

Manual de uso de **qbank**

Marcos Bujosa

22 de octubre de 2026

Índice

1. Introducción	3
1.1. Instalación rápida (para quien ya conoce Python)	3
2. Instalación paso a paso	3
2.1. Requisitos previos	3
2.2. Qué es un entorno virtual y por qué usarlo	4
2.3. Crear y activar un entorno virtual	4
2.4. Instalar <code>calcpop-qbank</code>	4
2.5. Nota para usuarios de Windows	5
2.6. Crear un banco de preguntas	5
3. Conceptos fundamentales	5
3.1. Fórmulas con <code>calcpop</code>	5
3.2. Supuesto — una hipótesis del enunciado	6
3.3. Cuestion — un ítem de respuesta	6
3.4. <code>ProblemaTipo</code> — un generador de variantes	7
3.5. Preguntas paramétricas con <code>setup</code>	9
4. El formato JSON	11
4.1. Campos	11
4.2. De lista de listas a JSON y vuelta	12
4.3. Un ejemplo con <code>setup</code> en JSON	12
4.4. Exportar la lista de listas a un fichero <code>.py</code> editable	13
4.5. Los ejemplos de <code>ejemplosManual/</code>	14
5. Exportar a AMC (<code>L^AT_EX</code>)	14
5.1. <code>AMCblock</code>	14
5.2. <code>AMC_multipart</code> — preguntas con varias partes	15
5.3. <code>QuizAMCProfe</code> — la vista profe en un fichero	16
5.4. Integración en el documento AMC	16
6. Exportar a Moodle (vía <code>L^AT_EX</code>)	16
6.1. Funciones	17
6.2. Ejemplo	17
6.3. Preguntas con varias partes	17
6.4. Subir y actualizar el banco en Moodle desde el terminal	18
7. La vista de lectura (<code>QuizLectura</code>)	19
8. Comprobación automática de preguntas (<code>lint</code>)	20

9. Referencia rápida de la API	22
9.1. Clases	22
9.2. Funciones JSON	22
9.3. Funciones AMC	23
9.4. Funciones Moodle	23
9.5. Vista de lectura	23
9.6. Comprobación automática	23
9.7. Proyecto de banco	23
9.8. Operadores de <code>calcprop</code> (disponibles tras <code>from qbank import *</code>)	23
10. Preguntas frecuentes	24
10.1. ¿Por qué algunas variantes no se generan?	24
10.2. ¿Cómo incluyo texto $\text{\LaTeX}$ en los enunciados?	24
10.3. ¿Puedo escribir un porcentaje, grados, comillas...?	24
10.4. ¿Cómo escribo <code>display math</code> ?	24
10.5. ¿Puedo incluir una figura en el enunciado?	24
10.6. ¿Para qué sirve <code>codchar</code> ?	25
10.7. ¿Cómo exporto varias preguntas a un mismo cuestionario Moodle?	25
11. Apéndice: API avanzada y compatibilidad	25
11.1. <code>ProblemaVF</code> — preguntas de verdadero/falso	25
11.2. Clases deprecadas	26
11.3. Editor visual en Jupyter (<code>ProblemaTipoEditor</code>)	26

1. Introducción

`qbank` es una librería Python para generar bancos de preguntas de opción múltiple cuyas respuestas correctas se determinan automáticamente mediante cálculo proposicional. A partir de una lista de supuestos y cuestiones, genera todas las variantes posibles de una pregunta combinando los elementos alternativos; para cada variante calcula qué respuestas son correctas y cuáles no, y exporta el resultado a $\text{\LaTeX}$ /AMC (exámenes en papel) o a Moodle. Se apoya en `calcprop`, el motor de cálculo proposicional.

Este documento es la **referencia**: la API, el formato JSON, los exportadores y el comprobador `lint`, con sus firmas exactas. No es el sitio por donde empezar:

- `README.org` explica cómo instalar y cómo crear un banco con `qbank-init`.
- `Tutorial.ipynb` recorre cinco ejemplos progresivos ejecutables.
- La *Guía de diseño* (`GuiaDiseno.org`, impresa en `GuiaDiseno.pdf`) explica cómo diseñar buenas preguntas: la mecánica de listas y mundos, el patrón recomendado, la redacción y las trampas de $\text{\LaTeX}$ y Moodle.

Los ejemplos ejecutables de este manual cargan los JSON de la carpeta `ejemplosManual/`, que está en el repositorio pero no viene con `pip install`. Para ejecutarlos, clona el repositorio (`git clone https://github.com/mbujosab/calcprop-qbank`) y lánzalos desde su carpeta raíz. Para crear tu banco no hace falta: `qbank-init` trae sus propias preguntas de ejemplo.

El vocabulario que usa este manual es el de la Guía: **variante** (cada versión concreta de la pregunta), **mundo** (las hipótesis activas en una variante), **hueco** (cada posición que se convierte en una opción: una *Cuestion* suelta o una lista de la que se sortea una), **parte** (cada bloque de opciones de una pregunta con varios), **opción por defecto** («Ninguna de las anteriores», que añade `export.last_choice`) y **vista profe** (todas las cuestiones de cada hueco con sus marcas).

1.1. Instalación rápida (para quien ya conoce Python)

```
pip install calcprop-qbank
```

`qbank-init` y `lint` llegan con la 0.6.0; si tenías una versión anterior, actualiza con `pip install -U calcprop-qbank`. La versión en desarrollo se instala desde el repositorio (hace falta `git`):

```
pip install "calcprop-qbank[jupyter]" @ git+https://github.com/mbujosab/calcprop-qbank"
```

Si no estás familiarizado con Python o los entornos virtuales, lee la sección siguiente.

2. Instalación paso a paso

2.1. Requisitos previos

`qbank` requiere Python 3.9 o superior. Comprueba si ya lo tienes instalado:

Sistema	Comando en terminal
Linux	<code>python3 --version</code>
macOS	<code>python3 --version</code>
Windows	<code>py --version</code>

Si el comando devuelve algo como `Python 3.11.2` estás listo. Si no, descarga Python desde python.org e instálalo (en Windows marca la opción *Add Python to PATH*).

2.2. Qué es un entorno virtual y por qué usarlo

Un entorno virtual es una carpeta que contiene una copia aislada de Python con sus propios paquetes instalados, independiente del resto del sistema. Esto evita conflictos entre proyectos que requieren versiones distintas de las mismas librerías.

Trabajar siempre dentro de un entorno virtual es una buena práctica, aunque no es estrictamente obligatorio para un uso sencillo de qbank.

2.3. Crear y activar un entorno virtual

Abre una terminal (en Windows: *Símbolo del sistema* o *PowerShell*) y sitúate en el directorio donde quieres trabajar, por ejemplo:

```
mkdir mis_cuestionarios
cd mis_cuestionarios
```

Crea el entorno virtual (llámalo como quieras; aquí usamos `.venv`):

Sistema	Comando
Linux / macOS	<code>python3 -m venv .venv</code>
Windows (cmd)	<code>py -m venv .venv</code>
Windows (PS)	<code>py -m venv .venv</code>

Actívalo:

Sistema	Comando
Linux / macOS	<code>source .venv/bin/activate</code>
Windows (cmd)	<code>.venv\Scripts\activate.bat</code>
Windows (PowerShell)	<code>.venv\Scripts\Activate.ps1</code>

Cuando el entorno está activo, verás su nombre entre paréntesis al comienzo del prompt, por ejemplo `(.venv)`. A partir de ese momento, `pip` y `python` apuntan al entorno virtual y no al sistema.

2.4. Instalar `calcprop-qbank`

Con el entorno activo:

```
pip install calcprop-qbank
```

o, para la versión en desarrollo (hace falta tener `git`):

```
pip install "calcprop-qbank[jupyter] @ git+https://github.com/mbujosab/calcprop-qbank"
```

Verifica la instalación:

```
python -c "import qbank; print('qbank', qbank.__version__)"
qbank-init --help
```

Si ves `qbank 0.6.0` (o posterior) y la ayuda de `qbank-init`, todo está en orden. Si en cambio sale `AttributeError: ... no attribute '__version__'` o `qbank-init` no existe, tienes una versión anterior a la 0.6: actualiza con `pip install -U calcprop-qbank`. Para los PDF y el XML de Moodle hace falta además `pdflatex` con los paquetes `automultiplechoice` (lo instala el programa [Auto Multiple Choice](#)) y `moodle` (`tlmgr install moodle` en `TEX Live`).

2.5. Nota para usuarios de Windows

En Windows, dentro del entorno virtual, usa `python` (no `python3`). Si al activarlo en PowerShell (`.venv\Scripts\Activate.ps1`) sale un error de permisos, ejecuta antes:

```
Set-ExecutionPolicy -Scope CurrentUser RemoteSigned
```

2.6. Crear un banco de preguntas

La forma recomendada de trabajar es un proyecto creado con `qbank-init`:

```
qbank-init mi-banco
cd mi-banco
make lint          # comprueba las preguntas de ejemplo (sin LaTeX)
make amc           # PDF de revisión (vista profe) en build/amc/
```

El proyecto contiene cinco preguntas de ejemplo, los scripts que las exportan, un `Makefile`, `nueva_pregunta.py` (un guion para escribir una pregunta en Python y guardarla como JSON) y, en `prompts/`, instrucciones para escribir preguntas con ayuda de una IA. Su [README.org](#) explica el flujo completo. El resto de este manual describe las piezas que esos scripts usan.

