herencia simple”, todas las propiedades y métodos de las clases bases son heredados por la subclase en cuestión.

Ejemplo:

```
class Bicicleta:
    def avanzar(self):
        pass

    def frenar(self):
        pass

class ArtefactoElectrico:
    def prender(self):
        pass

    def apagar(self):
        pass

# Hereda de Bicicleta y ArtefactoElectrico
class BiciElectrica(Bicicleta, ArtefactoElectrico):
    # A través de la herencia posee los métodos (avanzar, frenar,
    prender, apagar)
    pass
```

Polimorfismo

El polimorfismo es un concepto muy importante en la programación orientada a objetos, pero no es exclusivo de la misma.

Existen diferentes “tipos” de polimorfismo.

Un caso es el de polimorfismo paramétrico que ocurre cuando una misma función puede recibir parámetros de distinto tipo. Un ejemplo de esto es la función `len`

Ejemplo:

```
>>> len("hola mundo")
10
>>> len([1,2,3])
3
```

Vemos que en la primer invocación recibe una cadena mientras que en la segunda recibe una lista. En ambos casos devuelve la cantidad de elementos.

Por otra parte, y ahora sí enfocándonos en el paradigma de programación orientado a objetos, el concepto de polimorfismo se puede definir como:

“...la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.”³

Volviendo sobre el ejemplo de las bicicletas, podemos decir que todas las bicicletas pueden avanzar y frenar, es decir, todas las subclases de la clase “Bicicleta” entienden los mensajes “avanzar” y “frenar” por lo que independientemente del tipo de bicicleta con el que estemos trabajando puedo invocar a cualquiera de los métodos mencionados.

Esta característica permite que desde el código cliente, es decir quién usa a las clases Bicicleta y sus subclases, se desentienda del tipo de bicicleta con el que está trabajando. Esto brinda mayor flexibilidad y menor acoplamiento en el código.

Ejemplo:

```
# supongamos que tengo una lista de bicicletas (instancias)
bicicletas = [fixie, mountain_bike, electrica...]

# podría hacer lo siguiente:
for bicicleta in bicicletas:
    bicicleta.avanzar()
```

Como se muestra en el ejemplo, sin importar el tipo de bicicleta se invoca al método “avanzar”.

³ [https://es.wikipedia.org/wiki/Polimorfismo_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Polimorfismo_(inform%C3%A1tica))

Excepciones

Durante la ejecución de un programa pueden ocurrir diferentes tipos de errores:

- errores de codificación cometidos por el programador
- errores por entrada incorrecta de datos
- otros errores imprevistos.

Cuando ocurre un error, Python activa el proceso de manejo de excepciones. Este proceso detiene el programa e imprime un mensaje de error en la consola.

Ejemplo:

```
>>> res = 3 / 0
Traceback (most recent call last):
File "<input>", line 1, in <module>
res = 3 / 0
ZeroDivisionError: division by zero
```

Como vemos en el ejemplo, es un comportamiento poco amigable para el usuario. Incluso en el caso en que no se pueda revertir el error, nuestra aplicación debería ser capaz de manejar el error y devolver al usuario un mensaje más legible y entendible.

Manejando excepciones

Con el objetivo de darle mayor robustez a nuestra aplicación es que debemos atrapar y procesar las excepciones que ocurran durante la ejecución de nuestra aplicación.

Para esto Python, nos ofrece la construcción `try - except` que nos permite “intentar” ejecutar acciones y en caso de que se produzca alguna excepción darle el tratamiento correspondiente.

De manera simplificada, podemos escribir:

```
try:
    # Bloque de código a ejecutar
except:
    # Procesamiento del error
```

¿Cómo funciona la cláusula try-except?

1. Se comienza a ejecutar el bloque de instrucciones que se encuentran dentro del bloque `try`
 - a. Si no se produce ninguna excepción, se omite la cláusula `except` y se completa la ejecución de la instrucción `try`.
2. Si ocurre una excepción en cualquier instrucción de la cláusula `try`, el resto de las instrucciones se omiten y se ejecuta la cláusula `except`.

Ordenando esto último, diremos que dentro del bloque `try` pondremos las instrucciones que pueden generar una excepción y en el bloque `except` daremos tratamiento al error.

Ejemplo:

```
>>> try:
...     3 / 0
... except:
...     print("Error al intentar realizar la división")

Error al intentar realizar la división
```

Ahora bien, muchas veces vamos a necesitar responder de diferente forma según el error que se haya producido. Para esto en Python podemos atrapar diferentes excepciones, una (o más) en cada bloque `except`.

Veamos el siguiente ejemplo:

```
def login(user, password):
    pass

try:
    login(user, password)
except EmptyUserException:
    print("Debe ingresar un usuario.")
except EmptyPasswordException:
    print("Debe ingresar un password.")
except WrongCredentialsException:
```

```
print("Usuario o Password incorrecto.")
except Exception as e:
    print(f"Error inesperado! Error: {e}")
```

En este ejemplo vemos cómo podemos atrapar varias excepciones en la misma operación. El último bloque `except` además muestra el error producido.

Si quisiéramos dar el mismo tratamiento a más de 1 excepción, podemos realizar lo siguiente:

```
try:
    operacion()
except (Excepcion_1, Excepcion_2):
    # tratamiento del error para ambas excepciones
```

Cláusula `try...except...finally`

Python permite especificar la cláusula `finally` que se ejecutará más allá del resultado de la operación, es decir, se ejecutará siempre; ya sea que la operación finalizó con éxito o se produjo algún error.

Retomando el primer ejemplo:

```
>>> try:
...     3 / 0
... except:
...     print("Error al intentar realizar la división")
... finally:
...     print("Fin de la operación")

Error al intentar realizar la división
Fin de la operación
```

Y en caso de que la operación no genere errores, tendríamos la siguiente salida:

```
>>> try:
```

```
...     3 / 3
... except:
...     print("Error al intentar realizar la división")
... finally:
...     print("Fin de la operación")

1
Fin de la operación
```

Cláusula try...except...else...finally

Por último veremos la cláusula `else` en el contexto del manejo de excepciones.

Esta cláusula sólo se ejecutará en caso de que no ocurran excepciones.

El orden de ejecución sería:

1. `try`
2. `else`
3. `finally`

Veamos un ejemplo:

```
try:
    dividendo = int(input("Ingrese el dividendo: "))
    divisor = int(input("Ingrese el divisor: "))
except ValueError as verror:
    print("Debe ingresar un número entero!")
else:
    cociente = dividendo / divisor
    print(f'Cociente: {cociente}')
finally:
    print("Fin de la operación")

# al ejecutar
Ingrese el dividendo: 3
Ingrese el divisor: 2
```

```
Cociente: 1.5
Fin de la operación

# en caso de volver a ejecutar
Ingrese el dividendo: 3.1
Debe ingresar un número entero!
Fin de la operación
```

Excepciones definidas por el usuario

Al desarrollar una aplicación, con frecuencia debemos manejar errores pertenecientes al dominio del problema planteado, es decir, errores de lógica de negocio.

En Python podemos crear nuestras propias excepciones creando clases que heredan de la clase `Exception` o de cualquier otra subclase de la misma.

Ejemplo:

```
class EmptyUserException (Exception):
    pass

try:
    login(user, password)
except EmptyUserException:
    print("Debe ingresar un usuario.")
```

Un ejemplo que hace un mejor uso de las excepciones definidas por el usuario podría ser:

```
class EmptyUserException (Exception):
    def __init__(self, message, code):
        self.__message = message
        self.__code = code

    def __str__(self):
```

```
        return f'Error: {self.__code}. {self.__message}'

try:
    raise EmptyUserException("Debe ingresar un usuario", 123)
except EmptyUserException as e:
    print(e)

# obtendremos la siguiente salida:
Error: 123. Debe ingresar un usuario
```



Unidad 3

Patrones de Diseño

Patrones de diseño

Introducción

En ingeniería de software, un patrón de diseño es una solución **repetible** a un **problema común de diseño de software**. No es un diseño terminado que se puede transformar directamente en código, sino que es una descripción o plantilla sobre cómo resolver un problema que se puede utilizar en variadas y diversas situaciones.

Los patrones pueden acelerar el desarrollo al proporcionar paradigmas de desarrollo previamente probados y comprobados. La efectividad del diseño de software radica en tener en cuenta y anticiparse a problemas que pueden no ser visibles hasta que se comienza a implementar. El uso de patrones de diseño ayuda a prevenir problemas en general sutiles, que pueden causar problemas importantes.

Generalmente, las personas solo entienden cómo aplicar ciertas técnicas de diseño a ciertos problemas. El problema, es que estas técnicas son difíciles de aplicar a una gama más amplia de problemas. La idea de los patrones de diseño es que brinden soluciones generales, documentadas en un formato que no requiere detalles específicos vinculados a un problema en particular.

A su vez, los patrones permiten a los desarrolladores comunicarse utilizando nombres bien conocidos y entendidos. Es decir, generan un lenguaje común.

Así, nos encontramos con tres tipos de patrones: los creacionales, los estructurales y los de comportamiento.

Patrones creacionales

Estos patrones tienen que ver específicamente con la instanciación, la creación de objetos. Veamos algunos de ellos.

Abstract Factory

Propósito

Permite crear familias de objetos relacionados sin necesidad de especificar sus clases concretas.

Problema

Imaginemos que estamos creando un simulador de tienda de venta de artículos de home office. El código está compuesto por clases que representan lo siguiente:

1. Una familia de productos relacionados, digamos: Webcam + Headsets + Kit Teclado/Mouse.
2. Algunas variantes de esta familia. Por ejemplo, los productos Webcam + Headsets + Kit Teclado/Mouse están disponibles en tres variantes: los Combos Platinum, Gold y Starter.

Necesitamos una forma de crear objetos individuales para que combinen con otros objetos de la misma familia. Los clientes se incomodan bastante cuando reciben productos que no combinan (por ejemplo: solicitar un Combo Platinum y recibir allí un headset de nivel Starter).

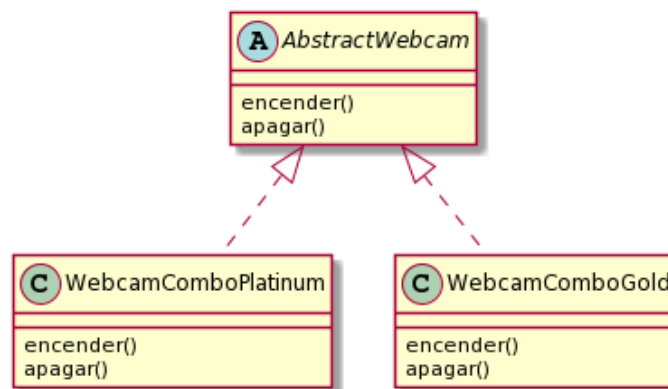
Además, por supuesto: algo que definitivamente no queremos es tener que cambiar el código al sumar nuevos productos o combos. Los comerciantes de esta clase de tiendas actualizan sus catálogos muy seguido, y debemos evitar tener que cambiar el código cada vez que esto suceda.

Solución

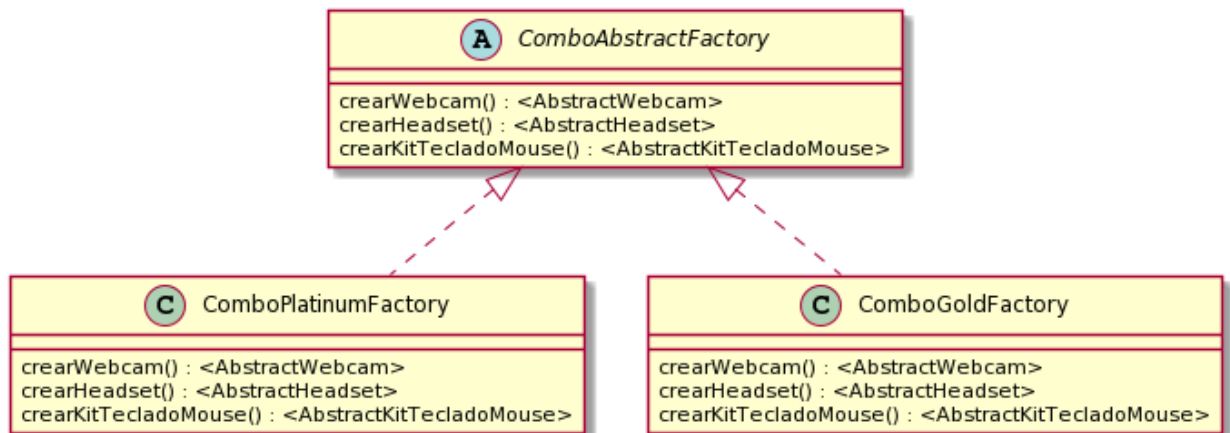
Lo primero que sugiere Abstract Factory es que declaremos de forma explícita interfaces (las cuales en Python trabajamos como clases abstractas) para cada producto diferente de la familia de productos (por ejemplo, Webcam, Headsets o Kit Teclado/Mouse).

Después podemos hacer que todas las variantes de los productos sigan esas interfaces.

Por ejemplo, todas las variantes de Webcam pueden heredar de la clase (abstracta, por supuesto) `AbstractWebcam`, así como todas las variantes de Headset pueden heredar de `AbstractHeadset`, y así sucesivamente.



El paso siguiente consiste en declarar la clase `ComboAbstractFactory`: una clase abstracta con métodos de creación para todos los productos que forman parte de la familia (por ejemplo, `crearWebcam`, `crearHeadset` y `crearKitTecladoMouse`). Estos métodos deben devolver productos abstractos representados por las clases abstractas que extrajimos previamente: `AbstractWebcam`, `AbstractHeadset`, `AbstractKitTecladoMouse`, etc.



¿Qué pasa con las variantes de los productos? Para cada variante de una familia de productos, creamos una clase de fábrica independiente, basada en `ComboAbstractFactory`. Recordemos que una fábrica es una clase que devuelve productos de un tipo particular. Por ejemplo, la `ComboGoldFactory` sólo puede crear objetos de `WebcamComboGold`, `HeadsetComboGold` y `KitTecladoMouseComboGold`.

El código cliente debe funcionar con fábricas y productos a través de sus respectivas abstracciones. Esto nos permite tanto cambiar el tipo de fábrica como la variante del producto que recibe el código cliente, sin tener que descomponerlo.

En resumen: al cliente no le debe importar la clase concreta de la fábrica con la que funciona.

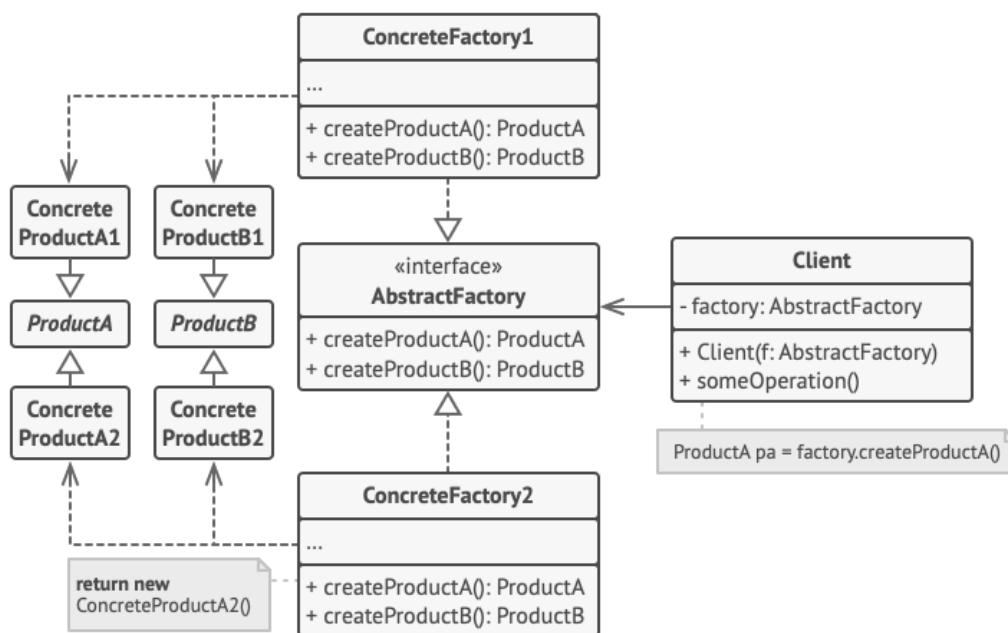
Digamos que el cliente quiere una fábrica para producir una webcam. El cliente no necesita conocer la clase de la fábrica y tampoco importa el tipo de webcam que obtiene. Ya sea el modelo del Combo Platinum o el del Starter, el cliente debe tratar a todas las webcams del mismo modo, utilizando la clase abstracta `Webcam`. Con este sistema, lo único que sabe el cliente sobre la webcam es que se encenderá y apagará. Además, sea cual sea la variante de webcam devuelta, siempre combinará con el Headset o el Kit de Teclado y Mouse producida por el mismo objeto de fábrica.

Queda otro punto por aclarar: si el cliente sólo está expuesto a las interfaces abstractas, ¿cómo se crean los objetos de fábrica? Normalmente, la aplicación crea un objeto de fábrica concreto

en la etapa de inicialización. Justo antes, la aplicación debe seleccionar el tipo de fábrica, dependiendo de la configuración o de los ajustes del entorno.

Estructura

1. Los Productos Abstractos declaran interfaces (clases abstractas) para un grupo de productos diferentes pero relacionados, que forman una familia de productos.
2. Los Productos Concretos son implementaciones distintas de productos abstractos agrupados por variantes. Cada producto abstracto (webcam, headset, etc) debe implementarse en todas las variantes dadas (Combo Platinum, Combo Gold, etc).
3. La interfaz (de nuevo, clase abstracta) Fábrica Abstracta declara métodos para crear cada uno de los productos abstractos.
4. Las Fábricas Concretas implementan métodos de creación de la fábrica abstracta. Cada fábrica concreta se corresponde con una variante específica de los productos y crea específicamente esas variantes de los productos.
5. A pesar de que las fábricas concretas instancian productos concretos, las firmas de sus métodos de creación sí o sí deben devolver los productos abstractos correspondientes. Así, el código cliente que utiliza una fábrica no se acopla a la variante específica del producto de esa fábrica. El Cliente puede funcionar con cualquier variante fábrica/producto concreta, siempre y cuando se comunique con sus objetos a través de interfaces abstractas.



Aplicabilidad

- Usamos Abstract Factory cuando el código debe funcionar con múltiples familias de productos relacionados, pero no queremos que dependa de las clases concretas de esos productos, ya que puede ser que no los conozcamos de antemano, o sencillamente queramos facilitar la extensibilidad.
- Abstract Factory ofrece una interfaz para crear objetos a partir de cada clase de la familia de productos. Mientras el código cree objetos a través de esta interfaz, no habrá que preocuparse por crear la variante errónea de un producto que no combine con los productos que ya ha creado la aplicación.
- Considerar la implementación de Abstract Factory cuando exista una clase con un grupo de métodos de fábrica que nublen su responsabilidad principal.
- En una aplicación bien diseñada, cada clase es responsable tan solo de una cosa. Cuando una clase lidia con distintos tipos de productos, puede que valga la pena extraer sus métodos de fábrica para ponerlos en una clase única de fábrica o una implementación completa de Abstract Factory.

Cómo implementarlo

1. Armar una matriz de distintos tipos de productos frente a variantes de dichos productos.
2. Declarar interfaces (clases abstractas) de producto para todos los tipos de productos. Luego, que todas las clases concretas de productos hereden de estas clases abstractas.
3. Declarar la interfaz de la fábrica abstracta con métodos de creación para cada producto abstracto.
4. Implementar clases concretas de fábrica, una por cada tipo de producto.
5. Crear código de inicialización de la fábrica en algún punto de la aplicación. Deberá instanciar una de las clases concretas de la fábrica, dependiendo de la configuración de la aplicación o del entorno actual. Pasar este objeto de fábrica a todas las clases que construyen productos.
6. Explorar el código y encontrar todas las llamadas directas a constructores de producto. Sustituirlas por llamadas al método de creación correspondiente dentro del objeto de fábrica.

