

Índice general

I	Introducción	1
1.	Qué puede hacer qbank	3
2.	Instalación rápida (para quien ya conoce Python)	4
II	Instalación paso a paso	5
3.	Requisitos previos	6
4.	Qué es un entorno virtual y por qué usarlo	7
5.	Crear y activar un entorno virtual	8
6.	Instalar calcprop y calcprop-qbank	9
7.	Nota para usuarios de Windows	10
III	Conceptos fundamentales	11
8.	Variables proposicionales con calcprop	12
9.	Supuesto — una hipótesis del enunciado	13
10.	Cuestion — un ítem de respuesta	14
11.	ProblemaTipo — un generador de variantes	15
12.	ProblemaVF — preguntas de verdadero/falso	17
13.	Preguntas paramétricas con setup	18
13.1.	Sin setup: combinatoria pura	18
13.2.	Con setup y valores aleatorios	18
13.3.	Con setup y valores fijos	19
13.4.	Con setup y precondiciones dependientes de los valores	19
14.	Guardar y cargar problemas en formato JSON	21
14.1.	De lista de listas a JSON	21
14.2.	De JSON a lista de listas	22
14.3.	Con setup paramétrico	23
14.4.	Campo seed: reproducibilidad independiente del orden	24
14.5.	Limitaciones	24
15.	Exportar la lista de listas a un fichero .py editable	25

IV Ejemplos progresivos	26
16.Ejemplo A — T/F literales, tres sublistas (ej_a_literales.json)	28
17.Ejemplo B — Supuesto en lista, dos mundos proposicionales (ej_b_mundos.json)	29
18.Ejemplo C — Axioma vacío y precondition por mundo (ej_c_precond.json)	30
19.Ejemplo D — setup paramétrico con interpolación $\theta\{var\}$ (ej_d_setup_vars.json)	31
20.Ejemplo E — Semántica lambda e inyección numérica (ej_e_lambda.json)	32
V Exportar a AMC (\LaTeX)	33
21.Funciones disponibles	35
22.Función unificada <code>AMCblock</code>	36
23.Firma	37
24.Ejemplo AMC básico	38
25.Ejemplo AMC multicolumna con «Ninguna de las anteriores»	39
26.Vista aplanada del profesor: <code>QuizAMCProfe</code>	40
27.Integración en el documento AMC	41
VI Exportar a Moodle (vía \LaTeX)	42
28.Flujo de trabajo	44
29.Funciones disponibles	45
30.Firma	46
31.Ejemplo Moodle completo	47
32.Ejemplo con <code>ProblemaVF</code> para Moodle	48
VII Preguntas multi-parte	49
33.Ejemplo	51
34.Exportar a AMC	52
35.Exportar a Moodle (cloze)	53
36.Persistencia JSON	54
37.Exportar a código Python editable	55
38.Compatibilidad: <code>ProblemaMultiParte</code> y <code>SubPregunta</code> (deprecados)	56
VIII Editor visual en Jupyter (<code>ProblemaTipoEditor</code>)	57
39.Instalación	59
40.Uso básico	60

41.Descripción del formulario	61
42.Ejemplo sin setup	62
43.Ejemplo con setup paramétrico	63
44.Obtener el problema desde el editor	64
45.Flujo completo: editor → JSON → AMC y Moodle	65
45.1. Paso 1 — Diseñar y guardar desde el editor	65
45.2. Paso 2 — Cargar el JSON	65
45.3. Paso 3a — Exportar a AMC	65
45.4. Paso 3b — Exportar a Moodle	66
45.5. Paso 4 — Generar múltiples instancias numéricas (problemas con setup)	66
IX Referencia rápida de la API	68
46.Clases	69
47.Funciones JSON	70
48.Funciones AMC	71
49.Funciones Moodle	72
50.Operadores de calcpop (disponibles tras <code>from qbank import *</code>)	73
X Preguntas frecuentes	74
51.¿Por qué algunas variantes no se generan?	75
52.¿Cómo incluyo texto \LaTeX en los enunciados?	76
53.¿Cómo escribo <code>display math</code> en enunciados para AMC?	77
54.¿Para qué sirve <code>codchar</code> ?	78
55.¿Cómo genero un diccionario de preguntas de varias secciones?	79

Manual de uso de **qbank**

Marcos Bujosa

16 de junio de 2026

Parte I

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 definidos por el usuario, `qbank` genera todas las variantes posibles de una pregunta combinando los elementos alternativos. Para cada variante calcula cuáles respuestas son correctas y cuáles son falsas, y exporta el resultado a los formatos $\text{\LaTeX}/\text{AMC}$ o Moodle.