3. Conceptos fundamentales

3.1. Fórmulas con `calccprop`

`qbank` importa `calccprop`, así que al escribir `from qbank import *` tienes acceso al motor lógico. Las variables proposicionales (en la Guía, *etiquetas*) se crean con `v()` y los nombres los eliges tú; solo tienen que ser consistentes dentro de la misma pregunta:

```
from qbank import *

A = v('A')    # variable proposicional A
```

Las fórmulas se construyen combinando variables con los operadores de Python:

Operador Python	Conectiva lógica	Ejemplo
<code>-v('A')</code>	Negación ($\neg A$)	<code>-v('A')</code>
<code>v('A') & v('B')</code>	Conjunción ($A \wedge B$)	<code>v('A') & v('B')</code>
<code>v('A') v('B')</code>	Disyunción ($A \vee B$)	<code>v('A') v('B')</code>
<code>v('A') >> v('B')</code>	Implicación ($A \implies B$)	<code>v('A') >> v('B')</code>
<code>v('A') ** v('B')</code>	Bicondicional ($A \iff B$)	<code>v('A') ** v('B')</code>
<code>alguno(v('A'), v('B'), v('C'))</code>	Alguna verdadera	<code>alguno(v('A'), v('B'), v('C'))</code>
<code>unoDe(v('A'), v('B'), v('C'))</code>	Exactamente una verdadera	<code>unoDe(v('A'), v('B'), v('C'))</code>

La semántica también acepta `True` (siempre verdadero) o `False` (siempre falso). `alguno` y `unoDe` aceptan cualquier número de proposiciones.

Atención — precedencia de operadores: la precedencia de Python difiere de la lógica matemática. En lógica, $\rightarrow$ tiene menor precedencia que $\wedge$ y $\vee$, por lo que $A \wedge B \rightarrow C$ se lee como $(A \wedge B) \rightarrow C$. En Python, `>>` tiene **mayor** precedencia que `&` y `|`, así que `v('A') & v('B') >> v('C')` se evalúa como `v('A') & (v('B') >> v('C'))`, que no es lo que se espera. Usa siempre paréntesis explícitos en implicaciones compuestas:

```
(v('A') & v('B')) >> v('C')    # correcto: (A ∧ B) → C
v('A') & v('B') >> v('C')      # incorrecto: A ∧ (B → C)
```

La función `test(formula, hipotesis)` comprueba si una fórmula es consecuencia lógica de una lista de fórmulas hipótesis:

```
test(v('C'), [v('A'), v('A') >> v('C')])    # True (modus ponens)
test(v('C'), [v('B')])                      # False (B no implica C)
test(True, [])                              # True
test(False, [])                             # False
```

No necesitas llamar a `test` directamente: `qbank` lo invoca al generar las variantes. Conviene tener presente lo que responde: «¿se deduce esto de las hipótesis?», no «¿es verdad?». De la hipótesis `v('crec')` no se sigue `-v('decr')`, salvo que un axioma declare que ambas son excluyentes (`unoDe(v('crec'), v('decr'))`).

3.2. Supuesto — una hipótesis del enunciado

Un **Supuesto** es una afirmación que se añade al enunciado del problema y cuya semántica se incorpora como hipótesis para evaluar las cuestiones que le siguen.

```
Supuesto(enunciado, semantica, precondition=True)
```

Parámetro	Tipo	Descripción
enunciado	cadena	Texto $\text{\LaTeX}$ del supuesto, tal como aparecerá impreso. Admite <code>@{variable}</code> (ver <code>setup</code>).
semantica	fórmula	Fórmula proposicional que el supuesto afirma verdadera.
precond	fórmula/bool	El supuesto solo se activa si esta precondition se cumple bajo las hipótesis ya acumuladas. Por defecto <code>True</code> (siempre activo).

Un **Supuesto** con **enunciado** vacío es un **axioma**: añade su fórmula sin texto.

```
Supuesto('$\mathcal{A}$ es verdadero. ', v('A'))
Supuesto('$\mathcal{A} \rightarrow \mathcal{C}$ ', v('A') >> v('C'))
Supuesto('', unoDe(v('inv'), v('sing')))          # axioma: los dos mundos son excluyentes
```

Si la precondition de un supuesto no se cumple en una variante concreta, esa variante se descarta.

3.3. Cuestion — un ítem de respuesta

Una **Cuestion** es una posible respuesta. `qbank` determina si es correcta o no según las hipótesis activas en cada variante.

```
Cuestion(enunciado, semantica, precondition=True, exp="")
```

Parámetro	Tipo	Descripción
enunciado	cadena	Texto $\text{\LaTeX}$ del ítem de respuesta. Admite <code>@{variable}</code> .
semantica	fórmula/bool	Fórmula cuya veracidad (como consecuencia de las hipótesis) determina si el ítem es correcto.
precond	fórmula/bool	La cuestión solo aparece si esta precondition se satisface; si no, la variante se descarta. Por defecto <code>True</code> .
exp	cadena	Explicación (retroalimentación) que Moodle muestra tras responder. Admite <code>@{variable}</code> . Opcional.

```
Cuestion("Entonces $\mathcal{C}$ es verdadero", v("C"))
Cuestion("Esta afirmación es siempre falsa", False)
Cuestion("$\mathcal{B}$ es consecuencia de $\mathcal{A}$", v("B"), precondition=v("A"))
Cuestion("$f(@{a}) < f(@{b})$", v("crec"), exp="Porque $@{a} < @{b}$ y $$$ es creciente.")
```

Semánticas y precondiciones como cadenas. Tanto `Supuesto` como `Cuestion` aceptan semántica y precond escritas como la cadena que las representa (`"v('A') >> v('B')"`, `"True"`, `"lambda ns: ns['x'] > 0"`), que es exactamente lo que guarda el formato JSON. Es la forma de escribir una lambda en Python si la pregunta se va a guardar con `save_problema`: una función Python no se puede serializar; su cadena, sí.

3.4. ProblemaTipo — un generador de variantes

`ProblemaTipo` toma una lista plana de componentes y genera todas las combinaciones posibles, una variante por cada combinación.

```
ProblemaTipo(componentes, setup=None, export=None, seed=None, nombre=None)
```

Parámetro	Descripción
<code>componentes</code>	Lista plana de cadenas (texto fijo), <code>Supuesto</code> y <code>Cuestion</code> , sueltos o en listas de alternativas (ver abajo).
<code>setup</code>	Código Python como cadena (lo que guarda el JSON) o función sin argumentos que devuelve un dict. Se ejecuta antes de cada variante.
<code>export</code>	Dict con las preferencias de exportación: <code>last_choice</code> , <code>cols</code> , <code>instances</code> . Lo leen los scripts de un banco, no las funciones.
<code>seed</code>	Entero que siembra <code>numpy.random</code> y <code>random</code> antes de recorrer las variantes (solo tiene sentido con <code>setup</code> aleatorio).
<code>nombre</code>	Identificador del problema (el campo <code>nombre</code> del JSON; debe coincidir con el nombre del fichero).

Las funciones de exportación no leen `export`: pasa tú `last_choice=`, `instances=` y `cols=` a `AMCblock`, `QuizMoodle`, `QuizAMCProfe` y las demás (los ejemplos de este manual lo hacen). Los scripts de un banco de `qbank-init` sí lo leen de cada JSON.

3.4.1. La mecánica: listas = sorteo, elementos consecutivos = opciones

Cada componente es, o bien un valor suelto (siempre se incluye), o bien una lista de dos o más valores del mismo tipo, de la que **se sortea uno** en cada variante. El número de variantes es el producto de las longitudes de las listas.

- Una lista de `Supuesto` son **mundos alternativos**: se activa uno por variante.
- Una lista de `Cuestion` **no** son «las opciones que se muestran juntas»: es un **hueco** del que se sortea una. El alumno ve tantas opciones como elementos-`Cuestion` consecutivos haya (sueitos o listas), más la opción por defecto si `export.last_choice` está activo.
- Cuando tras un bloque de cuestiones vuelve a aparecer texto o un `Supuesto`, se abre una **parte** nueva (una pregunta con varios bloques de opciones). Las hipótesis se acumulan a lo largo de las partes.

El patrón recomendado es tres huecos + `"export": {"last_choice": true}`: cuatro opciones, con la garantía de que al menos una es correcta y al menos una incorrecta. La Guía desarrolla todo esto; aquí basta un ejemplo completo.

3.4.2. Ejemplo

```
from qbank import *

p = ProblemaTipo(
    [
```

```

"Considere ",
[ # dos mundos: se sortea uno
  Supuesto("$\\mathcal{A}$ ", v("A")),
  Supuesto("$\\mathcal{B}$ ", v("B")),
],
[ # y otros dos, independientes de los anteriores
  Supuesto("y que $\\mathcal{A}\\rightarrow\\mathcal{C}$. ", v("A") >> v("C")),
  Supuesto("y que $\\mathcal{B}\\rightarrow\\mathcal{D}$. ", v("B") ** v("D")),
],
"Entonces, con toda seguridad: ",
[ # hueco 1
  Cuestion("se da $\\mathcal{C}$", v("C")),
  Cuestion("se da $\\mathcal{D}$", v("D")),
],
[ # hueco 2
  Cuestion("se da $\\mathcal{A}$", v("A")),
  Cuestion("se da $\\mathcal{B}$", v("B")),
],
[ # hueco 3
  Cuestion("se dan $\\mathcal{A}$ y $\\mathcal{C}$", v("A") & v("C")),
  Cuestion("se dan $\\mathcal{B}$ y $\\mathcal{D}$", v("B") & v("D")),
  Cuestion("se da $\\mathcal{E}$", False),
],
],
export={"last_choice": True, "cols": 2},
nombre="ejercicio",
)

```

```

for etiqueta, partes in p.por_partes():
    for enunciado, cuestiones in partes:
        print(f"Variante {etiqueta}: {enunciado}")
        for texto, correcto, activa, exp in cuestiones:
            print(f" {'✓' if correcto else '×'} {texto}")
        if int(etiqueta) == 3:
            break
print("...", len(list(p.por_partes()))), "variantes en total")

```

Variante 1: Considere $\mathcal{A}$ y que $\mathcal{A} \rightarrow \mathcal{C}$. Entonces, con
 $\hookrightarrow$ toda seguridad:

- ✓ se da $\mathcal{C}$
- ✓ se da $\mathcal{A}$
- ✓ se dan $\mathcal{A}$ y $\mathcal{C}$

Variante 2: Considere $\mathcal{A}$ y que $\mathcal{A} \rightarrow \mathcal{C}$. Entonces, con
 $\hookrightarrow$ toda seguridad:

- ✓ se da $\mathcal{C}$
- ✓ se da $\mathcal{A}$
- × se dan $\mathcal{B}$ y $\mathcal{D}$

Variante 3: Considere $\mathcal{A}$ y que $\mathcal{A} \rightarrow \mathcal{C}$. Entonces, con
 $\hookrightarrow$ toda seguridad:

- ✓ se da $\mathcal{C}$
- ✓ se da $\mathcal{A}$
- × se da $\mathcal{E}$

... 48 variantes en total

$2 \times 2 \times 2 \times 2 \times 3 = 48$ variantes, cada una con tres opciones más «Ninguna de las anteriores» (que el exportador añade y que en la variante 1 será incorrecta). `ejemplosManual/ejercicio.json` guarda este mismo problema en JSON, con una explicación (exp) en cada cuestión que aquí se omite por brevedad.