Pros y contras

Pros:

- Sí o sí, los productos que se obtienen de una fábrica son mutuamente compatibles.
- Se evita el acoplamiento entre productos concretos y código cliente.

- Principio de responsabilidad única. Se puede mover el código de creación de productos a un único lugar, logrando que el código sea más fácil de mantener.
- Principio abierto/cerrado. Se pueden introducir nuevas variantes de productos sin descomponer el código cliente existente.

Contras:

- Puede ser que el código se complique demás, ya que con el patrón se suman muchas clases nuevas.

Builder

Propósito

Nos permite construir objetos complejos paso a paso. Brinda la posibilidad de producir distintos tipos y representaciones de un objeto utilizando el mismo código de construcción.

Problema

Imaginemos un objeto complejo que requiere una inicialización muy trabajosa, paso a paso, con muchos campos y objetos anidados. En general este código de inicialización está oculto dentro de un constructor gigante, con una gran cantidad de parámetros.


O peor: disperso por todo el código cliente.

Por ejemplo, pensemos en cómo crear un objeto Auto. Para construir un auto sencillo, debemos construir cuatro ruedas y un volante, así como instalar un chasis, volante, pedales, colocar puertas y ponerle un motor. Pero ¿qué pasa si queremos un auto más grande y sofisticado, con alarma y otros extras (como sistema de audio con pantallas, cierre centralizado y levantavidrios eléctrico)?

La solución más sencilla es extender la clase base Auto y crear un grupo de subclases que cubran todas las combinaciones posibles de parámetros. Pero en cualquier caso, terminaremos con una gran cantidad de subclases. Más aún: cualquier parámetro nuevo, como el tipo de alarma, requerirá que agrandemos todavía más esta jerarquía.

Ahora bien: existe otra posibilidad que no implica generar subclases. Se puede crear un constructor enorme dentro de la clase base Auto, con todos los parámetros posibles para controlar el objeto casa. Aunque es cierto que esta solución elimina el tema de las subclases, genera otro problema: no todos los parámetros son necesarios todo el tiempo.

En la mayoría de los casos gran parte de los parámetros no se utilizarán, lo cual provocará que las llamadas al constructor sean... bastante feas.



Por ejemplo: solo una pequeña parte de los autos tienen sistema de audio con pantallas, por lo que los parámetros relacionados con sistema de audio con pantallas, serán inútiles la mayor parte de las veces.

Solución

Builder sugiere que quitemos el código de construcción del objeto de su propia clase y lo coloquemos en objetos independientes, llamados constructores.

El patrón Builder permite construir objetos complejos paso a paso. Además, no permite a otros objetos acceder al producto mientras se construye.

Builder organiza la construcción de objetos en pasos (`construirPuertas`, `construirMotor`, etc.). Para crear un objeto, se ejecutan estos pasos en un objeto constructor. Lo importante, es que no se requiere invocar todos los pasos. Se pueden invocar sólo aquellos que sean necesarios para generar una configuración puntual y específica de un objeto.

Puede suceder que algunos pasos de la construcción requieran de una implementación diferente cuando tengamos que construir distintas representaciones del producto. Por ejemplo, el combustible de un taxi puede ser gasoil, pero el combustible de un auto de carrera deben ser de altísimo octanaje y diseñado específicamente.

En este caso, podemos crear distintas clases constructoras que implementen el mismo conjunto de pasos de construcción, pero de otra forma. Entonces podemos utilizar estos constructores en el proceso de construcción (por ejemplo, una serie ordenada de llamadas a los pasos de construcción) para producir distintos tipos de objetos.

Los distintos constructores ejecutan la misma tarea de formas distintas.

Por ejemplo, imaginemos un constructor que construye todo utilizando fibra de carbono, materiales muy livianos y motores de altísimas revoluciones, otro que construye usando materiales reciclados y motores eléctricos y un tercero que utiliza materiales por encima del promedio (como cuero y componentes de alta tecnología). Al invocar la misma serie de pasos, obtenemos un auto de carreras del primer constructor, un auto 100% ecológico del segundo y un auto de alta gama del tercero. Aún así, esto funcionaría sólo si el código cliente tiene a su disposición una interfaz común para invocar los pasos de construcción.

Clase directora

Se puede ir más lejos y crear una clase directora: una en la que se encuentren una serie de llamadas a los pasos del constructor que utilizamos para construir un producto determinado. La clase directora define el orden en el que se deben ejecutar los pasos de construcción, mientras que el constructor proporciona la implementación de dichos pasos.

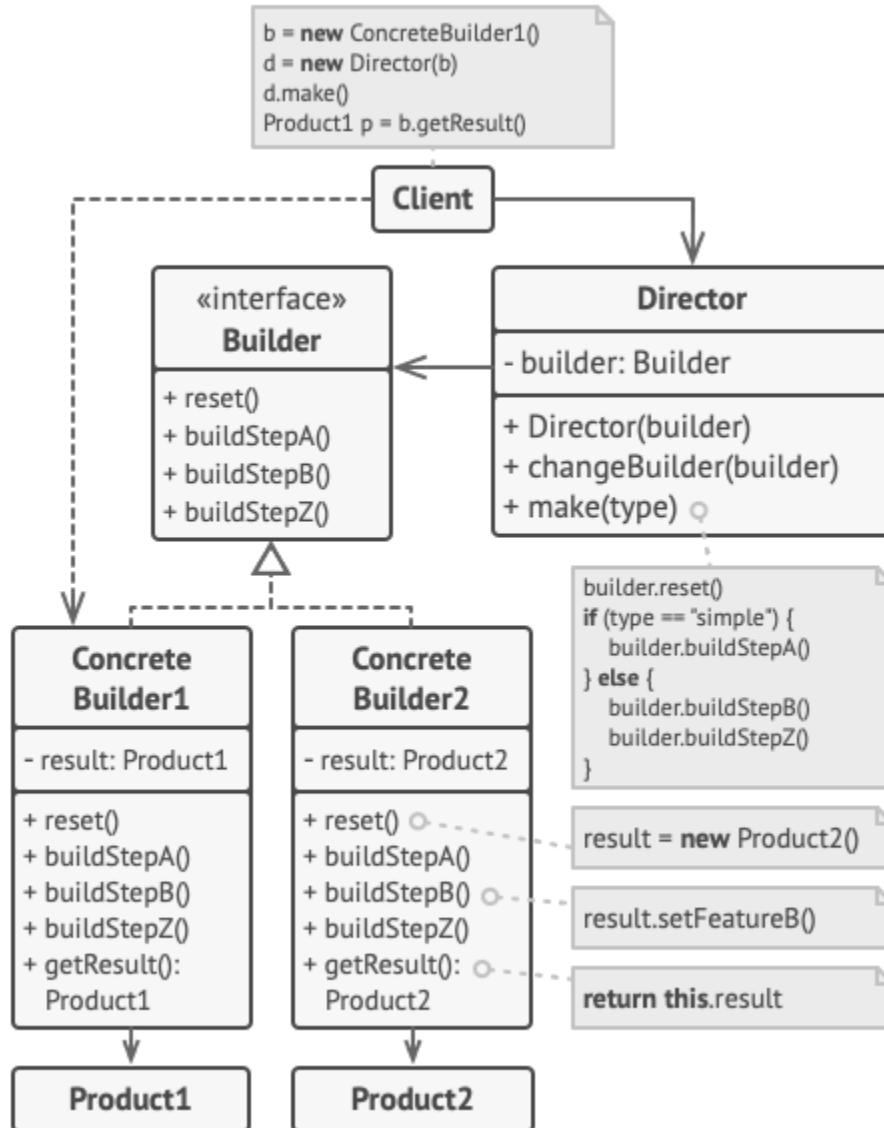
A su vez, sabe qué pasos ejecutar para lograr un producto que funcione.

No es 100% necesario tener una clase directora, ya que se pueden invocar los pasos de construcción en un orden específico directamente desde el código cliente. Sin embargo, la clase directora puede ser un buen lugar donde colocar distintas secuencias de construcción, a fin de poder reutilizarlas a lo largo del programa.

Asimismo, la clase directora esconde por completo los detalles de la construcción del producto al código cliente. El cliente sólo necesita asociar un objeto constructor con una clase directora, utilizarla para iniciar la construcción, y obtener el resultado.


Estructura

1. La interfaz Constructora declara determinados pasos de construcción, que todos los tipos de objetos constructores tendrán en común.
2. Los Constructores Concretos ofrecerán diferentes implementaciones de los pasos de construcción. Además, tienen la capacidad de crear productos que no siguen la interfaz común.
3. Los objetos resultantes, son los Productos. De existir productos contruidos por distintos objetos constructores, no requieren pertenecer a la misma jerarquía de clases o interfaz.
4. La clase Directora define el orden en el que se invocarán los pasos de construcción, por lo que es posible crear y reutilizar configuraciones específicas de los productos.
5. El Cliente debe asociar uno de los objetos constructores con la clase directora. Normalmente, se hace una sola vez con la clase directora (a través de los parámetros del constructor), que utiliza el objeto constructor para el resto de la construcción. Sin embargo, existe otra solución para cuando el cliente pasa el objeto constructor al método de producción de la clase directora. En este caso, se puede utilizar un constructor diferente cada vez que se produzca algo con la clase directora.



Aplicabilidad

- **Utilizamos Builder para evitar un “constructor telescópico”:** supongamos que tenemos un constructor con diez parámetros (todos opcionales). Invocar a un constructor de ese tamaño es muy poco práctico, por lo que en lugar de hacer eso podemos dotar a la clase de una serie de métodos para cargar los datos que necesitamos, mientras el resto viene con valores por defecto.
- **También lo utilizamos cuando queremos que el código sea capaz de crear distintas representaciones de ciertos productos (por ejemplo, termos de aluminio o de plástico):** se puede aplicar cuando la construcción de distintas representaciones de un mismo



producto requiera de pasos similares pero que varían en los detalles. La interfaz constructora base define todos los pasos de construcción posibles, mientras que los constructores concretos implementan esos pasos para construir representaciones particulares del producto. Mientras tanto, la clase directora va guiando el orden de la construcción.

- **Builder se puede usar para construir árboles con el patrón Composite u otros objetos complejos:** dado que Builder permite construir productos paso a paso, se podría aplazar la ejecución de ciertos pasos sin descomponer el producto final. Incluso es posible invocar pasos de forma recursiva, lo cual es útil cuando hace falta construir un árbol de objetos. Así, el constructor no expone el producto incompleto mientras ejecuta los pasos de construcción. Esto evita que el código cliente obtenga un resultado incompleto.


Cómo implementarlo

1. Asegurarse de poder definir claramente los pasos comunes de construcción para todas las representaciones disponibles del producto. De lo contrario, no será posible implementar el patrón.
2. Declarar estos pasos en la interfaz constructora base.
3. Crear una clase constructora concreta para cada una de las representaciones de producto e implementa sus pasos de construcción.
4. Recordar que se debe implementar un método para extraer el resultado de la construcción.
5. Analizar si es necesario crear una clase directora. Ésta puede encapsular formas diferentes de construir un producto utilizando el mismo objeto constructor.
6. El código cliente crea tanto el objeto constructor como el director. Antes de que empiece la construcción, el cliente debe pasar el objeto constructor al objeto director. En general, el cliente hace esto sólo una vez mediante los parámetros del constructor del director. El director utiliza el objeto constructor para el resto de la construcción.
7. El resultado de la construcción solo se puede obtener directamente del director si todos los productos siguen la misma interfaz. De lo contrario, el cliente deberá extraer el resultado del objeto constructor.

Pros y contras

Pros:

- Se pueden construir objetos paso a paso, aplazando pasos de la construcción o ejecutar pasos de forma recursiva.
- Es posible reutilizar el mismo código de construcción para construir distintas representaciones de productos.

- 
- Principio de responsabilidad única: se puede separar un código de construcción complejo de la lógica de negocio del producto.

Contras:

- La complejidad general del código aumenta, ya que Builder implica la creación de varias clases nuevas.

Singleton

Propósito

Patrón de diseño creacional que permite asegurarse de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia.

Problema

Singleton resuelve dos problemas al mismo tiempo (lo cual, si somos estrictos, rompe el Principio de responsabilidad única):

1. **Garantizar que una clase tenga una única instancia.** ¿Por qué querría alguien controlar cuántas instancias tiene una clase? El motivo más común es controlar el acceso a algún recurso compartido. Por ejemplo, una base de datos o un archivo.
Funciona así: imaginemos que creamos un objeto y luego de un tiempo decidimos crear otro nuevo. En lugar de recibir un objeto nuevo, obtendremos el que ya habíamos creado. Tengamos en cuenta que este comportamiento no es posible de implementar con un constructor convencional, ya que una llamada al constructor por diseño retornará un nuevo objeto.
2. **Proporcionar un punto de acceso global a dicha instancia.** ¿Te acordás de esas variables globales que usaste para almacenar objetos esenciales? Si bien es cierto que son útiles, también son poco seguras: cualquier código podría sobrescribir el contenido de esas variables y hacer fallar a la aplicación.
Al igual que una variable global, Singleton nos permite acceder a un objeto desde cualquier parte del programa. ¿Entonces cuál es la diferencia? Que además, evita que otro código sobreescriba esa instancia.
Este problema tiene otra cara: no queremos que el código que resuelve el primer problema esté disperso por toda la aplicación. Es mucho más conveniente tenerlo dentro de una clase, sobre todo si el resto del código ya depende de ella.

Solución

Todas las implementaciones de Singleton tienen estos dos pasos en común:

Las implementaciones de Singleton en general cuentan con estos dos pasos:

1. Hacer privado el constructor para evitar que otros objetos instancien la clase Singleton.
2. Crear un método estático que actúe como constructor. Por detrás, este método invoca al constructor privado para crear un objeto y lo guarda en un atributo estático. Las siguientes llamadas a este método devuelven el objeto almacenado en caché.

Mientras el código cliente tenga acceso a la clase Singleton, podrá invocar su método estático. De esta manera, cada vez que se invoque este método, siempre se devolverá el mismo objeto.

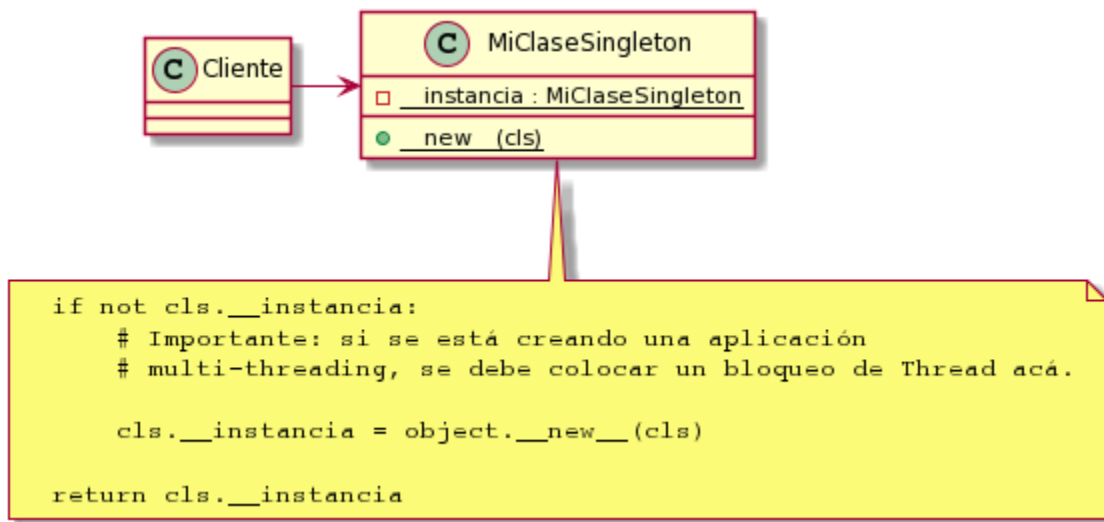
Ahora bien, esta descripción tiene un único problema: no es Python. En Python no existen el concepto de "constructor privado". Todas las clases cuentan con constructor, esté declarado o no.

En el caso de Python, debemos utilizar el método mágico `__new__`, el cual nos permite intervenir en el proceso de construcción de los objetos. Allí podremos analizar si existe una instancia ya creada (almacenada por ejemplo, en un atributo de clase privado). De no existir, crearemos una con `object.__new__(cls)` y la almacenaremos en el atributo privado mencionado anteriormente.

Analogía en el mundo real

El gobierno de un país es un buen ejemplo de Singleton. Un país sólo puede tener un gobierno oficial. Más allá de las identidades personales de los individuos que forman el gobierno, el título oficial de "Gobierno de X" es un único punto de acceso que identifica al grupo de personas a cargo.

Estructura



La clase Singleton declara el método de clase `__new__` que devuelve la misma instancia de su propia clase.

El constructor del Singleton no se encontrará oculto del código cliente, sino que se utilizará de forma normal. La única diferencia será que intervendremos el proceso de creación del objeto.

Aplicabilidad

- **Utilizamos Singleton cuando una clase de nuestro programa debe tener solo una instancia disponible para todos los clientes. Por ejemplo, un único objeto de conexión con la base de datos, compartido por todo el programa:** aquí, Singleton deshabilitará el resto de las maneras de crear objetos de una clase. O se creará un nuevo objeto, o se retornará uno existente si ya fue creado.
- **Este patrón también se utiliza cuando se requiere un control más estricto de las variables globales:** al contrario de estas, Singleton garantiza que exista una única instancia de la clase. Se encuentra pensado para que nada pueda sustituir la instancia en caché.

Importante: se debe tener en cuenta que siempre se podrá ajustar esta limitación y permitir la existencia de un número determinado de instancias Singleton. La única parte del código que requiere cambios es el cuerpo del método `__new__`.

Cómo implementarlo

1. Agregar un atributo estático privado a la clase para almacenar la instancia Singleton.
2. Declarar un método de creación estático público (`__new__`) para obtener la instancia Singleton.
3. Implementar una inicialización diferida dentro del método estático. Este debe crear un nuevo objeto en su primera llamada y colocarlo dentro del atributo estático. El método deberá devolver esa instancia en todas las llamadas siguientes.
4. Declara el constructor de clase como se requiera. El método estático de la clase seguirá siendo capaz de construir el objeto, y será transparente a los otros objetos.


Pros y contras

Pros:

- Se puede tener la seguridad de que la clase tiene una única instancia.
- Se obtiene un único punto de acceso a dicha instancia.
- El Singleton solo se inicializa realmente cuando se requiere la primera vez.

Contras:

- Vulnera el Principio de responsabilidad única. Resuelve dos problemas al mismo tiempo.
- Singleton puede ocultar un mal diseño. Por ejemplo, cuando los componentes del programa saben demasiado los unos sobre los otros (lo cual en rigor vulneraría el Principio de Inversión de Dependencia, que indica que las abstracciones no deberían



depender de las implementaciones, sino que los detalles deberían depender de las abstracciones).

- En entornos con múltiples hilos de ejecución, requiere de un tratamiento especial para que varios hilos no creen un objeto Singleton varias veces.
- Puede resultar engorroso realizar pruebas unitarias del código cliente del Singleton, ya que muchos frameworks de prueba dependen de la herencia a la hora de crear objetos simulados (mock objects). Debido a que la clase Singleton es privada y en la mayoría de los lenguajes resulta imposible sobrescribir métodos estáticos, habrá que pensar en una manera original de simular el Singleton. O, simplemente, no escribir las pruebas. O bien, no utilizar Singleton.

Patrones estructurales

Adapter

Propósito

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces originalmente incompatibles.

Problema

Imaginemos que estamos creando una aplicación de monitoreo de la evolución del dólar. La aplicación descarga la información del dólar desde varias fuentes en formato XML para presentarla al usuario con bonitos gráficos y diagramas.

En un momento dado, decidimos mejorar la aplicación integrando una biblioteca de construcción de gráficos de un tercero. Pero hay un problema: esta biblioteca solo funciona con datos en formato JSON.

No se puede utilizar la biblioteca de gráficos tal como está porque espera los datos en un formato incompatible con la aplicación.

¿Se podría cambiar la biblioteca para que funcione con XML? Sí, sin embargo, esto podría romper el código existente que depende de la biblioteca. Incluso podría no tener siquiera acceso al código de la biblioteca, lo que hace imposible dicha solución.