La librería se apoya en [calcprop](#), que proporciona el motor de cálculo proposicional.

Capítulo 1

Qué puede hacer qbank

- Definir preguntas con supuestos e ítems intercambiables.
- Calcular automáticamente la corrección de cada ítem según las hipótesis activas en cada variante.
- Exportar las variantes generadas a:
 - Ficheros `.tex` compatibles con [AMC](#) (Auto Multiple Choice).
 - Ficheros `.tex` compilables con el paquete `LATEX moodle`, que produce un `.xml` importable en Moodle.

Capítulo 2

Instalación rápida (para quien ya conoce Python)

```
pip install calcprop calcprop-qbank
```

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

Parte II

Instalación paso a paso

Capítulo 3

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*).

Capítulo 4

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`.

Capítulo 5

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.

Capítulo 6

Instalar calcprop y calcprop-qbank

Con el entorno activo:

```
pip install calcprop calcprop-qbank
```

Verifica la instalación:

```
python -c "from qbank import *; print('qbank OK')"
```

Si ves qbank OK todo está en orden.

Capítulo 7

Nota para usuarios de Windows

En Windows, dentro del entorno virtual, usa `python` (no `python3`). Si el primer `py -m venv .venv` falla con un error de permisos en PowerShell, ejecuta antes:

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

Parte III

Conceptos fundamentales

Capítulo 8

Variables proposicionales con `calccprop`

`qbank` importa automáticamente `calccprop`, así que al escribir `from qbank import *` tienes acceso al motor lógico. Las variables proposicionales se crean con `v()`:

```
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: `alguno(P, Q, R)` es verdadera si alguna de la lista lo es, y `unoDe(P, Q, R)` es verdadera si lo es exactamente una.

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 internamente al generar las variantes.

Capítulo 9

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 \LaTeX del supuesto, tal como aparecerá impreso.
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).

Ejemplo:

```
Supuesto('$\mathcal{A}$ es verdadero. ', v('A'))  
Supuesto('$\mathcal{A} \implies \mathcal{C}$. ', v('A') >> v('C'))
```

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

Capítulo 10

Question — un ítem de respuesta

Una `Question` es una posible respuesta en el cuestionario. `qbank` determina automáticamente si es verdadera o falsa según las hipótesis activas en cada variante.

```
Question(enunciado, semantica, precond=True, exp="")
```

Parámetro	Tipo	Descripción
<code>enunciado</code>	cadena	Texto \LaTeX del ítem de respuesta.
<code>semantica</code>	fórmula/bool	Fórmula cuya veracidad determina si el ítem es correcto.
<code>precond</code>	fórmula/bool	La cuestión solo aparece en la pregunta si esta precondición se satisface. Por defecto <code>True</code> .
<code>exp</code>	cadena	Explicación/retroalimentación (feedback) en \LaTeX . Opcional.

Ejemplos:

```
# Ítem siempre incluido; correcto cuando C es verdadero
Question("Entonces  $\mathcal{C}$  es verdadero", v("C"))

# Ítem siempre falso (False como semántica)
Question("Esta afirmación es siempre falsa", False)

# Ítem que solo aparece cuando A es verdadero; correcto cuando también B lo es
Question(" $\mathcal{B}$  es consecuencia de  $\mathcal{A}$ ", v("B"), precond=v("A"))

# Ítem con retroalimentación para el alumno
Question("Las columnas son linealmente independientes", -v("Mcoli"),
  exp=r"Solo cuando  $\text{rg}(\mathbf{X}) = k$ ")
```

Capítulo 11

ProblemaTipo — un generador de variantes

ProblemaTipo toma una lista de elementos y genera todas las combinaciones posibles, produciendo una variante por cada combinación. Los elementos de la lista pueden ser:

- Cadenas de texto (texto fijo del enunciado).
- Un único Supuesto o Cuestion (aparece igual en todas las variantes).
- Una lista de Supuesto o Cuestion (en cada variante se elige uno de ellos).