3.4.3. Iterar las variantes

por_partes(instances=1, base_seed=None, verbose=False) la vía recomendada. Devuelve (etiqueta, [(enunciado, cuestiones), ...]), un par por parte; cuestiones es una lista de tuplas (texto, correcto, activa, exp). Con `setup` aleatorio, `instances=N` recorre las variantes N veces con semillas distintas (`base_seed + i * 997`; si no se pasa `base_seed` se usa `seed`, y si tampoco hay `seed`, 42). Con `instances=1` y sin ninguna de las dos no se siembra nada: el resultado depende del estado del generador. Con `verbose=True` imprime los supuestos y cuestiones rechazados por su precondition.

por_partes_profe(verbose=True) la vista **profe**: el mismo formato, pero con *todas* las cuestiones de cada hueco (no una por variante) y las rechazadas por su precondition marcadas como 'rechazada por ...' en vez de descartar la variante. Es lo que conviene revisar antes de dar una pregunta por buena.

for etiqueta, enunciado, cuestiones in p el protocolo histórico, solo para ejercicios de una parte (con varias lanza `ValueError`).

3.5. Preguntas paramétricas con `setup`

El parámetro `setup` permite que los valores numéricos o simbólicos cambien en cada variante. En cada iteración:

1. Se ejecuta el `setup` y sus variables forman el **espacio de nombres** `ns` de la variante.
2. Los textos (enunciados, cuestiones y `exp`) que contengan marcadores `@{variable}` reciben la sustitución.
3. Las semánticas y preconditiones que sean `lambda ns: ...` se evalúan con ese diccionario.

El delimitador `@` se eligió para evitar conflictos con el modo matemático de $\text{\LaTeX}$. Para una arroba literal, escribe `@@`.

3.5.1. `setup` como cadena, `np.random` y `seed`

Escribe el `setup` como cadena de código (es lo que guarda el JSON, y la única forma que `save_problema` puede serializar) y usa `np.random`, no `random`: `seed` siembra `numpy.random` y `random`, pero los scripts de la vista **profe** de un banco solo siembran `numpy`.

```
p = ProblemaTipo(
    [
        "Sean $a = @{a}$ y $b = @{b}$. Entonces:",
        [
            Cuestion("$a + b = @{suma}$", True),
            Cuestion("$a + b = @{suma_falsa}$", False, exp="$@{a} + @{b} = @{suma}$."),
        ],
        [
            Cuestion("$a + b > 10$", "lambda ns: ns['a'] + ns['b'] > 10"),
            Cuestion("$a + b < 10$", "lambda ns: ns['a'] + ns['b'] < 10"),
        ],
        Cuestion("$a \\cdot b = @{producto}$", True),
    ],
    setup="import numpy as np\n"
        "a, b = (int(x) for x in np.random.randint(1, 10, 2))\n"
        "suma, suma_falsa, producto = a + b, a + b + 1, a * b",
    seed=1,
    export={"last_choice": True, "instances": 15},
)
```

- `@{a}`, `@{b}`, `@{suma}` se interpolan con los valores de cada variante.
- Las lambdas, escritas como cadena, se evalúan con los valores de la variante.
- `seed=1` hace la pregunta reproducible; `instances=15` recorre 15 veces (15 instancias) las 4 combinaciones (60 variantes, el mínimo que recomienda `lint` con `setup`).

El iterador es finito aunque el `setup` sea aleatorio: recorre una vez cada combinación de alternativas y llama al `setup` en cada una. Para obtener más instancias se usa `instances`, no un bucle a mano.

3.5.2. Precondiciones que dependen de los valores

El motor razona de forma proposicional y **no conoce** los valores numéricos: si ofreciéramos los supuestos « $a > b$ » y « $a < b$ » sin más, podría generar una variante con $a = 3$, $b = 7$ y el texto «Se sabe que $a > b$ ». La solución es una **precondición** `ns: ...` que solo deja pasar la variante cuando la afirmación es cierta para los valores sorteados:

```
ordenar = ProblemaTipo(
    [
        "Sean $a = @{a}$ y $b = @{b}$ dos números naturales. Se sabe que ",
        [
            Supuesto("$a > b$. ", True, precondition="lambda ns: ns['a'] > ns['b']"),
            Supuesto("$a < b$. ", True, precondition="lambda ns: ns['a'] < ns['b']"),
            Supuesto("$a = b$. ", True, precondition="lambda ns: ns['a'] == ns['b']"),
        ],
        "Indique qué afirmación es correcta: ",
        [
            Cuestion("$a - b > 0$", "lambda ns: ns['a'] - ns['b'] > 0"),
            Cuestion("$b - a > 0$", "lambda ns: ns['b'] - ns['a'] > 0"),
        ],
        [
            Cuestion("$a^2 > b^2$", "lambda ns: ns['a'] ** 2 > ns['b'] ** 2"),
            Cuestion("$a^2 < b^2$", "lambda ns: ns['a'] ** 2 < ns['b'] ** 2"),
        ],
        Cuestion("$a \\cdot b > 0$", True),
    ],
    setup="import numpy as np\na, b = (int(x) for x in np.random.randint(2, 10, 2))",
    seed=7,
    export={"last_choice": True, "instances": 15},
)
```

Para cada combinación el `setup` genera una tirada nueva; si su precondición no se cumple, esa variante se descarta (con `verbose=True` verás, por ejemplo, el mensaje `Supuesto: ... rechazado por ...`). Por eso el número de variantes por instancia no es fijo: de las $3 \times 2 \times 2 = 12$ combinaciones sobreviven unas cinco por instancia (las del supuesto que coincide con la tirada), y `lint` calcula que hacen falta 14 instancias para llegar a 60 (con las 15 de `export` salen 69). El problema `ejemplosManual/numeros.json` (el que carga el editor visual del cuaderno `Demo.ipynb`) es una versión ampliada de esta idea.

Con precondiciones numéricas, `por_partes_profe()` hace **un solo sorteo** por mundo, y un mundo cuya precondición no se cumple en ese sorteo no aparece: con `seed=7`, `ordenar.por_partes_profe()` muestra los tres mundos, pero con otra semilla podría mostrar menos (con `seed=2`, ninguno). La vista `profe` de un banco (`make amc`) hace varios sorteos para no perder mundos.

3.5.3. seed: reproducibilidad independiente del orden

Cuando varios problemas con `setup` aleatorio se generan en secuencia (en un bucle o un `Makefile`), los valores sorteados en cada problema dependerían del estado del generador en ese momento. El campo `seed` (entero) resuelve esto: antes de iterar las variantes, `qbank` siembra `numpy.random` y `random` con él. Así cada problema tiene su propia secuencia reproducible. También es la semilla base de `por_partes(instances=N)` cuando no se pasa `base_seed`.

3.5.4. setup como función (solo uso interactivo)

`setup` admite también una función sin argumentos que devuelve el diccionario de variables. Es cómodo en un cuaderno, pero un problema así **no puede guardarse** con `save_problema` (el código de una función no es recuperable); lo mismo ocurre con las lambdas escritas como funciones Python en `semantica` o `precond`. Escríbelos como cadenas si la pregunta va a vivir en un fichero.

4. El formato JSON

`qbank` convierte cualquier `ProblemaTipo` a JSON y lo reconstruye después. Es el formato en que se guardan las preguntas de un banco (lo que leen `lint` y los scripts de `qbank-init`).

4.1. Campos

```
{
  "version": "1",
  "tipo": "ProblemaTipo",
  "nombre": "L-01-Monotonia",
  "seed": 1,
  "setup": "import numpy as np\nna = int(np.random.randint(1, 10))",
  "export": {"last_choice": true, "cols": 2, "instances": 6},
  "componentes": [
    "Texto fijo ",
    {"tipo": "Supuesto", "enunciado": "", "semantica": "unoDe(v('A'), v('B'))", "precond":
      ↪ "True"},
    [
      {"tipo": "Supuesto", "enunciado": "alternativa 1 ", "semantica": "v('A')", "precond":
        ↪ "True"},
      {"tipo": "Supuesto", "enunciado": "alternativa 2 ", "semantica": "v('B')", "precond":
        ↪ "True"}
    ],
    " Entonces:",
    [
      {"tipo": "Cuestion", "enunciado": "¿A?", "semantica": "v('A')", "precond": "True",
        ↪ "exp": ""},
      {"tipo": "Cuestion", "enunciado": "¿B?", "semantica": "v('B')", "precond": "True",
        ↪ "exp": ""}
    ],
    {"tipo": "Cuestion", "enunciado": "$a = @{a}$", "semantica": "True", "precond": "True",
      ↪ "exp": ""}
  ]
}
```

Clave	Descripción
version	"1".
tipo	"ProblemaTipo" (o "ProblemaVF", ver el apéndice).
nombre	Identificador; debe coincidir con el nombre del fichero sin .json y no contener : (rompe make).
seed	Entero, null o ausente. Solo con setup aleatorio.
setup	Código Python como cadena (saltos de línea como \n) o null.
export	Opcional. last_choice (bool): opción por defecto. cols (int): columnas en AMC. instances (int): instancias con setup aleatorio.
componentes	La lista plana: cadenas, objetos Supuesto y Cuestion y listas de ellos.