Solución

Se puede crear un adaptador: un objeto especial que *adapta* la interfaz de un objeto, de forma que otro objeto pueda comprenderla.

Un adaptador envuelve uno de los objetos para ocultar la complejidad de la conversión que existe por detrás. El objeto envuelto no se entera de la existencia del adaptador. Por ejemplo, se puede envolver un objeto que opera con gramos y kilogramos con un adaptador que convierte todos los datos a onzas y libras.

Los adaptadores no solo convierten datos a distintos formatos, sino que también ayudan a colaborar a objetos con interfaces diferentes. Funciona así:

1. El adaptador obtiene una interfaz compatible con uno de los objetos existentes.
2. Con esta interfaz, el objeto existente puede invocar los métodos del adaptador sin problemas.

3. Al recibir una llamada, el adaptador pasa la solicitud al segundo objeto, pero en un formato y orden que ese segundo objeto entiende.

En algunos casos se puede incluso crear un adaptador bi-direccional que pueda convertir las llamadas en ambos sentidos.

Regresando a nuestra aplicación de la evolución del dólar: para resolver el dilema de los formatos incompatibles, es posible crear adaptadores de XML a JSON para cada clase de la biblioteca de gráficos con la que trabaje el código directamente. Luego, ajustar el código para que se comunique con la biblioteca específicamente a través de estos adaptadores. Cuando un adaptador recibe una llamada, traduce el XML entrante a una estructura JSON y pasa la llamada a los métodos correspondientes de un objeto de graficación envuelto.

Analogía en el mundo real

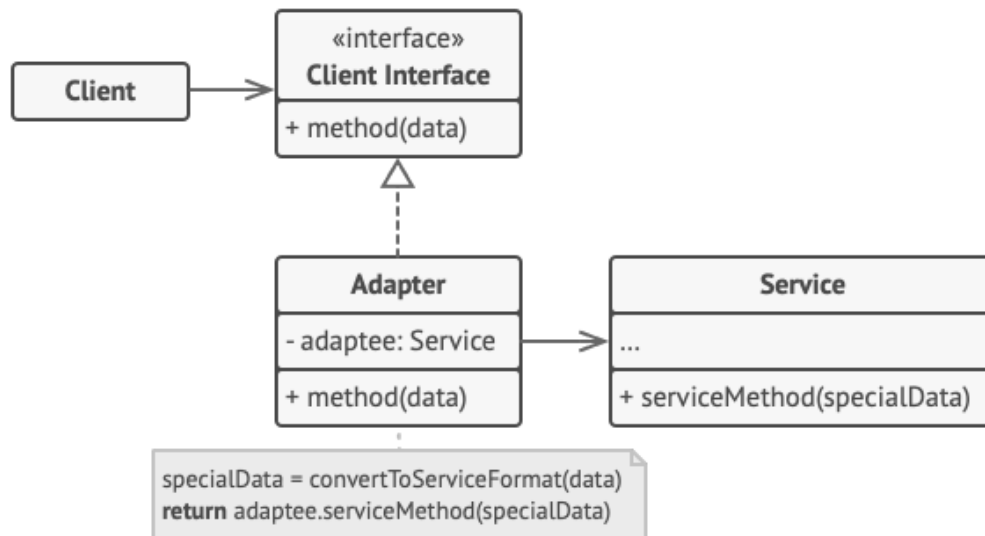
Cuando viajas de Argentina a Europa por primera vez, puede que sea un problema cargar la batería de la laptop. Los tipos de enchufe son diferentes en cada país, por lo que un enchufe de Argentina no sirve en Europa. El problema puede solucionarse utilizando un adaptador que incluya el enchufe argentino y el europeo.

Estructura

Esta implementación utiliza composición de objetos: el adaptador implementa la interfaz de un objeto y envuelve al otro. Puede implementarse en todos los lenguajes de programación populares.

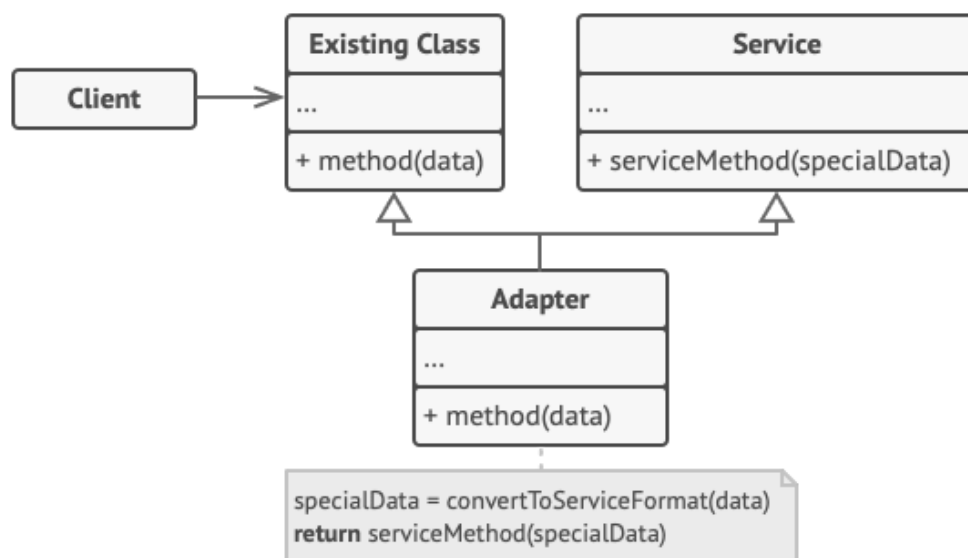
1. La clase Cliente contiene la lógica de negocio existente.
2. La Interfaz con el Cliente describe un modo de comunicarse, que otras clases deben seguir para poder colaborar con el código cliente.
3. Servicio es alguna clase útil (normalmente de una tercera parte o heredada). El código cliente no puede utilizar directamente esta clase porque sus interfaces son incompatibles.
4. La clase Adaptadora puede trabajar tanto con la clase cliente como con la clase de servicio: implementa la interfaz con el cliente y "envuelve" el objeto de la clase de servicio. A su vez, traduce las llamadas que recibe del cliente (a través de la interfaz adaptadora) en llamadas al objeto envuelto, en un formato que este último pueda comprender.
5. El código cliente queda desacoplado de la clase adaptadora concreta siempre y cuando funcione con la clase adaptadora a través de la interfaz con el cliente. Gracias a esto, es posible introducir nuevos tipos de adaptadores en el programa sin romper el código

existente. Esto puede resultar de utilidad si la interfaz de la clase de servicio se cambia, ya que se puede crear una nueva clase adaptadora sin cambiar el código cliente.



Clase adaptadora

Esta implementación utiliza herencia, porque la clase adaptadora hereda interfaces de ambos objetos al mismo tiempo. Es importante tener en cuenta que esta opción sólo puede implementarse en lenguajes de programación que tengan soporte para herencia múltiple, como el caso de Python o de C++.



Aplicabilidad

- **Utilizar Adapter cuando se requiera usar una clase existente, pero cuya interfaz no sea compatible con el resto del código:** este patrón permite crear una clase intermedia que sirva como traductora entre nuestro código y una clase heredada, de terceros o con una interfaz extraña.
- **También es viable su uso cuando se necesite reutilizar subclases existentes que carezcan de alguna funcionalidad común que no pueda sumarse a la superclase:** es posible extender cada subclase y colocar la funcionalidad que falta, dentro de las nuevas clases hijas. Sin embargo, se deberá duplicar el código en todas estas nuevas clases, lo cual no huele nada bien.

Una solución más elegante sería colocar la funcionalidad faltante dentro de un Adapter. Con ella se pueden "envolver" objetos a los que les falten funciones, obteniendo esas funciones necesarias dinámicamente. Para que funcione, dichas clases deben tener una interfaz común y la clase adaptadora debe seguir dicha interfaz. Este procedimiento es muy similar a otro patrón: Decorator.

Cómo implementarlo

1. Asegurarse de que existen al menos dos clases con interfaces incompatibles:
 - Una clase "servicio" que no se puede modificar (en general de un tercero, heredada o con muchas dependencias existentes).
 - Una o varias clases cliente que se beneficiarían de contar con una clase de servicio.
2. Declarar la interfaz con el cliente y describir el modo en que las clases cliente se comunican con la clase de servicio.
3. Crear la clase adaptadora, que siga la interfaz con el cliente. Dejar todos los métodos vacíos en principio.
4. Agregar un atributo a la clase adaptadora para almacenar una referencia al objeto de servicio. Lo común es inicializar este atributo a través del constructor, aunque a veces sirve pasarlo al adaptador cuando se invocan sus métodos.
5. Implementar en la clase adaptadora todos los métodos de la interfaz con el cliente. Esta clase adaptadora deberá delegar la mayoría del trabajo real al objeto de servicio, gestionando solo la interfaz o la conversión de formato de los datos.
6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente. Esto brindará la posibilidad de cambiar o extender las clases adaptadoras sin afectar al código cliente.

Pros y contras

Pros

- *Principio de responsabilidad única.* Es posible separar la interfaz o el código de conversión de datos, de la lógica de negocio del programa.
- *Principio Abierto/Cerrado.* Se pueden introducir nuevos tipos de adaptadores al programa sin necesidad de descomponer el código cliente existente. Esto, siempre y cuando trabajen con los adaptadores a través de la interfaz con el cliente.

Contras

- La complejidad general del código aumenta. Esto debido a que se requiere introducir un grupo de nuevas interfaces y clases. Algunas veces resulta más simple cambiar la clase de servicio de modo que coincida con el resto del código.

Decorator

Propósito

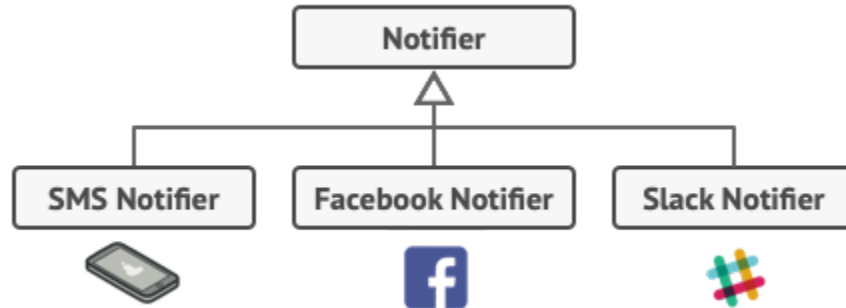
Es un patrón de diseño estructural que permite sumar funcionalidades a objetos, colocandolos dentro de objetos encapsuladores que contienen estas funcionalidades.

Problema

Imaginemos que trabajamos en una biblioteca de notificaciones. La misma permite a otros programas notificar a sus usuarios sobre eventos importantes.

La primera versión de la biblioteca se basa en la clase Notificador. Esta solo cuenta con algunos atributos, un constructor y un método send. El método puede aceptar un parámetro de mensaje de un cliente y enviar el mensaje a una lista de e-mails pasados a la clase notificadora a través de su constructor. Una aplicación de terceros que actuaba como cliente requería crear y configurar el objeto notificador una vez y después utilizarlo cada vez que sucediera algo importante.

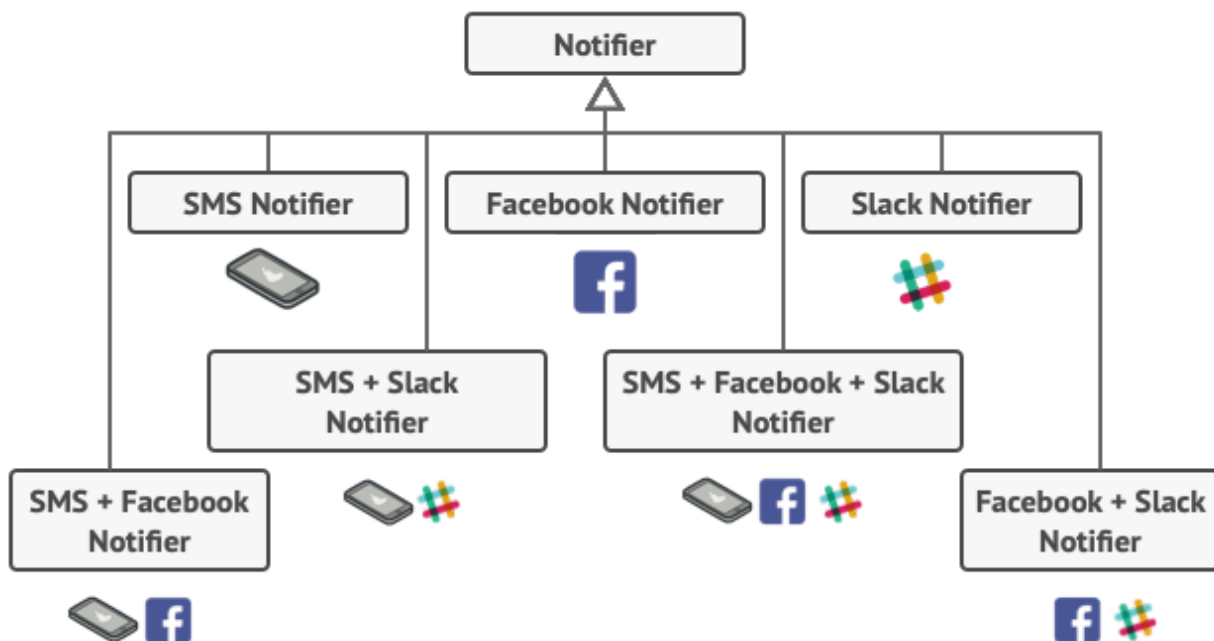
En un momento dado, notamos que los usuarios de la biblioteca esperan más que unas simples notificaciones por correo. A muchos les gustaría recibir mensajes SMS sobre asuntos importantes. Por otro lado, otros usuarios querrían recibir las notificaciones por Facebook y por último: a los usuarios corporativos les encantaría recibir notificaciones por Slack.



No debería ser muy complejo, ¿cierto? Extendimos la clase Notificador y colocamos los métodos adicionales de notificación dentro de nuevas subclases. Ahora el cliente debería instanciar la clase notificadora correcta y utilizarla para el resto de notificaciones.

Pero entonces, alguien nos hace una pregunta: "¿Por qué no utilizar distintos tipos de notificación al mismo tiempo? Si tu casa hay un incendio, probablemente quieras que te informen por todas las vías posibles".

Intentamos solucionar este problema creando subclases especiales que combinaban varios métodos de notificación dentro de una clase. Ahora bien, muy pronto se hizo evidente que esta solución "inflaría" mucho el código, tanto el de la biblioteca como el del cliente.



Entonces: necesitamos encontrar alguna otra forma de estructurar las clases de las notificaciones.

Solución

Cuando necesitamos alterar la funcionalidad de un objeto, en general lo primero que se viene a la mente es extender una clase. Sin embargo, es importante destacar que la herencia tiene varias limitaciones importantes de las que debemos ser conscientes.

Por ejemplo:

- La herencia es estática. No es posible alterar la funcionalidad de un objeto existente durante tiempo de ejecución. Sólo se puede sustituir el objeto completo por otro, creado a partir de una subclase diferente.
- Las subclases sólo pueden tener una clase padre. En la mayoría de lenguajes, la herencia no permite a una clase heredar comportamientos de varias clases al mismo tiempo (esto, por supuesto, no es así en Python).

Una de las formas de superar estas limitaciones es empleando Agregación o Composición en lugar de Herencia. Ambas alternativas funcionan prácticamente del mismo modo: un objeto tiene una referencia a otro y le delega parte del trabajo. Por otro lado, con la herencia el propio objeto puede realizar ese trabajo, heredando el comportamiento de la superclase.

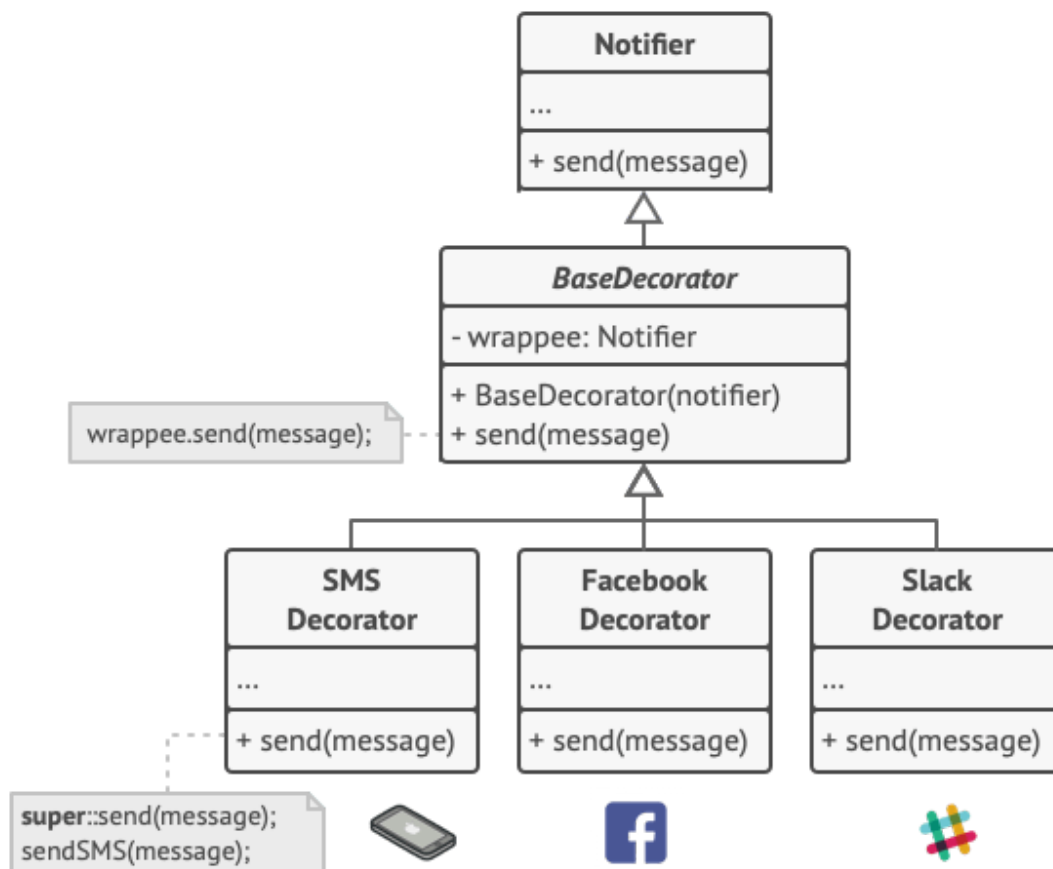
Con esta nueva solución es posible sustituir fácilmente el objeto "ayudante" por otro, cambiando el comportamiento del contenedor en tiempo de ejecución. Un objeto puede utilizar el comportamiento de varias clases con referencias a varios objetos, delegándoles todo tipo de tareas. Tanto la agregación como la composición son principios claves que se utilizan en muchos patrones de diseño, incluyendo el Decorator.



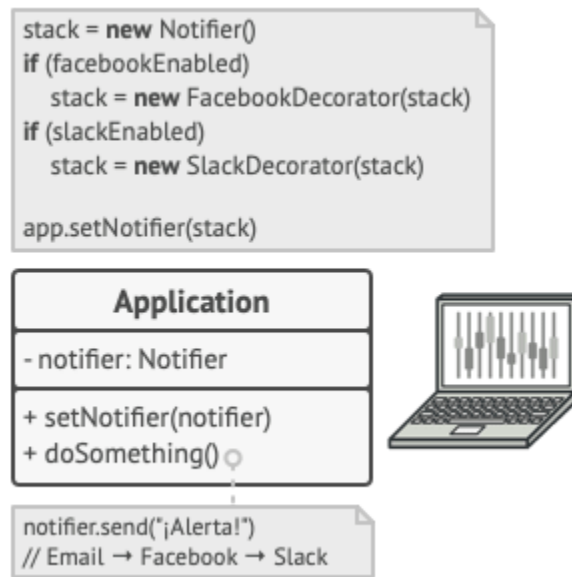
"Wrapper" (envoltorio, en inglés) es el sobrenombre alternativo de este patrón, que expresa con claridad su idea principal. Un wrapper es un objeto que puede vincularse con otro objeto objetivo. El wrapper contiene el mismo grupo de métodos que el objetivo y le delega todas las solicitudes que recibe. Sin embargo, el wrapper puede alterar el resultado haciendo alguna otra cosa, sea antes o después de pasar la solicitud al objetivo.