Cada Supuesto es un objeto con tipo, enunciado, semantica y precondition; cada Cuestion añade exp. Las fórmulas y lambdas van como cadenas, exactamente como se escribirían en Python. Las barras de L^AT_EX se doblan (\\mathcal).

Puedes añadir claves propias de nivel superior, como "_nivel": "M" o "_tema": qbank las ignora al cargar y save_problema las conserva.

Regla para instances: con setup aleatorio, instances = max(1, ceil(60 / variantes por instancia)), para que la vista alumno tenga al menos 60 variantes; lint imprime el valor necesario.

4.2. De lista de listas a JSON y vuelta

problema_to_dict devuelve el dict y save_problema lo escribe en un fichero; load_problema acepta una ruta o un dict y devuelve un ProblemaTipo listo para iterar.

```
import os, tempfile
ruta = os.path.join(tempfile.gettempdir(), "ordenar.json") # p. ej. preguntas/ordenar.json
save_problema(ordenar, ruta)
p2 = load_problema(ruta) # o load_problema(dict)
assert problema_to_dict(p2) == problema_to_dict(ordenar)
```

(El ejemplo guarda en un directorio temporal para no tocar los JSON de ejemplosManual/, que usan las secciones siguientes.)

Para bancos de varios problemas en un solo fichero, save_banco(lista, ruta) y load_banco(ruta); los proyectos de qbank-init usan en cambio un fichero por pregunta.

4.3. Un ejemplo con setup en JSON

El fichero ejemplosManual/matriz_rango.json:

```
{
  "version": "1",
  "tipo": "ProblemaTipo",
  "nombre": "matriz_rango",
  "seed": 7,
  "setup": "import numpy as np\nn = int(np.random.randint(3, 6))\nnk = int(np.random.randint(2,\n↪ n))",
  "export": {"last_choice": true, "cols": 2, "instances": 4},
  "componentes": [
    "Considere una matriz $A$ de $@{n}$ filas y $@{k}$ columnas. ",
    [
      {"tipo": "Supuesto", "enunciado": "Suponga que $A$ tiene rango máximo\n↪ ($\\mathrm{rg}(A)=@{k}$). ",
        "semantica": "v('MaxRk')", "precond": "True"},
      {"tipo": "Supuesto", "enunciado": "Suponga que $A$ tiene rango deficiente\n↪ ($\\mathrm{rg}(A)<@{k}$). ",
```

```

        "semantica": "-v('MaxRk')", "precond": "True"}
    ],
    "Indique qué afirmaciones son correctas: ",
    [
        {"tipo": "Cuestion", "enunciado": "Las ${k}$ columnas de $A$ son linealmente
        ↪ independientes.",
        "semantica": "v('MaxRk')", "precond": "True", "exp": "..."},
        {"tipo": "Cuestion", "enunciado": "Alguna columna de $A$ es combinación lineal de las
        ↪ demás.",
        "semantica": "-v('MaxRk')", "precond": "True", "exp": "..."}
    ],
    "... (otros dos huecos: $A^{\top}$ invertible / singular; $Ax = 0$ con solución única /
    ↪ infinitas)"
]
}

```

```

p = load_problema("ejemplosManual/matriz_rango.json")
for etiqueta, partes in p.por_partes():
    for enunciado, cuestiones in partes:
        print(f"Variante {etiqueta}: {enunciado}")
        for texto, correcto, activa, exp in cuestiones:
            print(f" {'✓' if correcto else '×'} {texto}")
    if int(etiqueta) == 2:
        break
print(len(list(p.por_partes()))), "variantes por instancia;",
      len(list(p.por_partes(instances=4))), "con instances=4"

```

Variante 1: Considere una matriz A de 3 filas y 2 columnas. Suponga que A tiene rango
 ↪ máximo ($\mathrm{rg}(A)=2$). Indique qué afirmaciones son correctas:

- ✓ Las 2 columnas de A son linealmente independientes.
- ✓ $A^{\top} A$ es invertible.
- ✓ El sistema $Ax = 0$ tiene solo la solución trivial.

Variante 2: Considere una matriz A de 4 filas y 2 columnas. Suponga que A tiene rango
 ↪ máximo ($\mathrm{rg}(A)=2$). Indique qué afirmaciones son correctas:

- ✓ Las 2 columnas de A son linealmente independientes.
- ✓ $A^{\top} A$ es invertible.
- × El sistema $Ax = 0$ tiene infinitas soluciones.

16 variantes por instancia; 64 con instances=4

La lógica proposicional y los valores numéricos actúan de forma independiente: $v('MaxRk')$ no «sabe» cuánto valen n y k ; su valor de verdad lo determina el supuesto elegido. El `setup` solo aporta los números que aparecen en el texto. El segundo supuesto afirma $-v('MaxRk')$ directamente, y por eso las cuestiones con esa semántica se deducen sin necesidad de un axioma unoDe.

4.4. Exportar la lista de listas a un fichero .py editable

`problema_to_python` devuelve el código Python que recrea el problema como listas editables, y `save_problema_py` lo escribe en un fichero. El resultado se ejecuta tal cual y puede volver a guardarse con `save_problema`: si el problema tiene `setup`, su código aparece en la cadena `_SETUP`, y las lambdas van entre comillas.

```

codigo = problema_to_python(load_problema("ejemplosManual/ejercicio.json"))
print("\n".join(codigo.splitlines()[:30])) # las primeras 30 líneas

```

```

from qbank import Supuesto, Cuestion, ProblemaTipo
from calcpop import *

ejercicio = [
    r'Considere ',
    [
        Supuesto(
            enunciado = r'$\mathcal{A}$ ',
            semantica = v('A'),
            precondition = True
        ),
        Supuesto(
            enunciado = r'$\mathcal{B}$ ',
            semantica = v('B'),
            precondition = True
        ),
    ],
    [
        Supuesto(
            enunciado = r'y que $\mathcal{A} \rightarrow \mathcal{C}$. ',
            semantica = v('A') >> v('C'),
            precondition = True
        ),
        Supuesto(
            enunciado = r'y que $\mathcal{B} \leftrightarrow \mathcal{D}$. ',
            semantica = v('B') ** v('D'),
            precondition = True
        ),
    ],
    r'Entonces, con toda seguridad: ',

```

4.5. Los ejemplos de ejemplosManual/

Cinco ficheros ilustran, de forma acumulativa, las mecánicas de este capítulo, y Tutorial.ipynb los recorre ejecutándolos: ej_a_literales (valores de verdad literales, tres huecos), ej_b_mundos (una lista de Supuesto), ej_c_precond (axioma unoDe y precondition por mundo), ej_d_setup_vars (setup, @{var}, seed, instances y exp dinámico) y ej_e_lambda (semántica lambda que inyecta fórmulas). Los otros cuatro (ejercicio, matriz_rango, numeros y superheroes) son los que usan este manual y Demo.ipynb. Todos pasan lint sin ERROR ni AVISO.

5. Exportar a AMC (L^AT_EX)

Las funciones AMC generan el código L^AT_EX para [Auto Multiple Choice](#): cada una devuelve una cadena con un bloque \element{...}, que el usuario escribe al fichero. El flujo completo (un .tex por pregunta, un documento que lo compila y los PDF de revisión) lo hacen los scripts de un proyecto qbank-init (make amc y make amc-student).

Las funciones no leen export del problema: pasa last_choice= y cols= en cada llamada (los scripts de un banco sí lo leen y se los pasan).

5.1. AMCblock

```

AMCblock(nombre, etiqueta, enunciado, cuestiones,
          last_choice=False, cols=1, profe=False, *,
          aux_latex="", last_choice_text="Ninguna de las anteriores")

```

Parámetro	Descripción
nombre	Nombre del grupo AMC (etiqueta $\text{\LaTeX}$). Los <code>:</code> y <code>_</code> se convierten en <code>-</code> .
etiqueta	Identificador único de esta variante dentro del grupo.
enunciado	Texto del enunciado ($\text{\$}\dots\text{\$}$ se convierte a $\text{\[}\dots\text{\]}$, que <code>automultiplechoice</code> prefiere).
cuestiones	Lista de tuplas (<code>texto</code> , <code>correcto</code> , <code>activa</code> , <code>exp</code>), la segunda componente de cada parte.
last_choice	Si <code>True</code> , añade la opción por defecto con <code>\lastchoices</code> : correcta solo si las demás son falsas.
cols	Columnas (>1 activa <code>\begin{multicols}{N}\AMCBoxedAnswers</code>).
profe	Si <code>True</code> , vista profe: las cuestiones rechazadas por su precondition van en <code>\explain{}</code> .
aux_latex	Instrucciones $\text{\LaTeX}$ extra dentro del bloque <code>\element{}</code> .
last_choice_text	Texto de la opción por defecto.

```
p = load_problema("ejemplosManual/ejercicio.json")
etiqueta, partes = next(iter(p.por_partes()))
enunciado, cuestiones = partes[0]
print(AMCblock("MiCuestionario", etiqueta, enunciado, cuestiones, last_choice=True, cols=2))
```

```
\element{MiCuestionario}{
\begin{questionmult}{MiCuestionario-1}
  Considere  $\mathcal{A}$  y que  $\mathcal{A} \rightarrow \mathcal{C}$ . Entonces, con toda
   $\rightarrow$  seguridad:
  \begin{multicols}{2}\AMCBoxedAnswers
    \begin{choices}
      \correctchoice{se da  $\mathcal{C}$ }
      \correctchoice{se da  $\mathcal{A}$ }
      \correctchoice{se dan  $\mathcal{A}$  y  $\mathcal{C}$ }
      \lastchoices
      \wrongchoice {Ninguna de las anteriores}
    \end{choices}
  \end{multicols}
\end{questionmult} }
```

Para escribir todas las variantes de la vista alumno:

```
with open("preguntas.tex", "w") as f:
    for etiqueta, partes in p.por_partes():
        enunciado, cuestiones = partes[0]
        f.write(AMCblock("MiCuestionario", etiqueta, enunciado, cuestiones, last_choice=True,
             $\rightarrow$  cols=2))
```

5.2. `AMC_multipart` — preguntas con varias partes

Un `AMCblock` por parte produciría dos preguntas con el mismo identificador y la segunda sin las hipótesis. `AMC_multipart` genera un único `questionmult` con un bloque de casillas por parte, cada uno con su propia opción por defecto si `last_choice`:

```
AMC_multipart(nombre, etiqueta, enunciado, partes,
               last_choice=False, cols=1, profe=False, *,
               aux_latex="", last_choice_text="Ninguna de las anteriores")
```

`partes` es la lista de pares (`enunciado`, `cuestiones`) que devuelven `por_partes()` y `por_partes_profe()`; `enunciado` es un texto común previo, normalmente "" porque el de la primera parte ya contiene las hipótesis. Los demás parámetros son los de `AMCblock`.

```

multi = load_problema("ejemplosManual/superheroes.json")      # tres partes
with open("preguntas_multi.tex", "w") as f:
    for etiqueta, partes in multi.por_partes():
        if len(partes) > 1:
            f.write(AMC_multipart("Superheroes", etiqueta, "", partes, last_choice=True))
        else:
            f.write(AMC_block("Superheroes", etiqueta, partes[0][0], partes[0][1],
                               ↪ last_choice=True))

```

5.3. QuizAMCProfe — la vista profe en un fichero

```

QuizAMCProfe(nombre, directorio, problema, cols=1, *, last_choice=False,
              aux_latex="", last_choice_text="Ninguna de las anteriores")
# Devuelve int: número de entradas escritas (una por mundo)

```

Recorre `por_partes_profe()` y escribe en `directorio/nombre_profe.tex` una pregunta AMC por mundo (con `AMC_multipart` si tiene varias partes), con todas las cuestiones de cada hueco marcadas y las rechazadas en `\explain{}`. `problema` puede ser un `ProblemaTipo` o un diccionario `{nombre: ProblemaTipo}`. `directorio` admite / final o no.

5.4. Integración en el documento AMC

Los ficheros generados se incluyen en el preámbulo del documento AMC con la orden `\input{preguntas.tex}`; dentro de `\onecopy`, `\insertgroup[1]{MiCuestionario}` inserta una variante por copia (sin el [1] se insertarían todas) y `\shufflegroup` las baraja. El README de la plantilla da el procedimiento completo; consulta la documentación de AMC para el formato del documento.

6. Exportar a Moodle (vía L^AT_EX)

Las funciones `Quiz*` escriben un fichero `.tex` autónomo que usa el paquete L^AT_EX `moodle`; al compilarlo se obtienen un PDF de revisión y el fichero `nombre-moodle.xml` importable en Moodle (*Banco de preguntas* → *Importar* → *Formato Moodle XML*). Como las de AMC, no leen `export:` pasa `last_choice=` e `instances=` (los scripts de un banco sí lo leen).

```

Python (qbank) → nombre.tex → xelatex / pdflatex → nombre.pdf + nombre-moodle.xml →
↪ Moodle

```

El `.tex` compila con cualquier motor (su cabecera distingue `pdflatex` de `xelatex` o `lualatex`). Si lo compilas **a mano**, usa `xelatex`: el XML sale declarado en UTF-8. Con `pdflatex` el paquete `moodle` lo declara en `iso-8859-1` aunque escribe en UTF-8 los caracteres que `codchar` no convierte (desde la 0.8.0 `codchar` cubre todas las letras del español, mayúsculas y ü incluidas, y los signos ¿ y ¡; quedan, por ejemplo, las comillas “ ”), y Moodle los mostraría mal (por ejemplo, `antigÃedad` en lugar de `antigüedad`); los proyectos de `qbank-init` compilan con `pdflatex` y corrigen la declaración después (`scripts/fix_moodle_xml.py`, que `make moodle` ejecuta).

El paquete `moodle` se instala con `tlmgr install moodle` (T_EX Live) o `sudo apt install texlive-latex-extra` (Debian/Ubuntu). Dos avisos: el paquete declara el XML en `iso-8859-1` pero escribe algunos caracteres en UTF-8, y escapa mal el «<» dentro de los bloques `cloze`; los scripts de `qbank-init` corrigen ambas cosas (`scripts/fix_moodle_xml.py`). Si exportas a mano, revisa el XML.

6.1. Funciones

Función	Tipo de problema	Qué escribe
QuizMoodle	ProblemaTipo	Vista alumno: una pregunta <i>cloze</i> por variante, con un bloque por parte.
QuizMoodleProfe	ProblemaTipo	Como QuizMoodle con los mismos parámetros, pero con las puntuaciones visibles.
QuizVFMoodle	ProblemaVF	Verdadero/falso (apéndice).

```

QuizMoodle (nombre, directorio, problema, last_choice=False, instances=1, *,
            aux_latex="", last_choice_text="Las demás opciones son falsas")
# Devuelve int: número de variantes escritas
QuizMoodleProfe(nombre, directorio, problema, *, aux_latex="",
                last_choice=False, instances=1,
                last_choice_text="Las demás opciones son falsas")

```

QuizMoodleProfe solo admite `last_choice`, `instances` y `last_choice_text` por palabra clave; con los mismos valores que en QuizMoodle escribe las mismas variantes. Las dos escriben `directorio/nombre.tex`: para tener ambas vistas, dales nombres distintos (p. ej. `MiCuestionario` y `MiCuestionarioProfe`). Como el resto de exportadores, ninguna de las dos lee `export` del problema: pásale `last_choice=True` si la pregunta usa la opción por defecto (o, con `ejercicio.json`, 16 de las 48 variantes saldrían sin ninguna opción correcta).

Parámetro	Descripción
<code>nombre</code>	Nombre del cuestionario (en Moodle, la categoría que agrupa las variantes).
<code>directorio</code>	Directorio de salida (con o sin / final).
<code>problema</code>	Un <code>ProblemaTipo</code> o un diccionario <code>{nombre: ProblemaTipo}</code> (todas van al mismo cuestionario).
<code>last_choice</code>	Si <code>True</code> , añade la opción por defecto a cada bloque. Moodle rechaza preguntas sin ninguna correcta.
<code>instances</code>	Instancias con semillas distintas (requiere <code>setup</code> aleatorio).
<code>aux_latex</code>	Cabecera L ^A T _E X adicional (p. ej. <code>\usepackage{mis-macros}</code>).
<code>last_choice_text</code>	Texto de la opción por defecto.

Cada bloque *cloze* recibe un peso igual a su número de opciones menos uno, y Moodle lo puntúa como una opción múltiple con varias respuestas (las correctas marcadas suman, las incorrectas marcadas restan, sin bajar de cero).

6.2. Ejemplo

```

p = load_problema("ejemplosManual/ejercicio.json")
QuizMoodle("EjemploMoodle", "ejemplosManual/", p, last_choice=True)

cd ejemplosManual
xelatex EjemploMoodle.tex

```

Se generan `ejemplosManual/EjemploMoodle-moodle.xml` (para importar en Moodle) y `ejemplosManual/EjemploMoodle.pdf` (vista previa de las preguntas). Las letras acentuadas, la ñ, la ü y los signos ¿ y ¡ las convierte `codchar` (ver la referencia) de forma automática.

6.3. Preguntas con varias partes

Un `ProblemaTipo` cuya lista plana abre varias partes se exporta como una pregunta *cloze* con un bloque de opciones por parte, sin hacer nada especial. El fichero `ejemplosManual/superheroes.json` es un ejercicio de tres partes (cuatro mundos, uno por héroe, y seis huecos repartidos en tres bloques):

```

p = load_problema("ejemplosManual/superheroes.json")
etiqueta, partes = next(iter(p.por_partes()))
for enunciado, cuestiones in partes:
    print(enunciado.strip() or "(sin texto)")
    for texto, correcto, _, exp in cuestiones:
        print(f"    {'✓' if correcto else '×'} {texto}")
print(len(list(p.por_partes())), "variantes")

```

Conteste qué afirmaciones sobre Spider-Man (Peter Parker) son ciertas. Uno de sus superpoderes

↪ consiste en

- ✓ tener sentidos superdesarrollados
- × volar

En su vestimenta destaca:

- × la capa
- ✓ el color azul

En cuanto al origen de sus superpoderes:

- ✓ son fruto de una mutación
- × realmente son fruto de entrenamiento y tecnología

256 variantes

```

import os
os.makedirs("ejemplosManual/exportaciones", exist_ok=True)
QuizMoodle("Superheroes", "ejemplosManual/exportaciones", p, last_choice=True) # Moodle: cloze
with open("ejemplosManual/exportaciones/superheroes_amc.tex", "w") as f: # AMC: un
    ↪ questionmult
    for etiqueta, partes in p.por_partes():
        f.write(AMC_multipart("Superheroes", etiqueta, "", partes, last_choice=True))

```

6.4. Subir y actualizar el banco en Moodle desde el terminal

Importar a mano (*Banco de preguntas* → *Importar*) basta para una pregunta suelta, pero al corregir una pregunta ya subida hay que borrar antes sus variantes viejas de la categoría, y con cientos de variantes por pregunta eso es tedioso y propenso a errores. El banco **qbank-PEconometria** tiene dos scripts que reproducen desde el terminal los formularios de Moodle con la sesión del navegador, sin servicios web ni permisos de administración: `scripts/moodle_sync.py` (banco de preguntas) y `scripts/moodle_quiz.py` (cuestionarios). No forman parte de la plantilla: están escritos para el Campus Virtual de la UCM (Moodle 4.3 con `auth_saml2`; constantes `BASE` y `COURSEID` al principio de `moodle_sync.py`, o `--base` y `--curso`). Para otro Moodle hay que copiarlos y comprobar que sus páginas son iguales; si no lo son, los scripts fallan sin tocar nada.

La sesión. Entra en el campus con el navegador, abre las herramientas de desarrollo (F12; Firefox: *Almacenamiento* → *Cookies*; Chrome: *Application* → *Cookies*), copia el **valor** de la cookie `MoodleSession` del dominio del campus y guárdalo, solo el valor, en `~/.config/moodle-ucm/session` (permisos `600`) o en la variable `MOODLE_SESSION`. Equivale a tu contraseña mientras dure la sesión: no la pegues en ningún fichero versionado. Caduca con la sesión del navegador (en la UCM, tras una hora de inactividad); si un script dice que no hay sesión, copia otra.