¿Cuándo se convierte un simple wrapper en decorator? Como mencionamos, el wrapper implementa la misma interfaz que el objeto envuelto. Es por esto que desde el punto de vista del cliente, estos objetos son idénticos. Lograr que el campo de referencia del wrapper acepte cualquier objeto que siga esa interfaz permitirá envolver un mismo objeto en varios wrappers distintos, sumándole el comportamiento combinado de todos ellos.

En el ejemplo de las notificaciones, dejemos la funcionalidad de las notificaciones por correo electrónico dentro de la clase base Notificador, pero convirtamos el resto de los métodos de notificación en decoradores.



El código cliente debe envolver un objeto notificador básico dentro de un grupo de decoradores que cumplan con las preferencias del cliente. Los objetos resultantes se estructurarán como una pila.



El código cliente trabajará con el último decorador de la pila. Debido a que todos los decoradores poseen la misma interfaz que la clase base notificadora, al código cliente no le importa si está trabajando con el objeto notificador "puro" o con el decorado.

Más aún: es posible aplicar la misma solución a otras funcionalidades, como el formateo de mensajes o la composición de una lista de destinatarios. El cliente puede decorar el objeto con los decoradores personalizados que requiera, en tanto y en cuanto tengan la misma interfaz.

Analogía en el mundo real

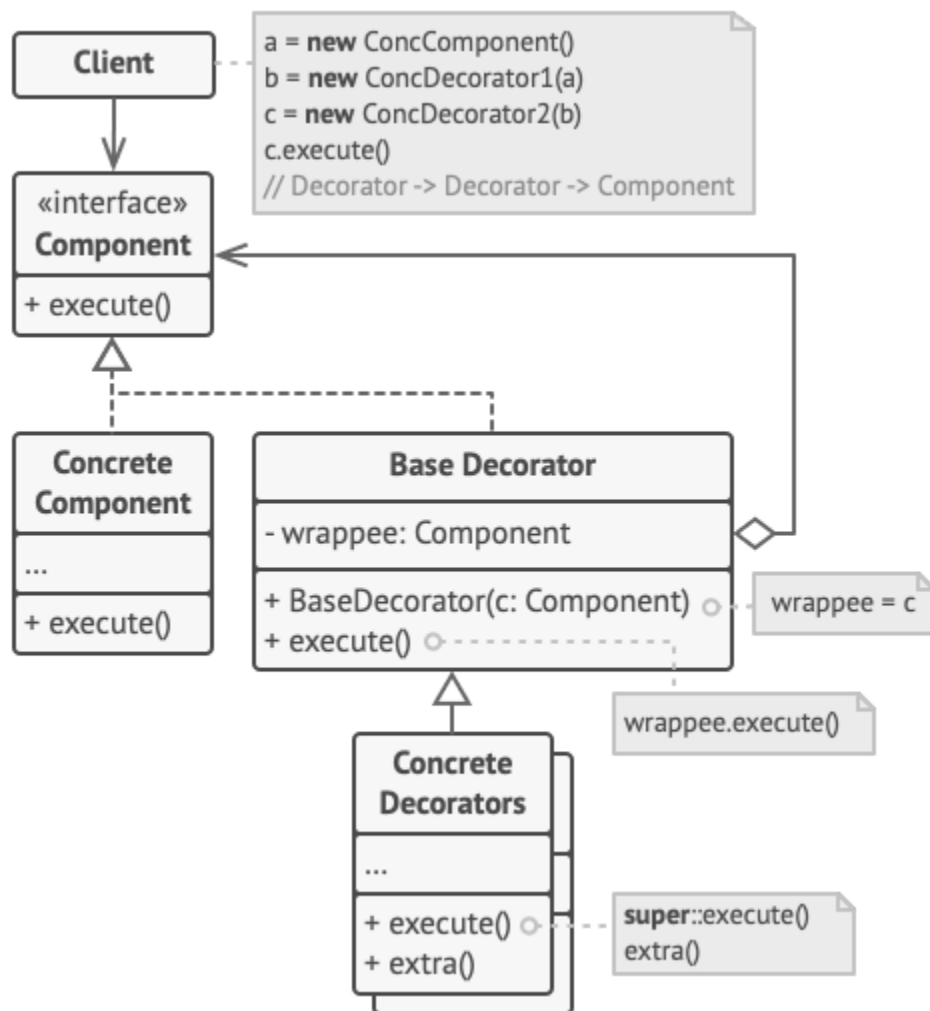
Un excelente ejemplo del uso de decoradores, es vestir ropa. Cuando tenemos frío, nos "cubrimos" con un suéter. Si seguimos teniendo frío a pesar del suéter, podemos sumar una campera. Y si está lloviendo, podemos ponernos un piloto. Todas estas prendas "extienden" nuestro comportamiento básico pero no son parte de nosotros. A su vez, podemos quitarnos fácilmente cualquier prenda si así lo deseamos.

Estructura

1. El Componente declara la interfaz común, tanto para wrappers como para objetos envueltos.
2. Componente Concreto es una clase de objetos envueltos. El mismo define un comportamiento básico, que los decoradores pueden modificar.
3. La clase Decoradora Base tiene un campo para hacer referencia un objeto envuelto (o decorado). En lenguajes estáticamente tipados, el tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos


como los decoradores. En los de tipado dinámico, esto no es necesario (aunque es posible). La clase decoradora base delega todas las operaciones al objeto envuelto.

4. Los Decoradores Concretos definen funcionalidades adicionales, las cuales se pueden añadir dinámicamente a los componentes. A su vez, sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.
5. El Cliente puede envolver componentes en varias capas de decoradores, en tanto y en cuanto trabajen con los objetos a través de la interfaz del componente.



Aplicabilidad

- **Utilizamos Decorator cuando necesitamos sumar funcionalidades adicionales a objetos durante tiempo de ejecución, sin descomponer el código que utiliza esos objetos.** Decorator permite estructurar nuestra lógica de negocio en capas, creando un



decorador para cada capa y componer objetos con varias combinaciones de esta lógica. Todo, en tiempo de ejecución. El código cliente puede tratar a todos estos objetos de la misma forma, ya que todos siguen una interfaz común.

- **Utilizamos este patrón cuando no sea posible extender el comportamiento de un objeto utilizando herencia, o bien resulte "extraño" (es decir, que nos quede código acoplado, o que rompamos el Principio de Sustitución de Liskov).** Muchos lenguajes de programación cuentan con la palabra clave final. Esta puede utilizarse para evitar que una clase siga extendiéndose. Para una clase final, la única forma de reutilizar el comportamiento existente será envolver la clase con tu propio wrapper, utilizando Decorator.

Cómo implementarlo

1. Asegurarse de que el dominio de negocio puede representarse como un componente principal, con varias capas opcionales encima.
2. Decidir qué métodos son comunes al componente principal y las capas opcionales. Crear una interfaz de componente y declarar esos métodos en la misma.
3. Crear una clase concreta de componente y definir ahí el comportamiento base.
4. Crear una clase base decoradora. Debe tener un campo para almacenar una referencia al objeto “envuelto”. Este campo debería declararse con el tipo de interfaz de componente para permitirle interactuar tanto con componentes concretos, como con decoradores. La clase decoradora base debe delegar todas las operaciones al objeto envuelto.
5. Asegurarse de que todas las clases implementan la interfaz de componente.
6. Crear decoradores concretos, heredando de la clase decoradora base. Un decorador concreto debe ejecutar sus comportamientos antes o después de la llamada al método correspondiente (que por supuesto, siempre delega al objeto envuelto).
7. El código cliente debe ser responsable de crear decoradores y luego componerlos, del modo que necesite.

Pros y contras

Pros

- Es posible extender el comportamiento de un objeto sin necesidad de crear otra subclase.
- Se puede agregar o quitar responsabilidades de un objeto **en tiempo de ejecución**.
- Se pueden combinar distintos comportamientos envolviendo un mismo objeto con varios decoradores.
- **Principio de responsabilidad única:** es posible dividir una clase que implementa muchas variantes posibles de comportamiento, en varias clases más pequeñas.

Contras

- Puede resultar difícil eliminar un decorator específico de la pila de decorators.
- Es complejo (pero no imposible) implementar un decorator de forma tal que su comportamiento no dependa del orden en la pila de decorators.
- El código de configuración inicial de las distintas capas puede tener un aspecto poco feliz.

Composite

Propósito

Es un patrón de diseño estructural que nos permite componer objetos en estructuras de árbol para luego trabajar con esas estructuras como si fueran objetos individuales.

Problema

El uso de Composite sólo tiene sentido cuando el modelo central de la aplicación puede representarse en forma de árbol.

Por ejemplo, imaginemos que tenemos dos tipos de objetos: Carpetas y Archivos. Una Carpeta puede contener varios Archivos así como cierto número de Carpetas. Estas Carpetas también pueden contener algunos Archivos o incluso otras Carpetas dentro, y así sucesivamente.


Digamos que decidimos crear un sistema de gestión de archivos que utiliza estas clases. Las carpetas pueden contener archivos sencillos y nada más, así como carpetas llenas de archivos... y otras carpetas. ¿Cómo determinamos el peso total de una carpeta?

Se puede intentar una solución directa: abrir todas las carpetas, repasar todos los archivos y calcular el peso total. Esto sería viable en el mundo real; pero en un programa no es tan lineal como recorrer un bucle. Se deben conocer de antemano las clases de Archivo y Carpetas a iterar, el nivel de anidación de las carpetas y otros detalles incómodos. Esto provoca que la solución directa sea muy complicada, o incluso imposible.

Solución

Composite sugiere trabajar con Archivos y Carpetas a través de una interfaz común que declara un método para calcular el peso total.

¿Cómo funcionaría este método? Para un archivo, sencillamente devuelve el peso del archivo. Para una carpeta, recorre cada archivo que contiene la caja, pregunta su peso y devuelve un total por la carpeta. Si uno de esos archivos fuera una carpeta, esa carpeta también comenzaría a



repasar su contenido y así sucesivamente, hasta que se calcule el peso de todos los componentes internos.

La gran ventaja de esta solución es que no hace falta preocuparse por las clases concretas de los objetos que componen el árbol. No es necesario saber si un objeto es un archivo simple o una sofisticada carpeta. Es posible tratarlos a todos por igual a través de la interfaz común. Cuando se invoca un método, los propios objetos pasan la solicitud a lo largo del árbol.

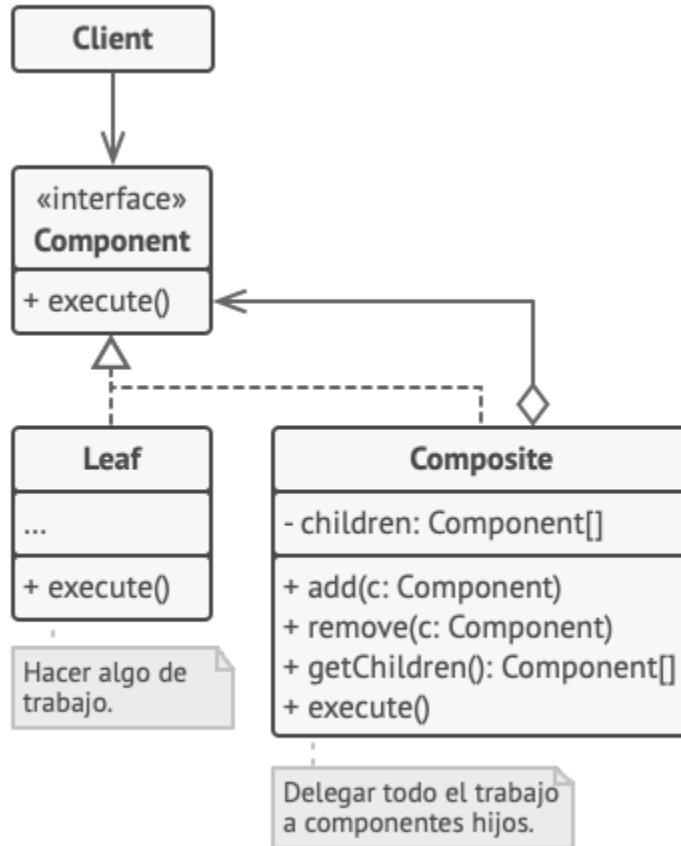
Analogía en el mundo real

Los ejércitos se estructuran en jerarquías. Un ejército está formado por divisiones; una división es un grupo de brigadas, las brigadas están formadas por pelotones, los cuales pueden dividirse en escuadrones.

Finalmente, un escuadrón es un grupo pequeño de soldados. Las órdenes se dan en la parte superior de la jerarquía y se pasan hacia abajo por cada nivel hasta que todos los soldados saben qué hay que hacer.

Estructura

1. Una interfaz **Componente** describe operaciones que son comunes a elementos simples y complejos del árbol.
2. La **Hoja** es un elemento básico de un árbol que no tiene subelementos. En general, los componentes de la hoja terminan realizando la mayor parte del trabajo real, ya que no tienen a nadie a quien delegarle el trabajo.
3. El **Contenedor** (también llamado compuesto) es un elemento que tiene subelementos: es decir, hojas u otros contenedores. Un contenedor no conoce las clases concretas de sus hijos. Éste funciona con todos los subelementos únicamente a través de la interfaz componente. Al recibir una solicitud, el contenedor delega el trabajo a todos sus subelementos, procesa los resultados intermedios y luego devuelve el resultado final al cliente.
4. El **Cliente** funciona con todos los elementos a través de la interfaz componente. Como resultado, el cliente puede funcionar de la misma manera tanto con elementos simples como complejos del árbol.




Aplicabilidad

- **Utilizamos Composite cuando sea necesario implementar una estructura de objetos con forma de árbol.** Composite proporciona dos tipos de elementos básicos que comparten una interfaz común: hojas simples y contenedores complejos. Un contenedor puede estar compuesto por hojas y por otros contenedores. Esto brinda la posibilidad de construir una estructura de objetos recursivos anidados, similar a un árbol.
- **También lo utilizamos cuando necesitamos que el código cliente trate elementos simples y complejos de la misma forma.** Todos los elementos definidos por Composite comparten la misma interfaz. Con esta interfaz, el cliente no necesita preocuparse por la clase concreta de los objetos con los que funciona.

Cómo implementarlo

1. Asegurarse de que el modelo de negocio (o al menos, una parte) pueda representarse como una estructura de árbol. Intentar dividirlo en elementos simples y contenedores. Recordar que los contenedores deben ser capaces de contener tanto elementos simples como otros contenedores.

- 
2. Declarar la interfaz componente con una lista de métodos que tengan sentido para componentes simples y complejos.
 3. Crear una clase “hoja” para representar los elementos simples. Un programa puede tener diferentes clases hoja.
 4. Crear una clase “contenedora” para representar elementos complejos. Incluir un campo en esta clase para almacenar las referencias a los subelementos. Este campo debe poder almacenar tanto hojas como contenedores, por lo que es importante asegurarse de declararla con el tipo de la interfaz componente. Al momento de implementar los métodos de la interfaz componente, recordar que el contenedor debe delegar **la mayor parte del trabajo** a los subelementos.
 5. Finalmente, definir los métodos para agregar y quitar elementos hijos dentro del contenedor. Aunque dichos métodos se pueden declarar en la interfaz componente, esto **violaría el Principio de segregación de interfaz**, dado que los métodos de la clase hoja estarían vacíos. Sin embargo, el cliente podrá tratar a todos los elementos de la misma manera, incluso al componer el árbol.

Pros y contras

Pros

- Es posible trabajar con estructuras de árbol complejas con facilidad: utilizamos polimorfismo y recursión en favor nuestro.
- **Principio abierto/cerrado:** se pueden introducir nuevos tipos de elementos en la aplicación sin descomponer el código ya existente, el cual funciona con el árbol de objetos.

Contras

- Puede llegar a resultar complejo brindar una interfaz común para clases cuya funcionalidad difiere demasiado. Algunas veces habrá que generalizar en exceso la interfaz componente, provocando que sea más difícil de comprender. A su vez, generalizar demasiado rompería el Principio de Segregación de Interfaz.

Patrones de comportamiento

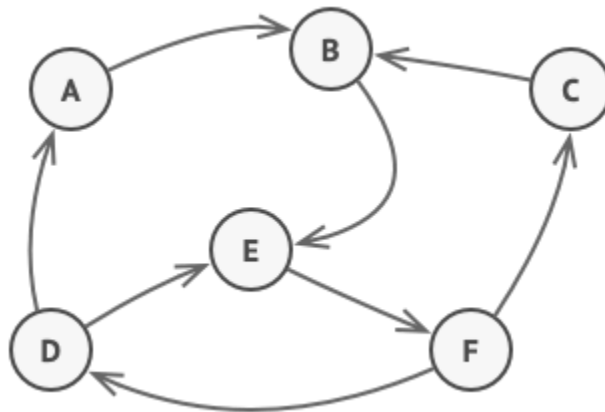
State

Propósito

State es un patrón de diseño de comportamiento que le da la posibilidad a un objeto de alterar su comportamiento cuando su estado interno se modifica. Hace parecer que el objeto cambiara su clase.

Problema

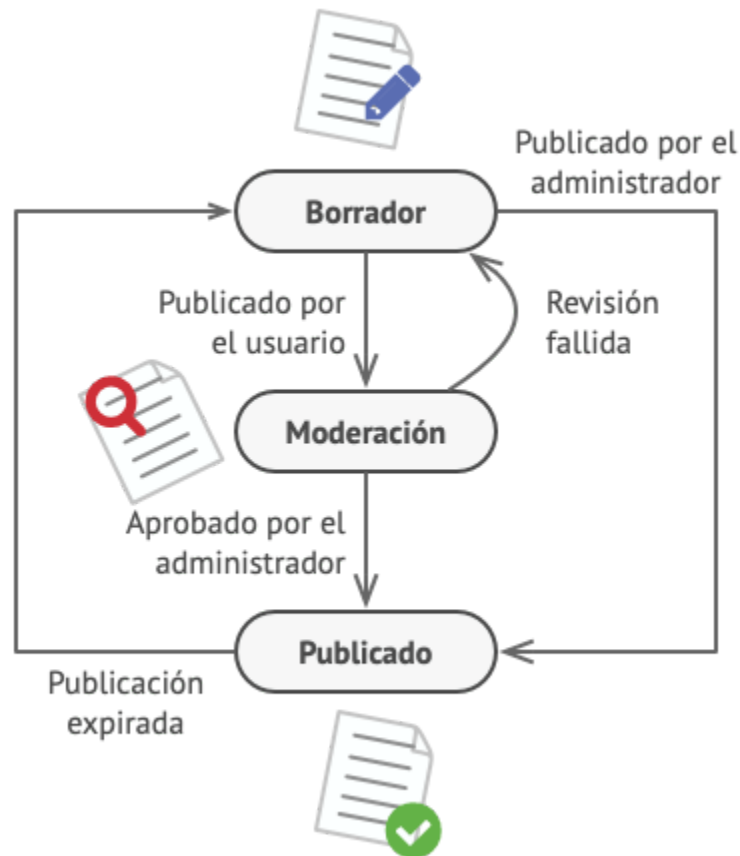
Este patrón está íntimamente relacionado al concepto de Máquina de estados finitos.



La idea fundamental es que en algún momento puntual, un programa puede encontrarse con un número **finito** de estados. En cada estado único, el programa se comporta de forma diferente y puede cambiar de un estado a otro instantáneamente. No obstante, el programa puede cambiar o no a otros estados, en base al estado actual. Estas transiciones también son finitas y predeterminadas.

También es posible aplicar esta solución a los objetos. Imaginemos que existe una clase `Documento`. Un documento puede encontrarse en alguno de estos estados: `Borrador`, `Moderación` y `Publicado`. El documento tiene un método `publicar`, que funciona de forma levemente distinta en cada estado:

- En `Borrador`, mueve el documento a moderación.
- En `Moderación`, hace público el documento, pero sólo si el usuario actual es un administrador.
- En `Publicado`, no hace nada en absoluto.