Siempre simulan. Sin `--ejecutar` los dos scripts solo dicen lo que harían; ejecútalos primero así y compara con lo que esperas.

```

PY=~ruta/a/qbank/.venv/bin/python
$PY scripts/moodle_sync.py categorias # categorías y nº de preguntas
$PY scripts/moodle_sync.py preguntas L-14-LeerElVIF # ids de una categoría
$PY scripts/moodle_sync.py reimportar build/moodle/X.xml # simula
$PY scripts/moodle_sync.py reimportar build/moodle/X.xml --ejecutar
$PY scripts/moodle_sync.py importar X.xml --ejecutar # solo importar
$PY scripts/moodle_sync.py borrar L-14-A L-14-B --ejecutar # vaciar categorías

```

```
$PY scripts/moodle_quiz.py cuestionarios          # cmid, nombre, ranuras, apertura
$PY scripts/moodle_quiz.py ranuras 185724         # ranuras de un cuestionario
$PY scripts/moodle_quiz.py aleatorias 185725 L-15-A L-15-B --ejecutar
```

Actualizar preguntas: reimportar. Recibe un XML con las preguntas corregidas (para unas pocas, fusiona solo sus `*-moodle.xml` en un fichero aparte; el de todo el banco también vale). Para cada categoría `$course$/top/<nombre>` del XML que ya exista borra sus preguntas visibles y, después, importa el XML una sola vez: Moodle reutiliza la categoría del mismo nombre (conserva su id) y crea las que falten. **Las categorías no se borran nunca**, y es deliberado: las ranuras aleatorias de un cuestionario filtran por el id de la categoría, y borrar una categoría para volver a importarla deja esas ranuras rotas. `--solo` y `--excluir` limitan las categorías afectadas. Salvaguardas: aborta si el recuento que muestra Moodle no cuadra con el listado o si una pregunta aparece en dos categorías; fotografía todo el banco antes y después de borrar y se detiene antes de importar si cambió algo fuera de lo previsto; al final compara el recuento de cada categoría con el XML y dice **TODO CORRECTO**.

Cuestionarios: aleatorias. Añade al final del cuestionario (por su `cmid`, el número de `mod/quiz/view.php?id=...`) una ranura «pregunta aleatoria» por cada categoría dada, una por página y en el orden dado, y comprueba después que las ranuras nuevas son las pedidas. Usa el mismo servicio AJAX que el botón del navegador.

Lo que se hace a mano, en el navegador: las fechas de apertura y cierre, la visibilidad para los estudiantes y quitar ranuras. `moodle_quiz.py` tiene una orden `apertura` que reenvía el formulario de ajustes, pero **no se usa**: la única vez que se probó, Moodle devolvió un 404 sin guardar nada y la sesión quedó bloqueada.

Con moderación. El cortafuegos de la UCM llegó a dejar colgadas dos sesiones (toda petición con esa cookie quedaba sin respuesta, y durante un rato no se pudo abrir sesión desde la misma IP) tras ráfagas de peticiones rápidas. Por eso los scripts esperan dos segundos entre peticiones. Haz una tarea, comprueba el resultado y para.

7. La vista de lectura (QuizLectura)

La *vista de lectura* es un documento $\text{\LaTeX}$ completo pensado para que el profesor (o un revisor) **lea y opine** sobre las preguntas, distinto del catálogo AMC, que se queda para el examen. Para cada pregunta contiene su tabla de mundos, una nota con cuántas opciones ve el alumno y cuál es la opción por defecto, y una serie de ejemplos: el enunciado en un recuadro y, debajo, las alternativas agrupadas por hueco («Hueco 1, 2, ...»; el alumno recibe una de cada hueco), cada una marcada V (verde) o F (rojo) con su `exp` ya interpolada debajo, en gris («la misma retroalimentación que la anterior» si se repite), y las que la precondition excluye en ese mundo en gris claro con su motivo. Al final de la pregunta se listan los mundos que un **Supuesto** descarta.

```
QuizLectura(nombre, directorio, problemas, *, aux_latex="", titulo=None,
             last_choice=None, last_choice_text="Ninguna de las anteriores",
             extras=0, max_semillas=400)          # → número de ejemplos
```

- `problemas` es un `ProblemaTipo`, una **lista** de ellos (un documento por lección es el uso previsto; el nombre de cada uno es su `nombre`) o un diccionario `{nombre: ProblemaTipo}`. Escribe solo `directorio/nombre.tex` (UTF-8); compilarlo con `pdflatex` es cosa del `Makefile` (`make lectura` en la plantilla).
- `aux_latex` es el preámbulo del proyecto (macros, `babel`), como en AMC y Moodle; el documento carga por su cuenta `inputenc`, `fontenc`, `amsmath`, `amssymb`, `geometry`, `xcolor`, `enumitem` y `graphicx`.

- `last_choice=None` lee `export.last_choice` de cada pregunta (una excepción a la regla de que los exportadores no leen `export`, porque en un mismo documento las preguntas difieren); solo afecta a la nota inicial.

Qué ejemplos salen:

- Sin `setup`: uno por mundo (cada combinación de `Supuesto` y textos alternativos).
- Con `setup` y sin más indicación: uno por mundo con la `seed` de la pregunta, más extras sorteos por mundo cuyo enunciado (sin cifras) sea distinto de los ya vistos.
- Con la clave `lectura` en el JSON (o el argumento `lectura` de `ProblemaTipo`), "`lectura`": {"cubrir": ["ctx", "rech"]}: en cada mundo, un ejemplo por cada combinación de valores de esas variables del `setup`, buscada recorriendo hasta `max_semillas` semillas. Las combinaciones del producto de los valores vistos que no aparezcan se listan al final como *no encontradas* (una `precond` puede hacerlas imposibles). Los valores deben ser discretos; `lint` avisa si son reales o si hay demasiadas combinaciones, y da `ERROR` si una variable no existe en el `setup`. Si la variable que interesa no existe como tal (por ejemplo, «rechaza» se deduce de un `p`-valor), basta definirla en el `setup`: `rech = p < 0.05`.

Los datos salen de `ProblemaTipo.por_partes_lectura(seed=None, verbose=False, descartes=False)`, que es como `por_partes_profe()` pero conserva los huecos y el espacio de nombres del `setup`: produce registros `Lectura(etiqueta, mundo, ns, partes, descarte)`, donde `mundo` es la tupla de índices elegidos en los slots de enunciado, cada parte es `ParteLectura(enunciado, huecos)` y cada alternativa `Alternativa(texto, valor, exp, motivo)` (`valor` es `None`, con `motivo`, si la precondición la excluye). Con `descartes=True` emite también los mundos descartados por un `Supuesto` (`partes=None`, `descarte` con el motivo).

8. Comprobación automática de preguntas (`lint`)

Antes de exportar conviene comprobar que la pregunta está bien construida. `lint` automatiza lo que de otro modo habría que mirar a ojo recorriendo todas las variantes. Acepta un `ProblemaTipo`, un dict o la ruta a un fichero, no modifica nada ni compila L^AT_EX, y devuelve una lista de `Aviso(nivel, codigo, mensaje, donde)` con nivel `ERROR`, `AVISO` o `INFO`, errores primero:

```
for aviso in lint("ejemplosManual/ej_c_precond.json"):
    print(aviso)
```

```
INFO  [constante] falsa en los 2 mundos en que aparece (semantica literal) @ comp 4[2]:
↪  $\det(A)$  puede ser cualquier número real
INFO  [constante] falsa en los 2 mundos en que aparece (semantica literal) @ comp 5[3]:
↪  $A^{-1} = A^{\text{top}}$  necesariamente
INFO  [constante] falsa en los 2 mundos en que aparece (semantica literal) @ comp 6[2]: el
↪ sistema  $Ax=b$  tiene siempre infinitas soluciones
```

Cada aviso lleva su nivel, su código entre corchetes, el mensaje y, tras @, dónde está: `comp 4[2]` es el componente 4 de la lista (contando desde 0), elemento 2. Aquí son tres `INFO`: tres distractores siempre falsos, puestos a propósito.

Un `ERROR` significa que la pregunta está rota; un `AVISO` hay que leerlo y decidir; un `INFO` es información. Las reglas, con el código de cada aviso:

mezcla (ERROR) en alguna variante y parte todas las opciones son correctas o todas incorrectas, contando la opción por defecto si `export.last_choice` está activo (misma regla que los exportadores).

pocas-opciones (AVISO) una parte muestra al alumno menos de tres opciones: el error típico de poner las opciones en una única lista de `Cuestion` (que sortea una). **opciones (AVISO)**: una parte sin ninguna cuestión (típicamente, un texto suelto tras la última `Cuestion`; esa parte no da además *mezcla*), o con un número de opciones que varía entre variantes.

variantes ERROR si la pregunta no genera ninguna variante válida. AVISO con menos de 16 variantes sin `setup`, o menos de 60 con `setup` (contando `export.instances`). INFO con más de 300: la vista alumno de AMC en un solo `examcopy` puede agotar la memoria de `pdflatex`.

ejecucion (ERROR) el `setup` o el recorrido de las variantes lanzan una excepción.

constante (AVISO) una `Cuestion` verdadera (o falsa) en todos los mundos en que puede aparecer, candidata a distractor memorizable. Es INFO si la semántica es el literal `True` o `False` (distractor deliberado) o una `lambda` con `setup` (la tabla de mundos usa un solo sorteo; compruébalo con la vista *profe*). **solape (AVISO)**: dos `Cuestion` con el mismo texto en el mismo hueco pueden aparecer en un mismo mundo.

negacion una *semantica* o *precond* usa `-v('X')` pero ningún *Supuesto* permite deducirlo (hace falta un *unoDe* que incluya `v('X')`, a cualquier profundidad, o un *Supuesto* que afirme o implique `-v('X')`). ERROR si `v('X')` es una alternativa de una lista; AVISO si niega un *Supuesto* suelto (siempre presente): nunca será cierta.