En general, las máquinas de estado se implementan con muchos condicionales (`if` o `switch`) que seleccionan el comportamiento adecuado en base al estado actual del objeto. Como sabemos, este “estado” es el grupo de valores de los atributos del objeto.

Más aún: es probable que hayas implementado un estado al menos una vez, aunque nunca hayas oído hablar de máquinas de estados finitos.

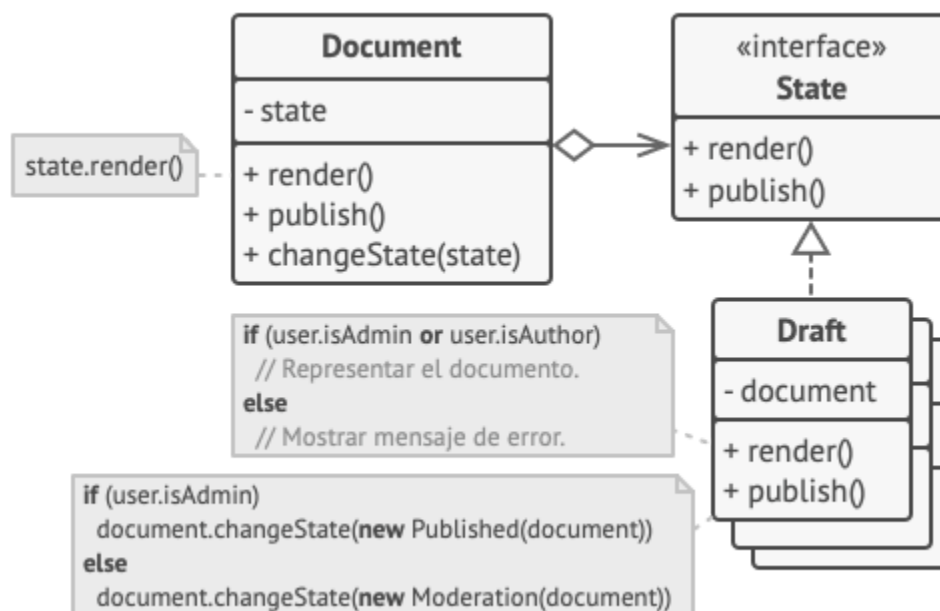
Generalmente, el punto débil de una máquina de estado basada en condicionales se manifiesta cuando comenzamos a sumar en la clase `Documento` tanto estados como comportamientos dependientes de estados. La mayor parte de los métodos tendrán condicionales horribles que eligen el comportamiento de un método en función del estado actual. Código así es muy difícil de mantener, básicamente porque cualquier cambio en la lógica de transición es muy probable que requiera modificar los condicionales de estado de cada método.

El problema empeora con la evolución del proyecto. Es bastante dificultoso predecir absolutamente todos los estados y transiciones posibles durante el diseño. Es por esto que una máquina de estados “liviana” (con una cantidad limitada de condicionales), con el tiempo puede llegar a crecer hasta convertirse en un desastre.

Solución

State sugiere crear clases nuevas para cada estado posible de un objeto y extraer cada comportamiento específico del estado para colocarlo dentro de estas clases.

En lugar de implementar todos los comportamientos, el objeto original (llamado contexto) almacena una referencia a uno de los objetos de estado que representa su estado actual y delega en él todo el trabajo relacionado con el estado.



Al momento de la transición del contexto a otro estado, simplemente sustituye el objeto de estado activo por otro que represente al nuevo estado. Por supuesto: esto sólo es posible si todas las clases de estado siguen la misma interfaz, y el contexto funciona con esos objetos a través de esa interfaz.

Si bien esta estructura puede llegar a resultar similar al patrón Strategy, hay una diferencia importante: mientras que las estrategias casi nunca se conocen, en State los estados pueden conocerse entre sí e iniciar transiciones de un estado a otro.

Analogía en el mundo real

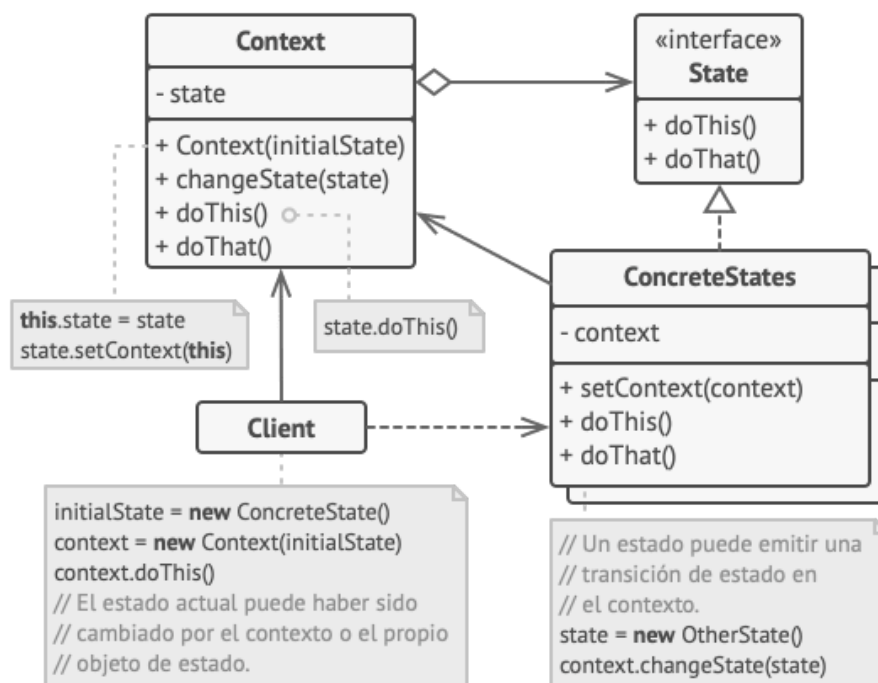
Los botones e interruptores de tu teléfono móvil se comportan de forma diferente dependiendo del estado del dispositivo. En general:

- Cuando el teléfono está desbloqueado, al presionar los botones se ejecutan funciones.
- Cuando el teléfono está bloqueado, presionar un botón desbloquea la pantalla.

- Cuando el teléfono necesita recargar la batería, pulsar un botón muestra la pantalla de batería baja.

Estructura

1. El Contexto (clase) guarda una referencia a uno de los objetos de estado y le delega todo el trabajo específico del mismo. El contexto se comunica con el objeto de estado a través de la interfaz de estado. A su vez, el contexto expone un setter para pasarle un nuevo objeto de estado.
2. La interfaz Estado declara los métodos específicos del estado. Dichos métodos deben tener sentido para todos los estados. Básicamente, no queremos que alguno de los estados tenga métodos inútiles que nunca son invocados (Principio de Segregación de Interfaz).
3. Los Estados ofrecen sus propias implementaciones para los métodos específicos del estado. Para evitar duplicar código similar a través de varios estados diferentes, es viable incluir clases abstractas intermedias que encapsulen comportamientos comunes. A su vez, los objetos de estado pueden disponer de una referencia al contexto. A través de esta referencia, el estado puede extraer cualquier información requerida del objeto de contexto, así como iniciar transiciones de estado.
4. El nuevo estado del contexto puede ser establecido tanto por el estado actual del contexto como por el estado concreto. Éstos también tienen la capacidad de realizar la transición de estado sustituyendo el objeto de estado vinculado al contexto.



Aplicabilidad

- **Utilizamos State cuando tenemos un objeto que se comporta de forma diferente dependiendo de su estado actual, exista mucha cantidad de estados y el código del estado cambie frecuentemente.** El patrón sugiere extraer todo el código específico del estado y colocarlo dentro de un grupo de clases específicas. Como resultado, es posible sumar nuevos estados o cambiar los existentes independientemente entre sí, reduciendo el costo de mantenimiento.
- **También lo usamos si existiese alguna clase afectada por condicionales que alteran el modo en que se comporta la clase de acuerdo con los valores actuales de sus atributos.** State permite extraer “ramas” de esos condicionales a métodos de las correspondientes clases de estado. Al hacer esto, es posible limpiar atributos temporales y métodos “helper” relacionados con código específico del estado de fuera de la clase principal.
- **Finalmente, utilizamos este patrón cuando tengamos código duplicado por estados similares y transiciones de una máquina de estados basada en condiciones.** El patrón State permite construir jerarquías de clases de estado compuestas y reducir así la duplicación, extrayendo el código común y colocándolo en clases abstractas base.

Cómo implementarlo

1. Decidir qué clase actuará como contexto. Puede ser alguna clase existente que ya tiene el código basado en el estado, o una nueva clase (en caso de que el código específico del estado se encuentre distribuido a lo largo de varias clases).
2. Declarar la interfaz de estado. Aunque puede replicar los métodos declarados en el contexto, es importante centrarse en los que puedan contener comportamientos específicos del estado.
3. Para cada estado, crear una clase derivada de la interfaz de estado. Luego repasar los métodos del contexto y extraer todo el código relacionado con ese estado para colocarlo en la clase recién creada.

Cabe la posibilidad de que al hacer esto último encontremos que depende de miembros privados del contexto. Algunas soluciones alternativas:

- Hacer públicos esos campos o métodos.
- Convertir el comportamiento que se está extrayendo para colocarlo en un método público en el contexto, e invocarlo desde la clase de estado. Esta forma es poco feliz pero rápida y siempre se puede arreglar más adelante.
- Anidar las clases de estado en la clase contexto. Esto último sólo en caso de que el lenguaje de programación que estemos utilizando soporte clases anidadas.

4. En la clase contexto, agregar un campo de referencia del tipo de interfaz de estado y un setter público que permita sobrescribir el valor de ese campo.
5. Repasar nuevamente el método del contexto y sustituir los condicionales de estado vacíos por llamadas a los métodos correspondientes del objeto de estado.
6. Para modificar el estado del contexto, crear una instancia de una de las clases de estado y pasarla a la clase contexto. Esto se puede hacer dentro de la propia clase contexto, en distintos estados, o bien en el cliente. Del modo que se haga, la clase se vuelve dependiente del estado concreto que acaba de instanciar.

Pros y contras

Pros

- *Principio de responsabilidad única.* Organizamos el código relacionado a estados particulares en clases completamente separadas.
- *Principio Abierto/Cerrado.* Agregamos nuevos estados sin cambiar clases de estado existentes o la clase contexto.
- Simplificamos el código del contexto eliminando condicionales incómodos de la máquina de estados.

Contras

- Aplicar el patrón puede resultar excesivo si la máquina de estados tiene unos pocos estados o cambia poco.

Strategy

Propósito

Es un patrón de diseño de comportamiento que permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.

Problema

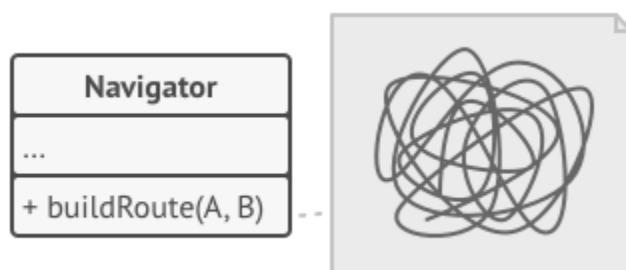
Vamos a crear una aplicación de navegación para viajeros. La aplicación gira alrededor de un mapa que ayuda a los usuarios a orientarse muy rápido en cualquier ciudad.

Una de las funcionalidades más solicitadas para la aplicación es la planificación automática de rutas. El usuario debe poder introducir una dirección y ver la ruta más rápida a ese destino mostrado en el mapa.

La primera versión de la aplicación sólo podía generar las rutas sobre calles y autopistas, lo que era absolutamente funcional para quienes viajaban en auto o en moto. Pero aparentemente, no

todo el mundo desea conducir durante sus vacaciones. Así, en la siguiente actualización, se sumó una opción para crear rutas a pie. Luego, se añadió otra opción para permitir a las personas usar el transporte público.

Ahora bien, esto era sólo el principio. Más tarde se planeó agregar rutas para ciclistas, y más tarde, otra opción para trazar rutas por todas las atracciones turísticas de una ciudad.



Desde el punto de vista comercial, la aplicación era un éxito. Sin embargo, la parte técnica provocaba muchos dolores de cabeza. Cada vez que se agregaba un nuevo algoritmo de enrutamiento, la clase principal del navegador prácticamente duplicaba su tamaño. En algún momento, se volvió demasiado difícil de mantener.

Cualquier modificación en alguno de los algoritmos, sea un bugfix sencillo o un mínimo ajuste de la representación de la calle, afectaba a toda la clase, lo cual iba aumentando las probabilidades de meter un error en un código ya funcional.

Adicionalmente, el trabajo en equipo se volvió poco eficiente. Las personas contratadas tras el lanzamiento comenzaron a quejarse por dedicar demasiado tiempo a resolver conflictos de integración. Implementar una nueva función exige cambiar la misma clase enorme, entrando en conflicto con el código producido por otras personas.

Solución

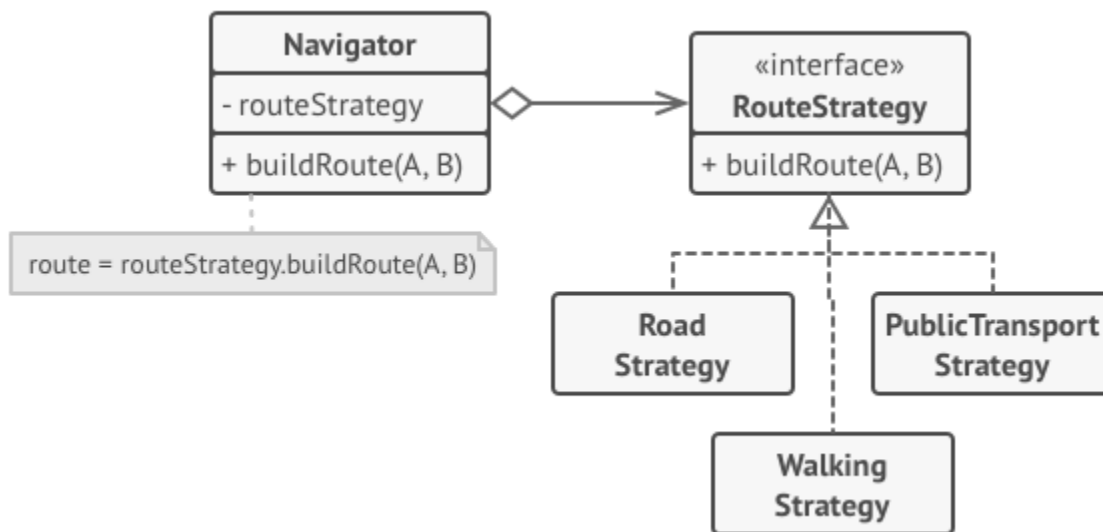
Strategy sugiere tomar esa clase que hace una cosa concreta de muchas formas diferentes y extraer todos esos algoritmos para colocarlos en clases separadas llamadas estrategias.

La clase original (contexto), debe tener un campo para almacenar una referencia a una de las estrategias. El contexto delega el trabajo a un objeto de estrategia en lugar de ejecutarlo por su cuenta.

La clase contexto no es responsable de seleccionar un algoritmo adecuado para la tarea. En su lugar, el cliente pasa la estrategia deseada. Más aún: la clase contexto no sabe mucho acerca de las estrategias. Funciona con todas las estrategias a través de la misma interfaz genérica, que

expone un único método para ejecutar el algoritmo encapsulado dentro de la estrategia seleccionada.

Así, el contexto se vuelve independiente de las estrategias concretas, con lo que es posible sumar nuevos algoritmos o modificar los existentes sin cambiar el código de la clase contexto o de otras estrategias.



En nuestra aplicación de navegación, todos los algoritmos de enrutamiento se pueden extraer y colocarse en su propia clase con un método `crearRuta`. El método acepta un origen y un destino, y devuelve una colección de puntos de control de la ruta.

Incluso teniendo los mismos argumentos, cada clase de enrutamiento puede crear una ruta diferente. A la clase navegadora no le importa qué algoritmo se selecciona ya que su única responsabilidad es representar un grupo de puntos en un mapa. La clase tiene un método para cambiar la estrategia de enrutamiento, de modo que sus clientes pueden reemplazar el comportamiento de enrutamiento por otro.

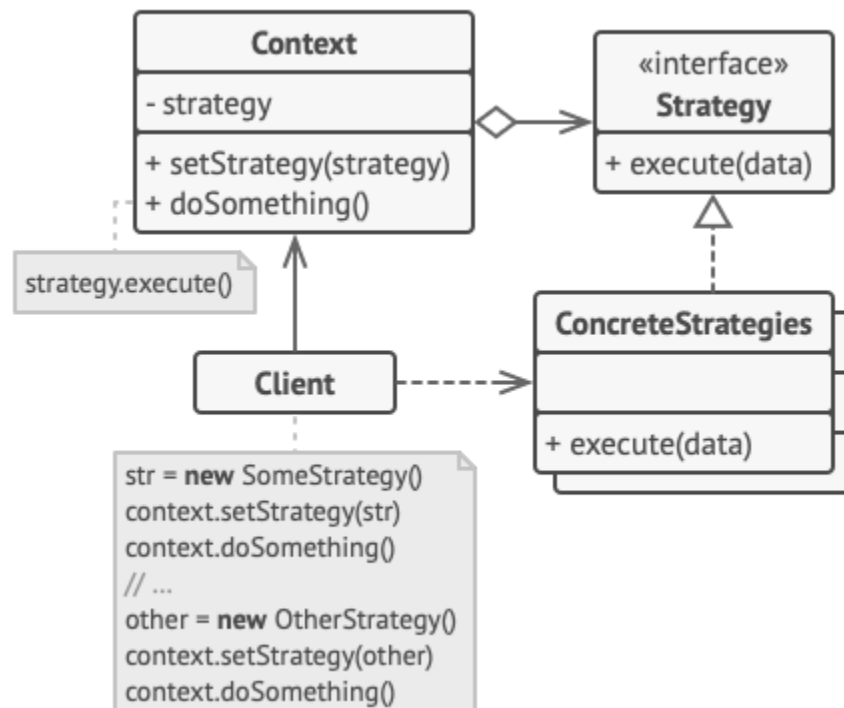
Analogía en el mundo real

Imaginemos que necesitamos llegar al aeropuerto. Se puede tomar el colectivo, pedir un taxi o ir en bicicleta. Éstas son las estrategias de transporte. Se puede elegir una de las estrategias, dependiendo de factores como el presupuesto o el tiempo.

Estructura

1. El Contexto (clase) mantiene una referencia a una de las estrategias concretas y se comunica con este objeto únicamente a través de la interfaz Estrategia.

2. La interfaz Estrategia es común a todas las estrategias concretas. Declara un método que el contexto utiliza para ejecutar la estrategia.
3. Las Estrategias Concretas implementan variaciones de un algoritmo, que serán utilizadas por la clase contexto.
4. Cada vez que la clase contexto necesita ejecutar el algoritmo, invoca el método de ejecución en el objeto de estrategia vinculado. El contexto no sabe con qué estrategia funciona o cómo se ejecuta el algoritmo.
5. El Cliente crea una estrategia específica y la pasa a la clase contexto. La clase contexto expone un setter que permite a los clientes sustituir la estrategia en tiempo de ejecución.



Aplicabilidad

- **Utilizamos Strategy cuando necesitemos utilizar variantes de un algoritmo dentro de un objeto y luego poder cambiar de un algoritmo a otro en tiempo de ejecución.** Este patrón permite alterar el comportamiento del objeto en tiempo de ejecución, asociándolo con objetos que pueden realizar subtarefas específicas de distintas maneras.
- **También se le da uso cuando existan clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.** Strategy permite extraer el comportamiento variante, para colocarlo en una jerarquía de clases separada y luego combinar las clases originales en una, reduciendo así el código duplicado.

- **Se utiliza el patrón para separar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan relevantes en el contexto de esa lógica.** El patrón Strategy permite aislar el código, los datos internos y las dependencias, del resto del código. Los distintos clientes reciben una interfaz simple para ejecutar los algoritmos y cambiarlos en tiempo de ejecución.
- **Strategy también sirve cuando la clase tenga un operador condicional que cambie entre distintas variantes del mismo algoritmo.** Nos permite suprimir dicho condicional, quitando todos los algoritmos para colocarlos en clases separadas, las cuales obviamente implementan la misma interfaz. El objeto original delega la ejecución a alguno de estos objetos, en lugar de implementar todas las variantes del algoritmo.

Cómo implementarlo

1. En la clase contexto, identificamos algún algoritmo que tienda a sufrir cambios frecuentes. También puede ser un condicional que seleccione y ejecute alguna variante del mismo algoritmo en tiempo de ejecución.
2. Declarar la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extraemos los algoritmos y los colocamos en sus propias clases. Todas las clases deben implementar la misma interfaz estrategia.
4. En la clase contexto, agregamos un campo para almacenar una referencia al objeto de estrategia. Exponemos un setter para reemplazar los valores de ese campo. El contexto debe trabajar con la estrategia únicamente a través de la interfaz estrategia. La clase contexto puede definir una interfaz que permita a la estrategia acceder a sus datos.
5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

Pros y contras

Pros

- Se pueden intercambiar en tiempo de ejecución algoritmos usados dentro de un objeto.
- Es posible aislar los detalles de implementación de un algoritmo del código que lo utiliza.
- Logramos sustituir herencia por composición.
- *Principio Abierto/Cerrado.* Se logra agregar nuevas estrategias sin tener que cambiar el contexto.

Contras

- Si sólo existen algunos algoritmos que raramente cambian, no hay una razón real para complejizar el programa en exceso con nuevas clases e interfaces que vengan con el patrón.

- Los clientes deben conocer las diferencias entre estrategias para poder elegir la correcta.
- Muchos lenguajes de programación modernos tienen un soporte de tipo funcional que permite implementar distintas versiones de un algoritmo dentro de un grupo de funciones anónimas. Entonces es posible utilizar estas funciones exactamente como se hubiesen utilizado los objetos de estrategia, pero sin saturar el código con clases e interfaces adicionales.

Template Method

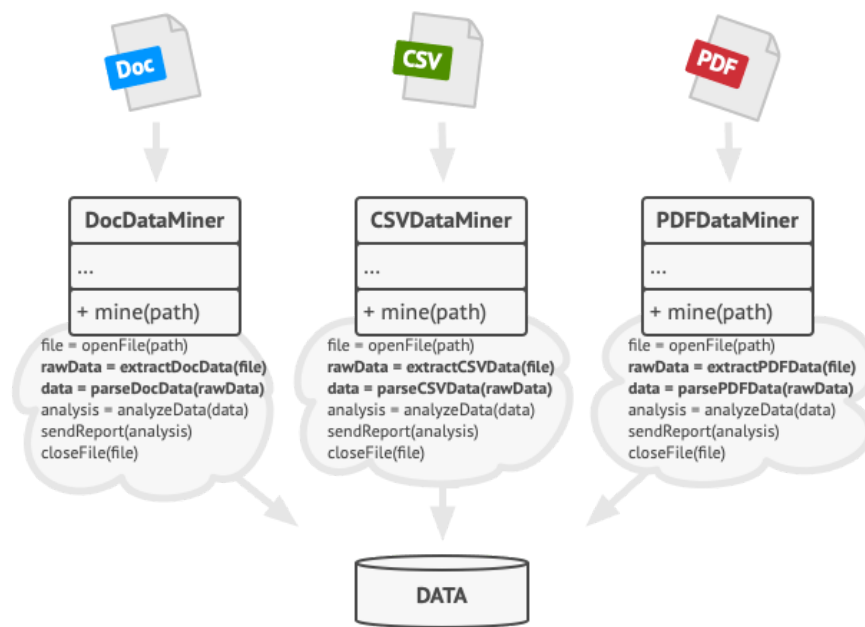
Propósito

Es un patrón de diseño de comportamiento que define el esqueleto de un algoritmo en la superclase pero permite que las subclasses sobrescriban pasos del algoritmo sin cambiar su estructura.

Problema

Estamos construyendo una aplicación de minería de datos que analiza documentos corporativos. Los usuarios suben a la aplicación documentos en distintos formatos (PDF, DOC, CSV) y ésta intenta extraer la información relevante de estos documentos en un formato uniforme.

La primera versión de la aplicación sólo funcionaba con archivos DOC. La siguiente versión soportaba archivos en formato CSV. Un mes después, le “enseñamos” a extraer datos de archivos PDF.



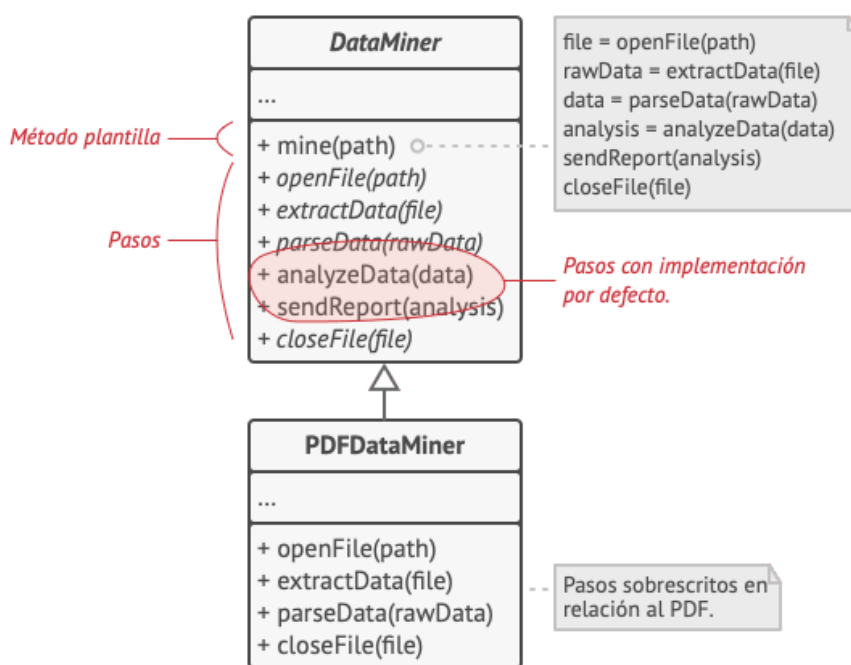
En algún momento, notamos que las tres clases tienen bastante código similar. Es cierto: el código para gestionar distintos formatos de datos es totalmente diferente en todas las clases. Sin embargo, el código para procesar y analizar los datos es casi idéntico. ¿No sería bueno eliminar la duplicación de código, dejando intacta la estructura del algoritmo?


Además, existe otro problema con el código cliente que utiliza esas clases. Existen muchos condicionales que seleccionan un curso de acción determinado, dependiendo de la clase del objeto de procesamiento. Si las tres clases de procesamiento tienen una interfaz común o una clase base, se pueden eliminar los condicionales en el código cliente y utilizar polimorfismo al invocar métodos en un objeto de procesamiento.

Solución

Template Method sugiere dividir un algoritmo en una serie de pasos, luego convertir estos pasos en métodos y finalmente, colocar una serie de llamadas a esos métodos dentro de un único método plantilla. Dichos pasos pueden ser abstractos, o contar con una implementación predeterminada. Para utilizar el algoritmo, el cliente debe aportar su propia subclase, implementar los pasos abstractos y sobrescribir algunos de los opcionales si es necesario (pero no el propio método plantilla).

Veamos cómo funciona en nuestra aplicación de minería de datos. Es posible crear una clase base para los tres algoritmos de análisis. Esta clase define un método plantilla consistente en una serie de llamadas a varios pasos de procesamiento de documentos.





Inicialmente, se pueden declarar los pasos como abstractos, forzando a las subclases a proporcionar sus implementaciones para estos métodos. En nuestro caso, las subclases ya cuentan con todas las implementaciones requeridas, por lo que lo único que habrá que hacer es ajustar las firmas de los métodos para que coincidan con los métodos de la superclase.

Ahora bien, analicemos qué podemos hacer para deshacernos del código duplicado. Aparentemente, el código para abrir/cerrar archivos y extraer/analizar información varía en función del formato de los datos, por lo que no tiene sentido tocar estos métodos. Sin embargo, la implementación de otros pasos, como analizar los datos sin procesar y generar informes es muy similar. Así, estos comportamientos pueden colocarse en la clase base, donde las subclases pueden compartir ese código.

Como se puede ver, existen dos tipos de pasos:

- Los abstractos, que deben ser implementados por todas las subclases.
- Los opcionales que ya tienen cierta implementación por defecto, pero aún así se pueden sobrescribir si es necesario.

Hay otro tipo de pasos, llamados hooks. Un hook es un paso opcional con un cuerpo vacío. Un método plantilla funcionará aunque el hook no se sobrescriba. En general, los hooks se colocan antes y después de pasos cruciales de los algoritmos, brindando a las subclases puntos extra de extensión para un algoritmo.

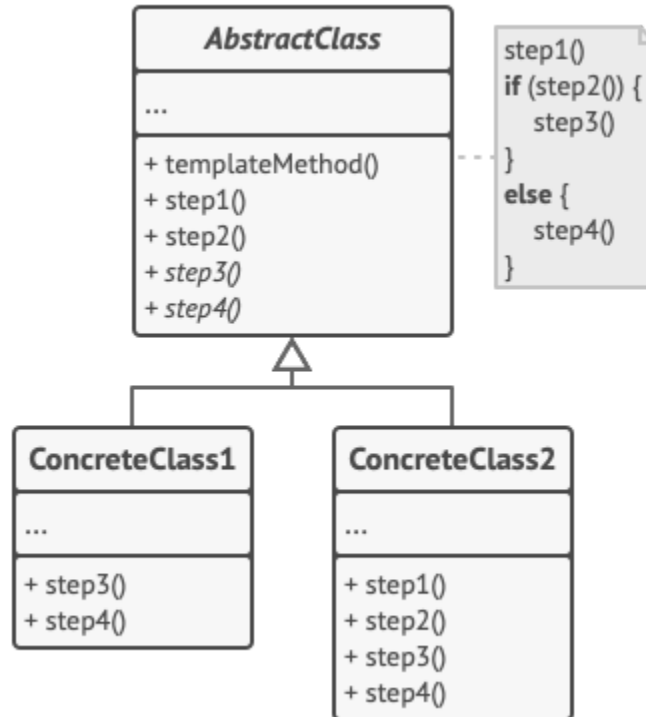
Analogía en el mundo real

El enfoque del template method se puede emplear en la construcción de viviendas en masa. El plan arquitectónico para construir una casa estándar puede tener varios puntos de extensión que permitirían a un propietario ajustar detalles de la casa resultante.

Cada paso de la construcción (colocar los cimientos, el armazón, construir las paredes, instalar las tuberías para el agua y el cableado para la electricidad, etc.) puede modificarse para que la casa resultante sea levemente diferente de las demás.

Estructura

1. La Clase Abstracta declara tanto métodos que actúan como pasos de un algoritmo, como el propio template method que ejecuta estos métodos en un orden determinado. Los pasos pueden declararse abstractos o contar con una implementación predeterminada.
2. Las Clases Concretas pueden sobrescribir todos los pasos, pero no el propio método plantilla.




Aplicabilidad

- **Utilizamos Template Method cuando queremos permitir a los clientes que extiendan pasos puntuales de un algoritmo, pero no todo el algoritmo o su estructura.** Template Method permite convertir un algoritmo monolítico en una serie de pasos individuales que se pueden extender con facilidad con subclasses, manteniendo intacta la estructura definida en una superclase.
- **También le damos uso cuando existan muchas clases que contengan algoritmos casi idénticos, pero con diferencias mínimas. Como resultado, puede que haya necesidad de modificar las clases cuando el algoritmo cambie.** Cuando se convierte un algoritmo así en un template method, también es viable enviar los pasos con implementaciones similares a una superclase, eliminando la duplicación del código. El código que varía entre subclasses puede permanecer en las subclasses.

Cómo implementarlo

1. Analizar el algoritmo objetivo para ver si es posible dividirlo en pasos. Considerar qué pasos son comunes a todas las subclasses y cuáles serán únicos.
2. Crear la clase base abstracta y declarar en ella tanto el template method como un grupo de métodos abstractos que representen los pasos del algoritmo. Perfilar la estructura del



algoritmo en el método plantilla ejecutando los pasos correspondientes. Considerar declarar el método plantilla como final para evitar que las subclases lo sobrescriban.

3. No hay inconveniente en que todos los pasos terminen siendo abstractos. No obstante, a algunos pasos les serviría tener una implementación por defecto. Las subclases no deben implementar estos métodos.
4. Pensar en agregar hooks entre los pasos cruciales del algoritmo.
5. Para cada variación del algoritmo, crear una nueva subclase concreta. Ésta debe implementar todos los pasos abstractos, pero también puede sobrescribir algunos de los opcionales.

Pros y contra

Pros

- Se puede permitir a los clientes que sobrescriban solo ciertas partes de un algoritmo grande, para que les afecten menos los cambios que tienen lugar en otras partes del algoritmo.
- Es posible colocar el código duplicado dentro de una superclase.

Contras

- Algunos clientes pueden verse limitados por el esqueleto proporcionado por un algoritmo.
- Existe la posibilidad de romper el *Principio de Sustitución de Liskov* eliminando una implementación predeterminada de un paso a través de una subclase.
- Los métodos plantilla tienden a ser más difíciles de mantener cuantos más pasos tengan.



Unidad 4

Generators

Definición

Introducidas con [PEP 255](#), los generators, o funciones generadoras, son un tipo especial de función que devuelve un lazy iterator (o iterador “perezoso”). Un iterador, es un objeto que puede recorrerse como una lista, pero a diferencia de éstas no almacenan su contenido en memoria sino que nos permiten obtener los datos producidos a medida que se solicitan.

Propósito

¿Para qué necesitamos los *generators*? ¿Cuándo conviene usarlos?

Muchas veces nos encontramos ante la situación de tener que trabajar con conjuntos de datos extremadamente grandes que llevan al extremo (o sobrepasan) la memoria de la máquina, o también puede presentarse la situación de tener que desarrollar una función que necesita mantener un estado interno cada vez que se llama, pero la función es demasiado pequeña para justificar la creación de una clase. En estos casos los *generators* pueden ayudar.

Ejemplos de uso

Leyendo archivos

Supongamos que estamos desarrollando una aplicación que requiere procesar un archivo y conocer su extensión, es decir, cuántas líneas tiene.

Nuestro primer intento podría ser algo como lo siguiente:

```
def txt_reader(file_name):
    file = open(file_name)
    result = file.read().split("\n")
    return result

txt_gen = txt_reader("mi_archivo.txt")
row_count = 0
for row in txt_gen:
    row_count += 1

print(f"La cantidad de líneas es {row_count}")
```

Como vemos, la función `txt_reader`, abre el archivo y carga todo su contenido en una lista. Ahora bien ¿Qué ocurre si al ejecutar la función, nuestro programa excede la memoria disponible?

Si esto ocurre, obtendremos un error del tipo `MemoryError` y nuestro programa será terminado por el sistema operativo.

Para evitar este inconveniente, podemos implementar una solución que utilice *generators*. Para lograrlo, podemos reescribir nuestra función `txt_reader` de la siguiente manera:

```
def txt_reader(file_name):  
    for row in open(file_name, "r"):  
        yield row
```

¿Cómo funciona esta nueva versión?

En pocas palabras, convertimos nuestra función `txt_reader` en un *generator*.

Analizando en profundidad el código, vemos que a diferencia de la versión original de la función, estamos abriendo el archivo e iterando por cada una de sus filas, mientras que en la versión original tratábamos de cargar todo el contenido en una variable.

Por otra parte, estamos utilizando la instrucción `yield` en vez de `return`.

La instrucción `yield` es la que nos permite convertir nuestra función en una función de tipo *generator*.

Generando secuencias infinitas

Cuando necesitamos generar una secuencia de números, por ejemplo, es común utilizar la función `range`. Ahora bien, ¿Cómo podemos generar una secuencia de números “infinita”?

En principio nos toparemos con el problema de no saber la cantidad de números a generar, dado que necesitamos una secuencia “infinita”. Pero suponiendo que lo podemos determinar de alguna forma, esa cantidad seguramente sean muchos números, tantos como para no poder cargarlos en una lista (ya que las listas se cargan en memoria principal).

Es por esto que debemos recurrir a los *generators*. Veamos un ejemplo:

```
def mi_secuencia_infinita():
    num = 0
    while True:
        yield num
        num += 1
```

Para utilizar esta función podemos realizar lo siguiente:

```
for i in mi_secuencia_infinita():
...     print(i, end=" ")

0 1 2 3 4 5
[...]
6157818 6157819 6157820 ...
```

También podríamos utilizar la instrucción *next*, que nos devuelve el próximo valor de un *generator*.

```
>>> gen = infinite_sequence()
>>> next(gen)
0
>>> next(gen)
1
....
```

De esta forma conseguimos generar una secuencia infinita de números utilizando *generators*.

La instrucción yield

Cuando se invoca a una función de tipo *generator*, se devuelve un iterador especial llamado *generator*. Para utilizarlo podemos utilizar una estructura de repetición como el *for* o una instrucción especial como *next*.

Cuando sobre un *generator*, se utiliza una función como *next*, el código dentro de la función se ejecuta hasta alcanzar la instrucción *yield*. En ese momento el programa suspende la ejecución de la función y devuelve el valor obtenido a quién invoca la función.

Cabe destacar la diferencia con la instrucción `return`, que detiene completamente la ejecución de la función.

Cuando se suspende una función, se guarda el estado de esa función. Esto incluye el estado de las variables locales, el puntero de instrucción, la pila interna y cualquier manejo de excepciones.

Esto le permite reanudar la ejecución de la función en futuras invocaciones. Cuando esto ocurre, la ejecución se reanuda en la instrucción siguiente a la instrucción `yield`.

Creando generators a través de expresiones

Al igual que las listas por comprensión, podemos construir expresiones que nos permiten crear *generators* en pocas líneas de código. También son útiles en los mismos casos en los que se utilizan listas por comprensión, con un beneficio adicional: el *generator* no almacena los datos en memoria antes de la iteración.

Veamos algunos ejemplos:

```
# Generator expression que me permite leer un archivo
f_gen = ( fila for fila in open("mi_archivo.csv") )

# Generator expression para obtener el cuadrado de varios números
C_gen = ( n**2 for n in range(3) )
```

La diferencia sintáctica con las listas por comprensión, es el uso de paréntesis en lugar de corchetes

Generators y Pipelines

Los *pipelines* permiten encadenar código para procesar grandes conjuntos de datos o flujos de datos sin desbordar la memoria principal de la máquina.

Supongamos que debemos procesar un archivo CSV muy grande con ventas de un comercio y debemos procesarlo para obtener el monto total de las ventas realizadas en el período comprendido en el archivo. De forma genérica nuestra estrategia para procesarlo podría ser:

1. Leer cada línea
2. Separar cada línea en una lista
3. Obtener los nombres de cada columna
4. Crear un diccionario con los nombres de columnas como clave y los datos como valores
5. Filtrar los datos y/o columnas que no nos interesan
6. Procesar los datos que necesitamos

Utilizando generators y pipelines, podemos expresar nuestra estrategia de la siguiente forma:

```
# crear generator para obtener las líneas del archivo
lines = (line for line in open("mi_archivo.csv"))

# crear generator para parsear los datos de cada línea del archivo
list_line = (s.rstrip().split(",") for s in lines)

# obtener la primer línea del csv, los nombres de cada columna
cols = next(list_line)

# crear diccionario con los nombres de columnas y los datos
data_dicts = (dict(zip(cols, data)) for data in list_line)

# crear generator para procesar los datos
funding = (
    int(data_dict["venta"]) for data_dict in data_dicts
    if data_dict["periodo"] == "2021"
)

# utilizo sum para sumarizar todos los elementos del generator
total = sum(funding)

print(f"El total de ventas del periodo es: ${total}")
```



Unidad 5

Threads

Introducción

Desarrollar una aplicación utilizando (múltiples) hilos de ejecución, nos permite ejecutar diferentes partes de la misma en forma concurrente, es decir, (casi) al mismo tiempo; lo que resulta en una aceleración en la ejecución. Por supuesto, para aprovechar este tipo de optimización, debemos diseñar nuestra aplicación de forma tal que podamos lanzar cada “parte” como un hilo de ejecución distinto.