interp (ERROR) queda algún `@{var}` sin sustituir en el texto renderizado (el escape `@@{x}`, que se imprime como `«@{x}»`, no cuenta).

huerfana (AVISO) una etiqueta `v('X')` usada en una `Cuestion` no aparece en ningún *Supuesto* (INFO si algún *Supuesto* es una `lambda`, porque entonces no se puede saber). **sin-uso (INFO)**: una etiqueta declarada que ninguna `Cuestion` usa.

carga, tipo, nombre, estructura (ERROR) el JSON no se puede leer o cargar (*carga*); no es un *ProblemaTipo* (*tipo*); el *nombre* no coincide con el fichero o lleva `:` o espacios, que los scripts del banco no admiten (*nombre*); alguna lista está vacía, mezcla tipos o va anidada, o `export` o `seed` están mal tipados (*estructura*; `"seed": true` tampoco vale). **lista-1, seed, instances (AVISO)**: listas de un solo elemento, *seed* incoherente con el uso de aleatorios, `instances > 1` sin `setup`.

texto un `%` sin barra es ERROR (en $\text{\LaTeX}$ empieza un comentario y el PDF sale vacío o roto sin que `pdflatex` pare). Son AVISO: `\%` dentro de `$.$.` (Moodle lo pierde), `°`, `«»` y comillas rectas (no existen en la codificación del paquete `moodle` o salen como `”`), `\(..\)` o `\[..\]` en vez de `$.$. / $$.$.`, citas a números de lección, `@{` sin cerrar, dobles espacios fuera de las fórmulas, comillas simples, textos que parecen código `lambda ns`: (desde JSON no se evalúan). Las dos reglas del `%` se aplican también al texto interpolado con unos sorteos del `setup` (el mensaje lo indica con «en el texto interpolado»).

setup (INFO) variantes por instancia e `instances` necesarias para llegar a 60.

patron (AVISO; solo con `patrones=True` o `--patrones`) *patrones* memorizables: lo que un alumno podría acertar *sin leer el enunciado* tras ver varias variantes. Muestra las variantes (40 sorteos del `setup`, o los que se pidan) y avisa de una `Cuestion` de verdad variable pero «casi constante» (el mismo valor en $\geq 85\%$ de sus apariciones; las constantes del todo las da *constante*), de una `lambda` con `setup` constante en toda la muestra (donde *constante* solo pudo dar INFO), de una cifra que dentro de un «tipo de frase» (el texto sin cifras) separa por completo las verdaderas de las falsas (una elasticidad verdadera siempre menor que 1 y su gemela inversa siempre mayor), y de un rasgo superficial de la opción —una palabra, una cifra concreta, repetir una cifra del enunciado, tener la cifra

mayor o menor del hueco, ser más larga que la mediana— que, con él o sin él, predice la verdad en el hueco ($\geq 85\%$ de acierto, ≥ 15 puntos sobre la tasa base, soporte a ambos lados y p-valor corregido). Los rasgos salen del propio banco (no hay lista de palabras ni dependencia del idioma), y no se miran relaciones aritméticas con los datos del enunciado: en una pregunta de cálculo la respuesta correcta *debe* ser función de ellos. Es más lenta que el resto del lint, por eso no corre por defecto; sus avisos son indicios que hay que leer: a veces el remedio es una gemela más (que la cifra grande sea a veces la verdadera), a veces no hay nada que corregir.

`tabla_mundos(problema)` devuelve una tabla de texto con el valor de cada Cuestion (V, F, o - si la rechaza su `precond`) en cada combinación de `Supuesto`, para localizar distractores constantes y auditar la lógica. Con `setup` usa un solo sorteo (sembrado con `seed`), así que los mundos filtrados por condiciones numéricas pueden no aparecer todos.

Desde la línea de órdenes (código de salida 1 si hay algún ERROR; `--quiet` omite los INFO y los ficheros sin avisos):

```
python -m qbank.lint preguntas/*.json --quiet
qbank-lint preguntas/L-01-Nombre.json --tabla      # equivalente, con la tabla de mundos
qbank-lint preguntas/*.json --quiet --patrones     # además, patrones memorizables (--patrones
↳ 100: más sorteos)
```

En un proyecto de `qbank-init`: `make lint`, `make tabla PREGUNTA=L-01-Nombre`, `make patrones (SEMILLAS=N para cambiar los sorteos)`, y `make amc` / `make moodle` pasan lint antes de exportar.

9. Referencia rápida de la API

9.1. Clases

```
Supuesto(enunciado, semantica, precond=True)
Cuestion(enunciado, semantica, precond=True, exp="")
    semantica y precond: fórmula calcpop, True/False, lambda ns: ..., o la cadena de cualquiera
    ↳ de ellos
ProblemaTipo(componentes, setup=None, export=None, seed=None, nombre=None)
    ↳ .por_partes(instances=1, base_seed=None, verbose=False)    vista alumno, por partes
    ↳ .por_partes_profe(verbose=True)                            vista profe: todas las
    ↳ cuestiones
    ↳ itera (etiqueta, enunciado, cuestiones)                    solo ejercicios de una parte
ProblemaVF(enunciado, cuestiones, NumPreguntas)                verdadero/falso (apéndice)
ProblemaTipoEditor(source=None)                                editor visual en Jupyter
↳ (apéndice)
qbank.__version__                                              versión instalada, p. ej.
↳ "0.6.0"
```

9.2. Funciones JSON

```
problema_from_dict(d)      ↳ ProblemaTipo o ProblemaVF
problema_to_dict(problema) ↳ dict (conserva las claves propias del JSON)
load_problema(ruta_o_dict) ↳ ProblemaTipo o ProblemaVF
save_problema(problema, ruta)
load_banco(ruta)           ↳ lista de problemas
save_banco(problemas, ruta) # acepta lista o dict
problema_to_python(problema, varname="ejercicio") ↳ str    (ProblemaTipo)
save_problema_py(problema, ruta, varname="ejercicio")
```

9.3. Funciones AMC

```
AMCblock      (nombre, etiqueta, enunciado, cuestiones,
               last_choice=False, cols=1, profe=False, *,
               aux_latex="", last_choice_text="Ninguna de las anteriores") → str
AMC_multipart(nombre, etiqueta, enunciado, partes,
               last_choice=False, cols=1, profe=False, *,
               aux_latex="", last_choice_text="Ninguna de las anteriores") → str
QuizAMCProfe (nombre, directorio, problema, cols=1, *, last_choice=False,
               aux_latex="", last_choice_text="Ninguna de las anteriores") → int (mundos)
AMC_VF        (nombre, etiqueta, enunciado, cuestiones, opc=None) → str
↪ (ProblemaVF)
```

9.4. Funciones Moodle

```
QuizMoodle    (nombre, directorio, problema, last_choice=False, instances=1, *,
               aux_latex="", last_choice_text="Las demás opciones son falsas") → int
               ↪ (variantes)
QuizMoodleProfe (nombre, directorio, problema, *, aux_latex="",
                 last_choice=False, instances=1,
                 last_choice_text="Las demás opciones son falsas") → int
QuizVFMoodle   (nombre, directorio, GenVar, num, opc=None) (ProblemaVF)
codchar(s)     vocales acentuadas minúsculas y ñ → secuencias LaTeX (lo aplican las funciones
               ↪ Moodle)
```

9.5. Vista de lectura

```
QuizLectura(nombre, directorio, problemas, *, aux_latex="", titulo=None,
              last_choice=None, last_choice_text="Ninguna de las anteriores",
              extras=0, max_semillas=400) → int (ejemplos)
ProblemaTipo.por_partes_lectura(seed=None, verbose=False, descartes=False)
# → Lectura(etiqueta, mundo, ns, partes, descarte); ParteLectura(enunciado, huecos);
# Alternativa(texto, valor, exp, motivo)
```

9.6. Comprobación automática

```
lint(problema_o_ruta, instances=None, seed=None, patrones=None) → list[Aviso]
# Aviso(nivel, codigo, mensaje, donde); nivel ∈ {ERROR, AVISO, INFO}; errores primero
# patrones=True (o nº de sorteos): añade la regla ll, patrones memorizables (más lenta)
tabla_mundos(problema_o_ruta) → str # tabla Cuestion × mundo
python -m qbank.lint FICHEROS.json [--instances N] [--seed S] [--tabla] [--quiet] [--patrones
↪ [N]]
# también `qbank-lint`; código de salida 1 si hay algún ERROR
```

9.7. Proyecto de banco

```
qbank-init DIR [--forzar]      copia la plantilla (preguntas/, scripts/, Makefile, config.py,
                               nueva_pregunta.py, README.org, prompts/, CLAUDE.md) en DIR;
                               ↪ equivale a
                               python -m qbank._plantilla DIR
```

9.8. Operadores de **calccprop** (disponibles tras **from qbank import ***)

```
v("X")          variable proposicional X
-v("X")          negación de X
v("X") & v("Y")  conjunción ( $X \wedge Y$ )
v("X") | v("Y")  disyunción ( $X \vee Y$ )
v("X") >> v("Y") implicación ( $X \rightarrow Y$ )
v("X") ** v("Y") bicondicional ( $X \leftrightarrow Y$ )
```

<code>alguno(P, Q, R)</code>	al menos una de la lista es verdadera
<code>unoDe(P, Q, R)</code>	exactamente una de la lista es verdadera
<code>True</code>	tautología (siempre verdadero)
<code>False</code>	contradicción (siempre falso)

`test(formula, lista_de_formulas) → bool`
 Devuelve True si formula es consecuencia lógica de la lista de hipótesis.

10. Preguntas frecuentes

10.1. ¿Por qué algunas variantes no se generan?

Si un `Supuesto` o una `Cuestion` tiene una precondition que no se satisface en una combinación determinada, esa variante completa se descarta. Es el comportamiento esperado: evita combinaciones de hipótesis incoherentes. En la vista profe, las cuestiones rechazadas se ven; un `Supuesto` rechazado hace desaparecer la variante también allí.

10.2. ¿Cómo incluyo texto L^AT_EX en los enunciados?