La programación con múltiples hilos también nos forzará a identificar secciones críticas de nuestra aplicación, donde realizaremos el manejo de recursos compartidos de forma sincronizada.

Definición

Antes de poder dar una definición sobre qué es un hilo, debemos analizar y comprender qué es un proceso.

De manera sencilla y simplificando varios pasos, podemos decir que cuando ejecutamos un programa, desde un simple script hasta una aplicación compleja, el sistema operativo debe cargar dicho programa en memoria para poder ser ejecutado. Una vez hecho esto, el sistema operativo copiará las instrucciones definidas en el programa al procesador para que sean ejecutadas.

A la instancia de un programa en memoria se la conoce como proceso.

Formalmente, diremos que un proceso es una unidad de actividad caracterizada por un solo hilo secuencial de ejecución, un estado actual y un conjunto de recursos del sistema asociados (memoria, descriptores de archivos, etc).

Tradicionalmente el concepto de proceso incluía tanto la propiedad de los recursos como así también la planificación de su ejecución. Con el advenimiento de los sistemas operativos modernos, éstas dos características fueron tratándose de manera independiente llegando a la siguiente definición:

- **Proceso:** es la unidad dueña de los recursos
- **Hilo (Thread):** es la unidad mínima de ejecución que puede ser planificada para su ejecución.

Cuando diseñamos una aplicación para que la misma posea múltiples hilos de ejecución, diremos que la aplicación es multi-hilo.

Un hilo, al igual que un proceso posee un estado, contexto, pila de ejecución. También posee acceso a memoria del proceso que los contiene, espacio de almacenamiento para variables locales.

Diferencias entre hilo y proceso

Las principales diferencias, desde el punto de vista del sistema operativo son:

Característica	Proceso	Hilo
Memoria compartida	Por defecto los procesos no comparten memoria entre sí	Los hilos de un mismo proceso, comparten la memoria asignada al mismo.
Creación y finalización	Lenta	Rápida
Cambio de contexto de ejecución	Lenta	Rápida
Comunicación	La comunicación entre procesos independientes requiere de la intervención del sistema operativo	Los hilos mejoran la eficiencia de la comunicación entre diferentes programas, al estar los hilos dentro del mismo proceso, la comunicación se puede realizar sin intervención del sistema operativo.

Aplicaciones con un único hilo (proceso)

Supongamos el siguiente script:

```
from time import sleep, perf_counter

def task():
    print('comenzando...')
    sleep(1)
    print('fin')
```

```
start_time = perf_counter()

task()
task()

end_time = perf_counter()

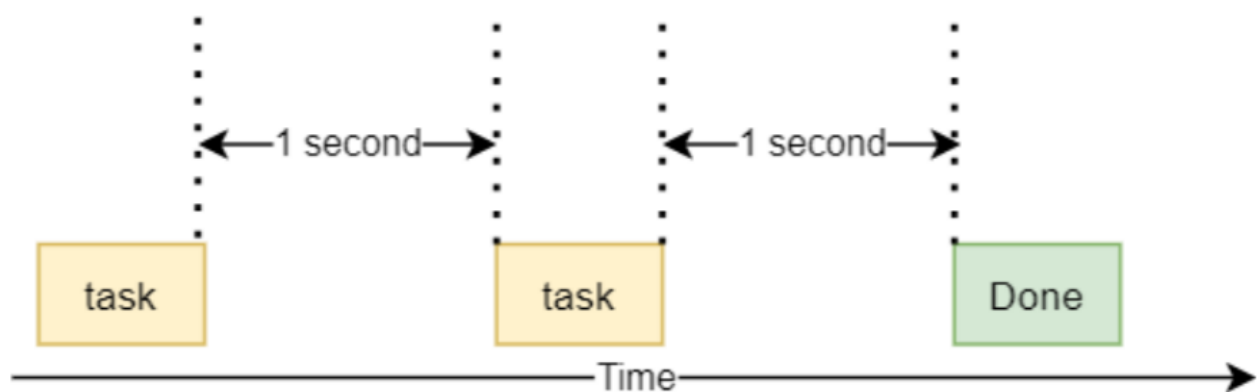
print(f'Tiempo de ejecución: {end_time - start_time: 0.2f} segundos')
```

Al ejecutar, obtendremos la siguiente salida:

```
comenzando...
fin
comenzando...
fin
Tiempo de ejecución: 2.00 segundos
```

Nota: es posible obtener distintas salidas en diferentes máquinas.

Como se esperaba, la ejecución del script demoró 2 segundos. Podríamos graficar la ejecución de la siguiente manera



Vemos que cada vez que se llama a la función *task*, y se alcanza la instrucción *sleep*, el procesador no ejecuta nada, permanece ocioso, lo cual no es muy eficiente.

Esta aplicación posee un único hilo de ejecución, el principal y es por esto que es una aplicación mono-hilo.

Aplicaciones multi-hilo (multi-threading)

Para utilizar/crear hilos dentro de nuestra aplicación, utilizaremos el módulo `threading`. Este módulo nos provee de la clase `Thread` con la cual podremos crear hilos.

El constructor de la clase `Thread` admite varios parámetros, los 2 más importantes son:

- `target`: función a ejecutar
- `args`: argumentos para pasar a la función (`target`).

Por otra parte, nos provee también de 2 métodos que son:

- `start`: permite lanzar la ejecución del hilo en cuestión
- `join`: indica al hilo principal que debe esperar la finalización de los hilos lanzados para continuar la ejecución.

Veamos un ejemplo:

```
from time import sleep, perf_counter
from threading import Thread

def task():
    print('comenzando...')
    sleep(1)
    print('fin')

start_time = perf_counter()

# Crea 2 hilos
t1 = Thread(target=task)
t2 = Thread(target=task)

# Lanza la ejecución de los hilos
t1.start()
t2.start()
```

```
# espera por la finalización de los hilos para poder continuar la
ejecución del hilo principal
t1.join()
t2.join()

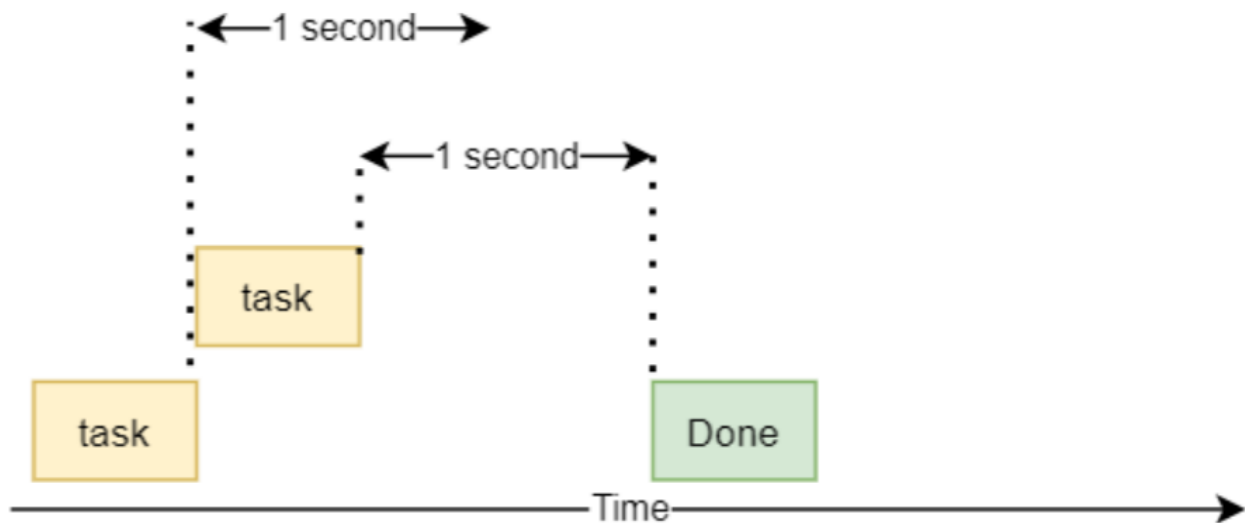
end_time = perf_counter()

print(f'Tiempo de ejecución: {end_time - start_time: 0.2f} segundos')
```

Y su salida es:

```
comenzando...
comenzando...
fin
fin
Tiempo de ejecución: 1.00 segundos
```

En este caso, podríamos graficar la ejecución de la siguiente manera



Si no usamos las instrucciones `join`, la salida sería la siguiente:

```
comenzando...
Comenzando...
```



```
Tiempo de ejecución: 0.00 segundos
fin
fin
```

¿Por qué?

Simplemente porque desde el proceso, se lanzan los hilos y se continúa con la ejecución del mismo sin esperar a que finalicen. Tener en cuenta que el proceso, de todas formas, esperará la finalización de los hilos antes de finalizar.

Un caso especial - daemons threads

Existe un caso “especial” de hilos, llamados “daemons”. Estos hilos se caracterizan por finalizar inmediatamente cuando el programa (hilo principal) finaliza.

Para crearlos, basta con agregar el parámetro `daemon=True` en el constructor:

```
Thread(target=task, args=(1,), daemon=True)
```

Nota: la instrucción `join` invalida el parámetro `daemon`.

Los daemons suelen ser herramientas útiles para realizar tareas en segundo plano (background), como puede ser leer mensajes de una interfaz (gráfica, cola de mensajes, etc)..

Trabajando con múltiples hilos


Cuando trabajamos con múltiples hilos, debemos (como se vió en el ejemplo anterior) lanzar cada uno y probablemente indicar al hilo principal que debe esperar por la finalización de los mismos para continuar. Esto podríamos hacerlo con las instrucciones `start` y `join` como se vió anteriormente, pero existe una forma más simple de realizarlo utilizando `ThreadPoolExecutor`.

`ThreadPoolExecutor` permite manejar la creación y finalización de un grupo de hilos de forma automática:

```
from concurrent.futures import ThreadPoolExecutor

with ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(task, range(3))
```

El parámetro `max_workers` indica que como máximo habrá 3 hilos que se ejecutarán de forma concurrente.



En el ejemplo anterior, las instrucciones `start` y `join` están implícitas dentro del bloque `with`. Al finalizar el bloque se realizan los “`join`” sobre cada thread lanzado.

Al usar `ThreadPoolExecutor` debemos estar muy atentos ya que si la función lanzada como thread, en nuestro caso “task”, genera una excepción `ThreadPoolExecutor` oculta dicha excepción lo que dificultará encontrar el error en nuestro código.

Concurrencia

Cuando desarrollamos aplicaciones utilizando threads, contamos con beneficios en cuanto a la velocidad de ejecución de nuestro programa gracias a que parte de nuestra aplicación se ejecutará de forma solapada. Pero la concurrencia también acarrea algunos problemas como son la *condición de carrera* o el *deadlock*. Por suerte, existen técnicas que nos ayudan a evitarlos, pero antes de seguir veamos qué son y cómo se producen dichos problemas.

Condición de carrera

Una definición formal puede ser: *“Una condición de carrera es una situación en la cual múltiples procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones”*.

De la definición se deriva que debe existir un recurso compartido (por ejemplo una variable global), 2 o más threads que accedan a dicho recurso (sin ningún tipo de control) y el resultado que se almacena en dicho recurso depende del orden de ejecución de los procesos o threads, en otras palabras, el último proceso que accede define el valor del recurso.

Esto nos conduce a pensar que al estar programando aplicaciones que aplican concurrencia en sus algoritmos debemos asegurar la integridad de los resultados, independientemente de la velocidad de ejecución de los procesos o threads que participan.

Analicemos esto con un ejemplo.

Supongamos una clase que realiza operaciones de I/O (entrada-salida), como puede ser lectura-escritura de un archivo, una base de datos, etc.

```
from concurrent.futures import ThreadPoolExecutor
import logging
import time
```

```

class GlobalWriter:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info(f"Comenzando Thread {name}...")
        local_copy = self.value
        # simulamos algún tipo de procesamiento con los datos
        local_copy += 1
        time.sleep(0.1)
        # termina la operación de I/O y continuamos con la ejecución
        self.value = local_copy
        logging.info(f"Finalizando Thread {name}")

```

La clase GlobalWriter, simula leer algo del disco, procesarlo y luego lo vuelve a escribir en disco.

¿Qué ocurriría si lanzamos 2 (o más) threads en paralelo que ejecuten el método update?

```

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    gw = GlobalWriter()
    logging.info(f"Valor inicial: {gw.value}")
    with ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(gw.update, index)

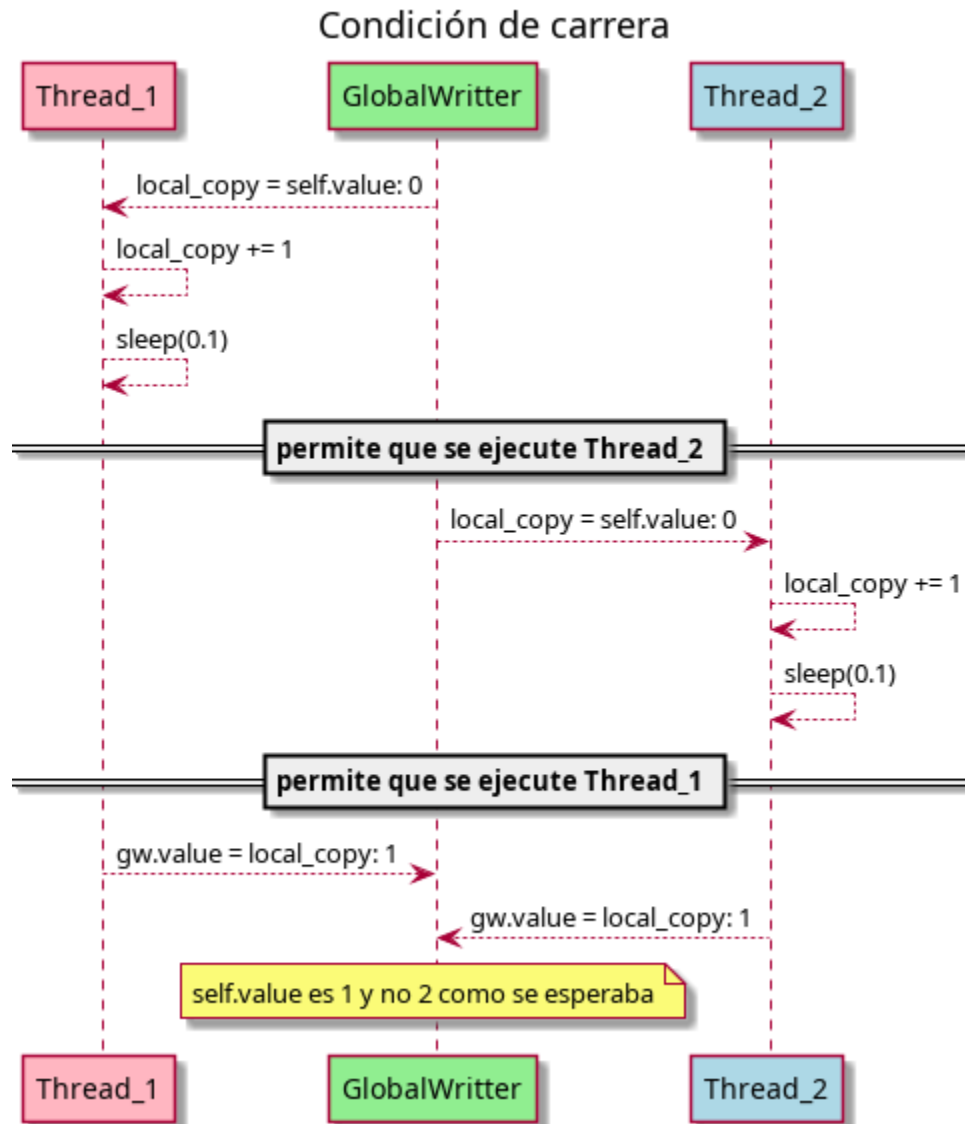
    logging.info(f"Valor final: {gw.value}")

```

La salida esperada sería que el valor final sea 2, pero si ejecutamos este código vemos que la salida es 1.

¿Por qué ocurre esto?

Como imaginarás, porque se dió una condición de carrera. Analicemos en profundidad qué ocurrió:



Cada vez que un thread es bloqueado porque realiza alguna operación de entrada-salida, el sistema operativo selecciona otro thread (que no esté bloqueado) y le cede el control del procesador para que pueda ejecutar. En nuestro ejemplo, la instrucción “sleep” simula una operación de entrada-salida.

Sincronización de hilos

Para evitar tener una condición de carrera en nuestra aplicación, debemos utilizar técnicas que nos permitan sincronizar la ejecución de los hilos de la misma. Básicamente, en aquellas aplicaciones que utilicen hilos y exista algún recurso compartido entre los mismos, debemos garantizar que sólo un hilo a la vez acceda a dicho recurso.

Para lograr la sincronización de los hilos de una aplicación, primero debemos identificar las secciones críticas, es decir las porciones de código que realizan lecturas y escrituras sobre recursos compartidos.

Una vez identificadas las secciones críticas y garantizar que sólo un hilo a la vez ejecute las instrucciones que allí se encuentran, utilizaremos una estructura denominada semáforo para permitir o rechazar el acceso a la misma por parte de los distintos hilos de la aplicación.

Para implementar semáforos en Python, utilizamos el objeto `threading.Lock()`. Este objeto implementa lo que comúnmente se conoce como MuTex (Mutual Exclusion) o semáforo binario.

Como mencionamos anteriormente, Lock es un objeto que actúa como una barrera. Solo un hilo a la vez puede acceder a la sección crítica. Cualquier otro hilo ingresar, el mismo se bloqueará hasta que el hilo que anteriormente ingresó a la sección crítica la abandone.

Este mecanismo garantiza que el sistema operativo no intercambiará un hilo si se encuentra ejecutando las operaciones de la sección crítica.

Retomando el objeto Lock, posee 2 operaciones básicas que son:

- `.release ()`: libera el semáforo
- `.acquire ()`. Bloquea el semáforo.

¿Cómo funcionan los semáforos?

Supongamos que tenemos una aplicación con los hilos A y B. Antes de ingresar a su sección crítica, el hilo A debe llamar a `my_lock.acquire()` para intentar obtener el bloqueo del semáforo. Si el semáforo no fue tomado previamente, el hilo A podrá acceder a su sección crítica y ejecutar las operaciones que allí se encuentren.

Cuando el hilo B desea acceder a su sección crítica, también deberá ejecutar `my_lock.acquire()`, pero como el semáforo ya había sido bloqueado por el hilo A, el hilo B pasará a un estado de espera (bloqueado) hasta que el hilo A libere el semáforo.

Para liberar el semáforo cada hilo deberá ejecutar la instrucción `my_lock.release()`.

También podemos utilizar el objeto Lock como un context manager con la cláusula with, de forma tal que no tendremos necesidad de invocar de forma explícita las funciones .acquire y .release para bloquear y liberar el semáforo.

Veamos un ejemplo. Si modificamos el código del ejemplo anterior y agregamos el uso de semáforos obtenemos lo siguiente:

```
from threading import Lock
from concurrent.futures import ThreadPoolExecutor
import logging
import time


class GlobalWriter:
    def __init__(self):
        self.value = 0
        self._lock = Lock()

    def update(self, name):
        with self._lock:
            logging.info(f"Comenzando Thread {name}")
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.info(f"Finalizando Thread {name}")
```

Deadlock

Un deadlock lo podemos definir como el bloqueo permanente de un conjunto de procesos o hilos que o bien compiten por recursos o bien se comunican entre sí.

Un conjunto de procesos o hilos está interbloqueado cuando cada uno se encuentra bloqueado a la espera de un evento (por lo general, la liberación de un recurso) que sólo puede generar otro proceso o hilo bloqueado del mismo conjunto.



Cuando desarrollamos aplicaciones utilizando hilos, no debemos olvidarnos de liberar los semáforos ya que de lo contrario, podremos generar un deadlock.

Un ejemplo simple de esta situación sería el siguiente:

```
import threading

l = threading.Lock()
print("antes primer bloqueo")
l.acquire()
print("antes del segundo bloqueo")
l.acquire()
...
```

Si ejecutamos este ejemplo, al alcanzar la segunda ejecución de la función `.acquire()`, el hilo se bloqueará a la espera de que se libere el semáforo.

Para evitar este tipo de errores, es recomendable maximizar el uso del objeto `Lock` como context manager siempre que se pueda.



Unidad 6

Coroutines

Introducción

En cierta forma, las corrutinas (*coroutine*) son funciones similares a los generadores (*generators*) vistos anteriormente, pero se diferencian principalmente en que:

- los generadores son productores de datos.
- mientras que las corrutinas son consumidores de datos

Si no recordás qué era un *generator* o cómo utilizarlo, podés volver a leer a la unidad [Unidad 4 - Generators](#)

Definición

A modo de definición, podemos decir que una *coroutine* es un *generator* al cual podemos enviarle datos, como parámetros a una función, para que sean utilizados (consumidos) por la misma.

Usando yield como expresión

Antes de empezar, recordemos cómo se creaba y utilizaba un *generator*

```
def mi_secuencia_infinita():
    num = 0
    while True:
        yield num
        num += 1

mg = mi_secuencia_infinita()
for n in mg:
    Print(n)
```

En este ejemplo, creamos el *generator* “mi_secuencia_infinita” que luego utilizamos con un *for*.

Ahora bien, este ejemplo siempre comienza la secuencia en 0 (cero). ¿Qué ocurre si necesitamos enviarle algún valor para modificar el inicio de la secuencia?

Alguien podríamos sugerir que realizáramos lo siguiente:

```
def mi_secuencia_infinita(n):
    num = n
    while True:
        yield num
        num += 1
```

Y funciona, pero ¿Qué ocurre si deseamos ir modificando constantemente el valor inicial de la secuencia?

Para poder realizar esto, debemos modificar nuestro *generator* y convertirlo en una *coroutine* y para lograrlo, debemos utilizar la instrucción *yield* como una expresión.

```
def mi_coroutine():
    num = 0
    while True:
        n = (yield num)
        if n is not None:
            num = n

    num += 1
```

Esta forma de utilizar la instrucción *yield*, permite manipular el valores dentro de la *coroutine*. Más importante aún, permite que enviemos un valor al *generator/coroutine* mientras la estamos utilizando.

Send y close

Existen dos métodos especiales que todo *generator* posee, y por ende una *coroutine* también, que son:

- **Send:** permite enviar valores a un *generator/coroutine*
- **Close:** permite cerrar (finalizar) un *generator/coroutine*

Send

Como sabemos, el sólo hecho de definir un *generator* no hace que el mismo se ejecute. Para que el *generator* comience a producir valores, debemos utilizar la instrucción *next* o *send*. Con las *coroutines* ocurre exactamente lo mismo.

La instrucción `send` permite enviarle valores al [generator/coroutine](#)

Ejemplo:

```
def mi_coroutine():
    ...

s1 = mi_coroutine()
s1.send(None) # seria como hacer next(s1)
> 0
s1.send(88)
> 89
```

Close

Como su nombre lo indica, `.close` nos permite detener un [generator](#). Esto puede resultar especialmente útil cuando tenemos generadores de secuencias infinitas.

Este método genera una excepción de tipo [GeneratorExit](#) que podemos atrapar dentro del [generator/coroutine](#) y realizar tareas de finalización del mismo.

Ejemplo:

```
def mi_coroutine():
    num = 0
    try:
        while True:
            n = (yield num)
            if n is not None:
                num = n

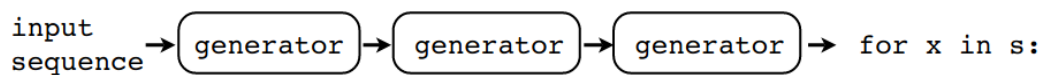
            num += 1
    except GeneratorExit:
        print("Cerrando coroutine...")
```

Pipelines

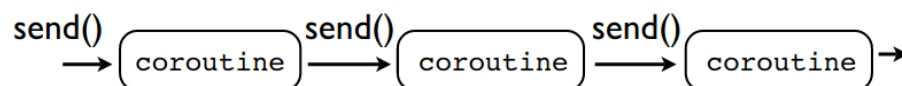
Al igual que los *generators*, las *coroutines* admiten encadenar su ejecución permitiendo de esa forma crear *pipelines*. Si bien el concepto es el mismo, existen algunas diferencias.

La primer diferencia es conceptual ya que los pipelines realizados con *generators* siguen una estrategia de *pull* de datos mientras que los pipelines creados con *coroutines* siguen una estrategia de tipo *push*, es decir, una coroutine envía datos a la siguiente (utilizando el método *send()*).

Estrategia Pull (*Generators*)

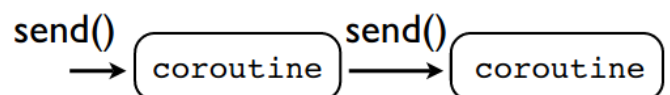


Estrategia Push (*Coroutines*)

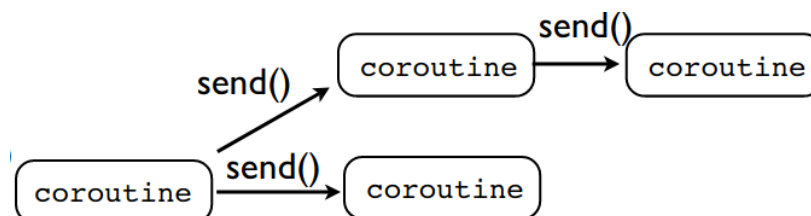


La segunda diferencia radica en que los *pipelines* creados con *coroutines* brindan mayor flexibilidad a la hora de definir la forma de cómo realizar el encadenamiento ya que la misma puede ser secuencial, ramificada o en forma de emisión (broadcasting).

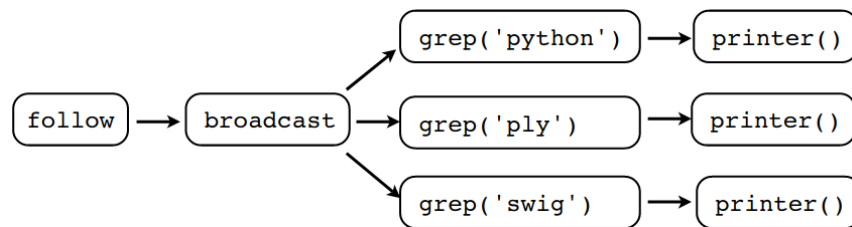
Secuencial



Ramificada (branch)



Broadcasting



Tener en cuenta que en todos los casos se requiere de un productor de los datos como primer paso del [pipeline](#).

El esqueleto para una [coroutine](#) utilizada en un [pipeline](#) será:

```
def mi_coroutine(target):
    try:
        while True:
            item = (yield)      # Recibe un elemento
            ...
            target.send(some_value) # envía datos a la próxima coroutine

    except GeneratorExit:
        # Tareas de finalización
```

Veamos un ejemplo:

Supongamos que tenemos las siguientes [coroutines](#), `grep_coroutine` y `print_coroutine`:

- `grep_coroutine`: recibe un texto y si el mismo cumple con determinado patrón de búsqueda lo envía a la próxima coroutine.
- `print_coroutine`: recibe e imprime por pantalla un texto.

```
def grep_coroutine(target, pattern):
    print("Buscando", pattern)
    while True:
        line = (yield)
```

```

        if pattern in line:
            target.send(line)

def print_coroutine():
    while True:
        line = (yield)
        print(line)

# Creamos las coroutines
p = print_coroutine()
next(p)
gc = grep_coroutine('python', p)
next(gc)

# procesamos el archivo "mi_archivo.txt", imprimiendo sólo aquellas
líneas que contienen el patrón python
for l in open("mi_archivo.txt"):
    gc.send(l)

```

Bonus track

Podemos usar un [decorator](#) para no tener que estar realizando la invocación de la instrucción [next](#) por cada [coroutine](#).

```

def coroutine(func):
    def start(*args, **kwargs):
        cr = func(*args, **kwargs)
        next(cr)
        return cr
    return start

@coroutine
def grep_coroutine(target, pattern):

```

```

...

@coroutine
def print_coroutine():
    ...

gc = grep_coroutine('python', print_coroutine())
for l in open("mi_archivo.txt"):
    gc.send(l)

```

Broadcasting

Para realizar un broadcast nuestra *coroutine*, debe recibir un listado de *coroutines*.

Ejemplo:

```

@coroutine
def broadcast(targets):
    while True:
        item = (yield)
        for target in targets:
            target.send(item)

gc1 = grep_coroutine('python', print_coroutine())
gc2 = grep_coroutine('C/C++', print_coroutine())
bc = broadcast([gc1, gc2])
for l in open("mi_archivo.txt"):
    bc.send(l)

```



Unidad 7

Administradores de dependencias y Entornos virtuales

Entornos Virtuales

Definición

Lo primero que debemos responder es ¿Qué es un entorno virtual?

Podemos definir un entorno virtual como un conjunto de directorios que contienen algunos scripts. Estos directorios y scripts proporcionan:

- Una versión específica de Python, por ejemplo Python 3.9.
- Una carpeta con bibliotecas de terceros que hemos o iremos instalando.

El objetivo principal de los entornos virtuales es proporcionar entornos aislados para cada uno de los distintos proyectos en los que estemos trabajando. Esto permite que cada proyecto pueda tener sus propias dependencias y versión de Python, independientemente de las dependencias que tengan los demás proyectos o el sistema operativo.

Beneficios

Imaginemos que estamos trabajando en dos proyectos, dos aplicaciones distintas o que por lo menos requieren versiones distintas de alguna de las bibliotecas a utilizar. Supongamos que en ambos debemos utilizar el paquete *request*, pero por diferentes motivos en uno requerimos la versión 2.24 mientras que en el otro requerimos la versión 2.25. Si no utilizáramos entornos virtuales, no podríamos utilizar las diferentes versiones del paquete.

Por lo tanto, para poder salvar dicha situación podemos utilizar los entornos virtuales.

En resumen, los beneficios de utilizar entornos virtuales son:

- Podemos tener varios entornos, con distintos conjuntos de paquetes, sin conflictos entre ellos y satisfacer de esta manera los requisitos de diferentes proyectos al mismo tiempo.
- Podemos liberar nuestros proyectos, cada uno con sus propias dependencias.

Usando entornos virtuales

Para utilizar entornos virtuales, a partir de la versión 3.x de Python, contamos con el módulo *venv*. Si estamos utilizando alguna versión anterior podemos utilizar el paquete *virtualenv*.

Para crear un entorno virtual, debemos seguir los siguientes pasos:

```
# Creamos un directorio para nuestro proyecto
usuario@pc:~$ mkdir mi_proyecto
# ingresamos al directorio
usuario@pc:~$ cd mi_proyecto/
# creamos el entorno virtual
usuario@pc:~$ python3 -m venv env38
```

Hay que tener en cuenta que la creación de los entornos virtuales depende de las versiones de Python que tengamos previamente instaladas en nuestro sistema. Por ejemplo, si quisiéramos crear un entorno virtual con la versión 3.6, además de tener dicha versión instalada, debemos ejecutar:

```
# creamos el entorno virtual
usuario@pc:~$ python3.6 -m venv env36
```

En el ejemplo anterior, creamos un (directorio) entorno virtual llamado env38. Este directorio posee la siguiente estructura:

```
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── easy_install
│   ├── easy_install-3.8
│   ├── pip
│   ├── pip3
│   ├── pip3.8
│   ├── python -> python3.8
│   ├── python3 -> python3.8
│   └── python3.5 -> /Library/Frameworks/Python.framework/Versions/3.8/bin/python3.8
├── include
├── lib
│   └── python3.8
│       └── site-packages
└── pyvenv.cfg
```

Estos directorios contiene lo siguiente:

- **bin**: archivos que interactúan con el entorno virtual
- **include**: encabezados C que compilan los paquetes de Python
- **lib**: una copia de la versión de Python junto con un directorio de paquetes donde se instalan las dependencias

También encontramos copias o enlaces simbólicos a diferentes herramientas de Python, así como a los propios ejecutables de Python. Todo esto garantiza que todo el código y los comandos de Python se ejecuten dentro del contexto del entorno actual, logrando de esta forma aislarse del resto de los proyectos o de los paquetes globales del sistema.

Activar un entorno virtual

Para que nuestro proyecto finalmente utilice el entorno virtual que acabamos de crear, debemos activarlo. Para esto utilizaremos la siguiente instrucción:

```
usuario@pc:~$ source env38/bin/activate
(env38) usuario@pc:~$
```

Cada vez que activemos un entorno virtual, veremos que el prompt de la línea de comando cambia, agregando al principio el nombre del entorno virtual activo.

Una vez que activamos un entorno virtual, todos los paquetes que instalemos se instalarán dentro del directorio [env38/lib/python3.8/site-packages](#) sin afectar el resto de los proyectos o los paquetes del sistema.

Desactivar un entorno virtual

Para desactivar un entorno virtual, simplemente ejecutaremos el comando [deactivate](#).

```
(env38) usuario@pc:~$ deactivate
usuario@pc:~$
```

Administradores de dependencias

Los administradores de dependencias, son herramientas que nos permiten gestionar las dependencias de nuestro proyecto y en algunos casos crear nuestros propios paquetes.

Dentro de los más conocidos tenemos:

- pip
- pipenv
- poetry
- conda

pip

Quizás el más simple y sencillo de utilizar. Por lo general al instalar Python, también instalaremos pip.

Sus opciones más comunes son:

- `search <nombre_paquete>`: permite buscar paquetes en PyPi
- `show <nombre_paquete>`: muestra información sobre el paquete. El paquete debe estar instalado.
- `list`: muestra un listado de los paquetes instalados.
- `install <nombre_paquete>`: permite instalar la última versión de un paquete
 - `install <nombre_paquete>=<versión>`: permite instalar una versión particular.
 - `install --upgrade <nombre_paquete>`: permite actualizar un paquete
 - `install -r requirements.txt`: permite instalar todos los paquetes especificados en el archivo requirements.txt.
- `uninstall <nombre_paquete>`: desinstala un paquete
- `freeze`: genera un listado de los paquetes instalados. Es útil si queremos generar el archivo requirements.txt
 - `pip freeze >> requirements.txt`

La desventaja principal, frente al resto de los administradores de paquetes, es que debemos manejar los entornos virtuales por separado y que no proporciona comandos simples para la construcción de nuestros propios paquetes.

Para más información se puede consultar el sitio oficial: [Pip](#)

pipenv

Permite crear y administrar automáticamente entornos virtuales. Gestiona los paquetes a partir del archivo Pipfile en el cual agrega/elimina paquetes a medida que los instala/desinstala.

También genera un archivo llamado Pipfile.lock, con información de los paquetes instalados.

Para instalar pipenv, la forma más sencilla es utilizando pip:

```
usuario@pc:~$ pip install pipenv
```

Para instalar un paquete, podemos utilizar el comando:

```
usuario@pc:~$ pipenv install <nombre_paquete>
# podemos instalar una versión particular de un paquete
usuario@pc:~$ pipenv install <nombre_paquete>=<version>
```

Cabe destacar que pipenv creará de forma automática, en caso de que no exista, un entorno virtual para el proyecto.

Cuando instalemos un paquete, se generarán los archivos Pipfile y Pipfile.lock, donde se especificarán los paquetes instalados y sus dependencias.

Pipenv nos permite gestionar por separado los paquetes que necesita nuestra aplicación para funcionar, de aquellos que sólo serán utilizados para desarrollo.

Para instalar un paquete sólo para desarrollo, podemos utilizar el comando:

```
usuario@pc:~$ pipenv install <nombre_paquete> --dev
```

Luego cuando estemos en un entorno productivo sólo ejecutaremos

```
# solo instalamos paquetes productivos
usuario@pc:~$ pipenv install
```

Y en caso de querer instalar también los paquetes utilizados para desarrollo:

```
usuario@pc:~$ pipenv install --dev
```

Pipenv posee otros comandos útiles como:

- run: permite ejecutar un comando o script dentro del entorno virtual sin necesidad de activarlo previamente.
- shell: permite activar el shell de python.
- update <nombre_paquete> para actualizar la versión de un paquete
- uninstall <nombre_paquete> para desinstalar un paquete

Para más información se puede consultar el sitio oficial: [Pipenv](#)

poetry

De la documentación oficial, la descripción nos dice: “Poetry es una herramienta para la gestión de dependencias y el empaquetado en Python. Permite declarar las bibliotecas de las que depende su proyecto y las administrará (instalará / actualizará) por usted.”

Al igual que con pipenv, la forma más sencilla de instalar *poetry* es utilizando pip

```
usuario@pc:~$ pip install poetry
```

Poetry nos permite, además de la gestión de paquetes, iniciar un proyecto, creando el directorio de trabajo. Para ello podemos ejecutar el comando:

```
usuario@pc:~$ poetry new mi_proyecto

# Esto genera un directorio con la siguiente estructura:
mi_proyecto
├── README.rst
├── mi_proyecto
│   └── __init__.py
├── pyproject.toml
└── tests
    ├── __init__.py
    └── test_mi_proyecto.py
```

El archivo **pyproject.toml** es el utilizado por **poetry** para gestionar las dependencias de paquetes.

Manejando dependencias

El comando **poetry init** nos permite crear un archivo **pyproject.toml** a partir de ciertas preguntas que debemos responder para la configuración particular del proyecto.

```
usuario@pc:~$ poetry init
```

Luego para agregar dependencias a nuestro proyecto, podemos utilizar los siguientes comandos:

```
usuario@pc:~$ poetry add <nombre_paquete>
# podemos instalar una versión exclusiva para desarrollo
usuario@pc:~$ poetry add -D <nombre_paquete>
```

Para poder instalar los paquetes especificados en el archivo **pyproject.toml** debemos utilizar el comando **poetry install**.

Para actualizar nuestras dependencias, debemos utilizar el comando:

```
usuario@pc:~$ poetry update
```


Este comando actualiza todas las dependencias del proyecto.

Generando un paquete

Para generar nuestros propios paquetes a partir de nuestro proyecto utilizaremos el comando **build**.

```
usuario@pc:~$ poetry build
```

El comando compilará y creará los archivos **wheels** como resultado de la construcción del proyecto. Estos archivos serán generados dentro del directorio **dist**.



Para más información se puede consultar el sitio oficial: [Poetry](#)

Bibliografía

- Unidad 1. Conceptos básicos. Disponible en:
 - <https://www.pythontutorial.net/>
 - <https://www.pythoncheatsheet.org/>
- Unidad 2. Diseño orientado a objetos. Disponible en:
 - <https://www.pythontutorial.net/python-oop/>
- Unidad 3. Patrones de diseño. Disponible en:
 - <https://sourcecmaking.com/>
 - <https://refactoring.guru/es>
- Unidad 4. Generators. Disponible en:
 - <https://realpython.com/introduction-to-python-generators/>
- Unidad 5. Threads. Disponible en:
 - <https://realpython.com/intro-to-python-threading/>
 - <https://tutorialedge.net/python/concurrency/python-threadpoolexecutor-tutorial/>
 - <https://www.geeksforgeeks.org/joining-threads-in-python/>
- Unidad 6. Coroutines. Disponible en:
 - <http://www.dabeaz.com/coroutines/Coroutines.pdf>
 - <https://medium.com/analytics-vidhya/python-generators-coroutines-async-io-with-examples-28771b586578>
 - <https://www.geeksforgeeks.org/coroutine-in-python/>
- Unidad 7. Administradores de dependencia. Disponible en:
 - <https://ahmed-nafies.medium.com/pip-pipenv-poetry-or-conda-7d2398adbac9>
 - <https://pipenv.pypa.io/en/latest/>
 - <https://python-poetry.org/>
 - <https://www.pythoncheatsheet.org/blog/python-projects-with-poetry-and-vscode-part-1/>
 - <https://www.pythoncheatsheet.org/blog/python-projects-with-poetry-and-vscode-part-2/>
 - <https://www.pythoncheatsheet.org/blog/python-projects-with-poetry-and-vscode-part-3/>