Los campos `enunciado` y `exp` son cadenas Python que se insertan tal cual en el `.tex`. Usa cadenas crudas (`r"..."`) para no doblar las barras invertidas:

```
Cuestion(r"$\mathbf{X}^{\top}\mathbf{X}$ es invertible", -v("Mcoli"))
```

En JSON cada barra se dobla: `"$\mathbf{X}^{\top}\mathbf{X}$"`.

10.3. ¿Puedo escribir un porcentaje, grados, comillas...?

El símbolo `%` es, en L^AT_EX, el inicio de un comentario: escribe siempre `\%` (y fuera de los dólares: al `$5$, \%`). Nada de `°` (usa `^{\circ}`) ni de comillas angulares «» o rectas "..." (usa ``así'``): el paquete `moodle` no las admite o las deforma. Acentos, ñ, ¿ y ¡ sí se pueden teclear. `lint` avisa de todo esto; la Guía lo explica.

10.4. ¿Cómo escribo display math?

Con `$$...$$`, nunca `\[...\]` ni `\(...\)` directamente: el paquete `moodle` necesita los dólares, y los exportadores AMC convierten `$$...$$` a `\[...\]` automáticamente (`automultiplechoice` a veces produce errores «Missing \$» con `$$`). Separa dos expresiones en línea contiguas con un espacio (`$A$ $B$`, no `$A$B$`).

10.5. ¿Puedo incluir una figura en el enunciado?

Sí, con `\includegraphics` en el texto (el enunciado es L^AT_EX), pero cada destino la quiere de una forma. En AMC vale un PDF a tamaño natural; los documentos de la plantilla ya cargan `graphicx`:

```
"Observe la figura. \begin{center}\includegraphics{fig.pdf}\end{center} Entonces:"
```

El paquete `moodle` incrusta la imagen en el XML (en base64) y solo sabe de mapas de bits: hace falta un PNG del mismo nombre y compilar con `pdflatex -shell-escape`, porque llama a `base64`. Un patrón que funciona: generar `fig.pdf` y `fig.png` (p. ej. con `pdftoppm -png -r 200`), referenciar el PDF en el JSON y, en el script de Moodle, sustituir `.pdf` por `.png` en el `.tex` (con `width` igual a la anchura natural del PDF, para que Moodle no lo agrande); el banco de Probabilidad y Estadística del autor lo hace así en su `scripts/gen_moodle.py`. Si la ruta de la figura va en el `setup` como variable e interpolada con `@{fig}`, la misma pregunta sirve para los

dos destinos. Ten en cuenta el tamaño: una figura en cada una de 150 variantes multiplica el XML.

10.6. ¿Para qué sirve `codchar`?

`codchar(s)` convierte las letras del español con tilde o diéresis, mayúsculas y minúsculas, y la ñ a órdenes de acento de $\text{\LaTeX}$ ($\acute{a} \rightarrow \text{\texttt{'a}}$, $\ddot{u} \rightarrow \text{\texttt{"u}}$), y los signos ¿ y ¡ a `\texttt{?questiondown}` y `\texttt{!exclamdown}`. Las funciones Moodle lo aplican automáticamente; no hace falta llamarlo a mano. El paquete `moodle` convierte esas órdenes en entidades HTML (`´`, `ü`, `¿`), que se leen bien se declare el XML como se declare. Un carácter que `codchar` no cubra (las comillas “ ”, por ejemplo) pasa tal cual: con `xelatex` sale bien; con `pdflatex` llega al XML en UTF-8 con la declaración `iso-8859-1` y lo corrige `scripts/fix_moodle_xml.py` de la plantilla (ver la sección 6). Hasta la 0.7.0 `codchar` solo cubría las minúsculas acentuadas y la ñ, y un banco con una copia antigua de ese script mostraba en Moodle `antigÃ¼edad` o `Â¿Qué`. Para AMC no suele hacer falta si el documento principal usa `inputenc` con UTF-8.

10.7. ¿Cómo exporto varias preguntas a un mismo cuestionario Moodle?

Pasa un diccionario a `QuizMoodle`: todas van al cuestionario `MiExamen`, y cada clave da nombre a sus variantes (`Seccion-A-1`, `Seccion-A-2`, ...):

```
QuizMoodle("MiExamen", "salida/", {"Seccion-A": p_a, "Seccion-B": p_b}, last_choice=True)
```

Un banco de `qbank-init` guarda cada pregunta en su propio fichero, y el script `scripts/merge_moodle_xml.py` fusiona sus XML en `_banco.xml`.

11. Apéndice: API avanzada y compatibilidad

11.1. `ProblemaVF` — preguntas de verdadero/falso

`ProblemaVF` sirve para un formato distinto: un banco de afirmaciones del que se extrae aleatoriamente un subconjunto en cada variante. No usa cálculo proposicional; la corrección de cada ítem se indica con `True` o `False`. No lo revisa `lint` ni lo exportan los scripts de `qbank-init`: para un banco nuevo, una pregunta de tres huecos de `ProblemaTipo` cubre el mismo uso con todas las herramientas.

```
ProblemaVF(enunciado, cuestiones, NumPreguntas)
```

Parámetro	Descripción
<code>enunciado</code>	Texto del enunciado común a todas las variantes.
<code>cuestiones</code>	Lista de tuplas (<code>texto</code> , <code>correcto</code>).
<code>NumPreguntas</code>	Cuántas afirmaciones se eligen aleatoriamente por variante.

```
enunciado = "Indique qué afirmaciones son verdaderas:"
banco = [
    ("Todo alumno que va a clase aprueba", False),
    ("Todo alumno que sabe aprueba", True),
    ("Si suspenden todos es frustrante", True),
    ("Si aprueban todos eres buen profesor", False),
]
import random
random.seed(42)
GenVar = iter(ProblemaVF(enunciado, banco, 3)) # 3 ítems por variante

for _ in range(2):
```

```

etiqueta, enunc, cuestiones = next(GenVar)
print(f"Variante {etiqueta}: {enunc}")
for texto, correcto in cuestiones:
    print(f" {'✓' if correcto else '×'} {texto}")

```

Variante 1: Indique qué afirmaciones son verdaderas:

- × Todo alumno que va a clase aprueba
- × Si aprueban todos eres buen profesor
- ✓ Todo alumno que sabe aprueba

Variante 2: Indique qué afirmaciones son verdaderas:

- ✓ Todo alumno que sabe aprueba
- × Todo alumno que va a clase aprueba
- ✓ Si suspenden todos es frustrante

Se exporta con `AMC_VF(nombre, etiqueta, enunciado, cuestiones)` (un bloque por variante) y `QuizVFMoodle(nombre, directorio, GenVar, num)` (num variantes). Se guarda y carga en JSON sin restricciones.

11.2. Clases deprecadas

ProblemaTipoProfe emite `DeprecationWarning`; usa `ProblemaTipo(...).por_partes_profe()`. Firma:

```
ProblemaTipoProfe(lista, setup=None)
```

ProblemaMultiParte y **SubPregunta** `ProblemaMultiParte` emite `DeprecationWarning` y devuelve un `ProblemaTipo` equivalente (lista plana); `SubPregunta` es su contenedor. Para código nuevo, define directamente un `ProblemaTipo` y recórrelo con `por_partes()`. Firmas:

```
ProblemaMultiParte(componentes, subpreguntas, setup=None)
SubPregunta(intro, cuestiones)
```

11.3. Editor visual en Jupyter (**ProblemaTipoEditor**)

`ProblemaTipoEditor` es un formulario interactivo para diseñar un `ProblemaTipo` desde un cuaderno de Jupyter, sin escribir la lista de listas a mano. Requiere `ipywidgets`:

```
pip install "calccprop-qbank[jupyter]"
```

(la orden con `git+https://...` de la instalación rápida ya incluye el extra `[jupyter]`).

En **JupyterLab** ≥ 4 es necesario que la extensión `@jupyter-widgets/jupyterlab-manager` esté habilitada: compruébalo con `jupyter labextension list`. Si el widget se muestra como texto (`VBox(children=...)`), ejecuta una sola vez desde el terminal:

```

mkdir -p ~/.local/share/jupyter/labextensions/@jupyter-widgets
ln -sf $(python -c "import jupyterlab_widgets, os; \
    print(os.path.dirname(jupyterlab_widgets.__file__))")/labextension \
    ~/.local/share/jupyter/labextensions/@jupyter-widgets/jupyterlab-manager

```

y reinicia JupyterLab (en Jupyter Notebook clásico no hace falta).

```

from qbank import ProblemaTipoEditor
editor = ProblemaTipoEditor()                # editor vacío
editor = ProblemaTipoEditor('mi_problema.json') # cargar desde fichero
p = editor.to_problema()                     # ProblemaTipo listo para iterar
d = editor.to_dict()                         # dict JSON equivalente
save_problema(editor.to_problema(), "mi_problema.json") # o el botón Guardar

```

El editor no tiene un método `save`: para guardar, usa el botón **Guardar** o pasa `editor.to_problema()` a `save_problema`, como en la última línea del bloque.

El formulario tiene una cabecera (**Nombre:**, **Setup:** con el campo **semilla:**, y **Export:** con la casilla `last_choice` y el campo `cols:`), la lista de huecos y una barra de botones. Cada hueco (Hueco 1, Hueco 2, etc.) tiene un tipo, *texto*, *supuestos* o *cuestiones*, y sus alternativas (+ `alt`): filas con enunciado, semántica, precondition y, en las cuestiones, explicación. Los botones de cada hueco lo suben o bajan (▲ / ▼), insertan otro hueco debajo (+) o lo borran (×); el botón + hueco añade uno al final. La barra inferior tiene ► **Preview** (genera hasta N variantes, con la casilla *vista profe*), { } **JSON** y </> **.py** (muestran el JSON o el guion Python), **Guardar** y **Cargar** (en el fichero indicado) y ⬇ **.json** / ⬇ **.py** (descargas al navegador). Al guardar conserva las claves de `export` que el formulario no edita (como `instances`) y las claves propias del JSON. `Demo.ipynb` (sección 8) lo muestra en uso.