```
ProblemaTipo(lista_de_elementos)
```

ProblemaTipo es un iterador: cada llamada a `next()` o cada iteración en un bucle `for` devuelve una tupla (etiqueta, enunciado, cuestiones):

Campo	Contenido
etiqueta	Número de variante como cadena ("1", "2", ...).
enunciado	Texto completo del enunciado de esa variante.
cuestiones	Lista de tuplas (texto, correcto, activa, explicacion) por ítem.

Ejemplo mínimo:

```
from qbank import *

p = ProblemaTipo([
    "Dado que ",
    [
        Supuesto("$\\mathcal{A}$ es verdadero, ", v("A")),
        Supuesto("$\\mathcal{B}$ es verdadero, ", v("B")),
    ],
    "indique qué opción es correcta: ",
    [
        Cuestion("$\\mathcal{A}$ es verdadero.", v("A")),
        Cuestion("$\\mathcal{B}$ es verdadero.", v("B")),
    ],
])

for etiqueta, enunciado, cuestiones in p:
    print(f"Variante {etiqueta}: \n {enunciado}")
    for texto, correcto, activa, exp in cuestiones:
        marca = "✓" if correcto else "×"
        print(f" {texto} {marca} ")
```

Variante 1:

Dado que \mathcal{A} es verdadero, indique qué opción es correcta:
 \mathcal{A} es verdadero. ✓

Variante 2:

Dado que \mathcal{A} es verdadero, indique qué opción es correcta:
 \mathcal{B} es verdadero. ×

Variante 3:

Dado que \mathcal{B} es verdadero, indique qué opción es correcta:

\mathcal{A} es verdadero. \times

Variante 4:

Dado que \mathcal{B} es verdadero, indique qué opción es correcta:

\mathcal{B} es verdadero. \checkmark

Este ejemplo produce 4 variantes (2 supuestos \times 2 cuestiones).

Capítulo 12

ProblemaVF — preguntas de verdadero/falso

ProblemaVF sirve para un formato distinto: un banco de afirmaciones del que se extraen aleatoriamente un subconjunto en cada variante. No usa cálculo proposicional; la corrección de cada ítem se indica directamente con True o False.

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

Parámetro	Descripción
enunciado	Texto del enunciado común a todas las variantes.
cuestiones	Lista de tuplas (texto, correcto).
NumPreguntas	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) # para generar siempre el mismo ejemplo
GenVar = iter(ProblemaVF(enunciado, banco, 3)) # 3 ítems por variante

for _ in range(3):
    etiqueta, enunc, cuestiones = next(GenVar)
    print(f"\nVariante {etiqueta}:\n {enunc} ")
    for texto, correcto in cuestiones:
        marca = "✓" if correcto else "×"
        print(f" {texto} {marca}")
```

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 ✓
```

Variante 3:

```
Indique qué afirmaciones son verdaderas:
Todo alumno que va a clase aprueba ×
Si suspenden todos es frustrante ✓
Si aprueban todos eres buen profesor ×
```

Capítulo 13

Preguntas paramétricas con setup

El parámetro opcional `setup` de `ProblemaTipo` permite definir problemas cuyos valores numéricos o simbólicos cambian en cada variante. Cuando se proporciona, en cada iteración:

1. Se llama a `setup()` y el dict que devuelve se convierte en el **espacio de nombres** de la variante.
2. Los textos de los slots que contengan marcadores `@variable` reciben la sustitución correspondiente.
3. Las semánticas y precondiciones que sean callable (`lambda ns: ...`) se evalúan con ese diccionario.

El delimitador `@` se eligió para evitar conflictos con el modo matemático de \LaTeX . Para obtener un arroba literal en el texto, escribe `@@`.

13.1. Sin setup: combinatoria pura

El uso habitual de `ProblemaTipo` (sin `setup`) ya se ha visto en la sección anterior. No hace falta ningún código de `setup`; los textos son fijos y la variación viene solo de la elección entre alternativas.

13.2. Con setup y valores aleatorios

El siguiente ejemplo genera preguntas en las que los valores numéricos cambian en cada iteración porque el `setup` usa `random`:

```
import random

def numeros_aleatorios():
    a = random.randint(1, 9)
    b = random.randint(1, 9)
    return {'a': a, 'b': b, 'suma': a + b}

ejercicio_param = ProblemaTipo(
    [
        "Sean $a = @a$ y $b = @b$. ",
        [
            Cuestion("$a + b = @suma$", True),
            Cuestion("$a + b > 10$", lambda ns: ns['a'] + ns['b'] > 10),
        ],
    ],
    setup=numeros_aleatorios,
)
```

En este ejemplo:

- La cadena "Sean \$a = @a\$ y \$b = @b\$. " interpola `@a` y `@b` con los valores de cada variante.
- La semántica `True` de la primera `Cuestion` indica que la afirmación « $a + b = @suma$ » siempre es correcta (el propio texto incluye ya la respuesta correcta).
- La semántica `lambda ns: ns['a'] + ns['b'] > 10` se evalúa con los valores de la variante.

Importante: cuando `setup` devuelve valores distintos en cada llamada (como aquí, con `random`), el iterador no tiene fin. Controla externamente cuántas variantes generas con un `break`, `itertools.islice` o un `range` acotado.

13.3. Con `setup` y valores fijos

Si el `setup` devuelve siempre el mismo diccionario, la combinatoria sigue siendo finita pero los textos usan los parámetros fijos para componer el enunciado. Esto es útil para parametrizar el mismo tipo de pregunta con constantes que se definen una sola vez:

```
def parametros_fijos():
    return {'n': 3, 'k': 2}

ejercicio_fijo = ProblemaTipo(
    [
        "Considere una matriz de  $n \times k$ . ",
        [
            Supuesto("con rango máximo. ", v("MaxRk")),
            Supuesto("con rango deficiente. ", -v("MaxRk")),
        ],
        [
            Cuestion("Tiene  $k$  columnas linealmente independientes.", v("MaxRk")),
        ],
    ],
    setup=parametros_fijos,
)
```

13.4. Con `setup` y precondiciones dependientes de los valores

Cuando un `Supuesto` afirma algo sobre los valores aleatorios del `setup` (por ejemplo, un orden entre dos números), conviene evitar que el enunciado muestre una afirmación falsa. 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 $a > b$ », que es falso para esos números.

La solución es una **precondición** lambda `ns: ...` que solo deja pasar la variante cuando la afirmación es realmente cierta para los valores sorteados:

```
import random

def dos_numeros():
    a = random.randint(2, 9)
    b = random.randint(2, 9)
    return {'a': a, 'b': b}

ordenar = ProblemaTipo(
    [
        "Sean  $a = @a$  y  $b = @b$  dos números naturales. Se sabe que ",
        [
            Supuesto(" $a > b$ . ", True, precond=lambda ns: ns['a'] > ns['b']),
            Supuesto(" $a < b$ . ", True, precond=lambda ns: ns['a'] < ns['b']),
            Supuesto(" $a = b$ . ", True, precond=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),
        ],
    ],
    setup=dos_numeros,
)
```

Para cada combinación (supuesto \times cuestión) el `setup` genera una tirada nueva; si su precondición no se cumple, esa variante se descarta (verás un aviso «=Supuesto ... rechazado por ...=»). De este modo:

- el supuesto mostrado siempre coincide con el orden real de `@a` y `@b`;
- la corrección de cada cuestión se calcula con los valores reales (`lambda ns: ...`), no con una variable proposicional abstracta.

El problema `numeros.json` que carga el editor visual en el notebook de demostración `ejemplosManual/Demo.ipynb` (sección «8.2 Editor cargando un ejemplo con setup») es una versión ampliada de esta misma idea: añade el supuesto « $a \neq b$ » —para que rara vez se descarten todos los supuestos a la vez— y reparte las cuestiones en dos grupos de opciones que se combinan (una de cada grupo por variante), además de calcular `@suma` y `@producto` en el `setup`. Puedes lanzarlo en el navegador, sin instalar nada, con Binder: [abrir Demo.ipynb en mybinder](#). Como el `setup` es aleatorio, el caso « $a = b$ » aparece rara vez; si solo quieres ver unas pocas variantes, acota la generación con `itertools.islice` igual que en el resto de ejemplos con `setup` aleatorio.

Capítulo 14

Guardar y cargar problemas en formato JSON

qbank puede convertir cualquier ProblemaTipo a JSON y reconstruirlo después. Esto permite guardar el banco de preguntas en un fichero, compartirlo o cargarlo desde el editor visual sin tener que reescribir el código Python.

Restricción de nombres de fichero: el nombre del fichero `.json` no debe contener `:`. GNU Make interpreta `:` como separador de regla, por lo que un fichero llamado `L-10-Gujarati:Table6-3.json` rompe cualquier Makefile que use `*.json` como dependencia. Los exportadores convierten automáticamente `:` y `_` en `-` dentro del L^AT_EX generado, pero el nombre del fichero en disco es responsabilidad del usuario. El editor visual muestra un aviso si el nombre introducido contiene `:`.

14.1. De lista de listas a JSON

Dado un ProblemaTipo creado directamente en Python, basta llamar a `problema_to_dict` para obtener el dict JSON, o a `save_problema` para guardarlo en un fichero:

```
from qbank import *

ejercicio = [
    "Considere ",
    [
        Supuesto("$\\mathcal{A}$", v("A")),
        Supuesto("$\\mathcal{B}$", v("B")),
    ],
    [
        Supuesto("y que $\\mathcal{A}\\rightarrow\\mathcal{C}$", v("A") >> v("C")),
        Supuesto("y que $\\mathcal{B}\\Leftrightarrow\\mathcal{D}$", v("B") ** v("D")),
    ],
    "Conteste: ",
    [
        Cuestion("¿Se da $\\mathcal{C}$?", v("C"),
            exp="Solo si $\\mathcal{A}$ y $\\mathcal{A}\\rightarrow\\mathcal{C}$"),
        Cuestion("¿Se da $\\mathcal{D}$?", v("D")),
    ],
]

p = ProblemaTipo(ejercicio)

import json
d = problema_to_dict(p)
print(json.dumps(d, ensure_ascii=False, indent=2))
```

```
{
  "version": "1",
  "tipo": "ProblemaTipo",
  "nombre": "",
  "setup": null,
  "componentes": [
    "Considere ",
    [
```

```

{
  "tipo": "Supuesto",
  "enunciado": "$mathcal{A}$, ",
  "semantica": "v('A')",
  "precond": "True"
},
{
  "tipo": "Supuesto",
  "enunciado": "$mathcal{B}$, ",
  "semantica": "v('B')",
  "precond": "True"
}
],
[
  {
    "tipo": "Supuesto",
    "enunciado": "y que $mathcal{A} \rightarrow \mathcal{C}$.",
    "semantica": "v('A') >> v('C')",
    "precond": "True"
  },
  {
    "tipo": "Supuesto",
    "enunciado": "y que $mathcal{B} \leftrightarrow \mathcal{D}$.",
    "semantica": "v('B') ** v('D')",
    "precond": "True"
  }
],
"Conteste: ",
[
  {
    "tipo": "Cuestion",
    "enunciado": "¿Se da $mathcal{C}$?",
    "semantica": "v('C')",
    "precond": "True",
    "exp": "Solo si $mathcal{A}$ y $mathcal{A} \rightarrow \mathcal{C}$"
  },
  {
    "tipo": "Cuestion",
    "enunciado": "¿Se da $mathcal{D}$?",
    "semantica": "v('D')",
    "precond": "True",
    "exp": ""
  }
]
]
}

```

Para guardar en un fichero:

```
save_problema(p, "ejemplosManual/ejercicio.json")
```

14.2. De JSON a lista de listas

Para recuperar el problema, `load_problema` devuelve un `ProblemaTipo` listo para iterar:

```
p2 = load_problema("ejemplosManual/ejercicio.json")
for etiqueta, enunciado, cuestiones in p2:
    print(f"Variante {etiqueta}: {enunciado}")
    for texto, correcto, activa, exp in cuestiones:
        marca = "√" if correcto else "×"
        print(f" {marca} {texto}")
```

Variante 1: Considere \mathcal{A} , y que $\mathcal{A} \rightarrow \mathcal{C}$. Conteste:
 ✓ ¿Se da \mathcal{C} ?

Variante 2: Considere \mathcal{A} , y que $\mathcal{A} \rightarrow \mathcal{C}$. Conteste:
 × ¿Se da \mathcal{D} ?

Variante 3: Considere \mathcal{A} , y que $\mathcal{B} \leftarrow \mathcal{D}$. Conteste:
 × ¿Se da \mathcal{C} ?

Variante 4: Considere \mathcal{A} , y que $\mathcal{B} \leftarrow \mathcal{D}$. Conteste:
 × ¿Se da \mathcal{D} ?

Variante 5: Considere \mathcal{B} , y que $\mathcal{A} \rightarrow \mathcal{C}$. Conteste:
 × ¿Se da \mathcal{C} ?

Variante 6: Considere \mathcal{B} , y que $\mathcal{A} \rightarrow \mathcal{C}$. Conteste:
 × ¿Se da \mathcal{D} ?

Variante 7: Considere \mathcal{B} , y que $\mathcal{B} \leftarrow \mathcal{D}$. Conteste:
 × ¿Se da \mathcal{C} ?

Variante 8: Considere \mathcal{B} , y que $\mathcal{B} \leftarrow \mathcal{D}$. Conteste:
 ✓ ¿Se da \mathcal{D} ?

El problema cargado se comporta exactamente igual que el original. También es posible cargar directamente desde un dict sin pasar por fichero, usando `problema_from_dict(d)`.

14.3. Con setup paramétrico

Cuando el problema usa `setup`, el código Python del `setup` debe escribirse como una **cadena de texto** dentro del dict JSON (no como una función Python). De este modo el fichero JSON es autosuficiente: al cargarlo, `load_problema` reconstruye el callable a partir de esa cadena y lo ejecuta al comienzo de cada iteración.

El siguiente ejemplo genera una pregunta sobre rango de matrices en la que las dimensiones n (filas) y k (columnas) cambian en cada variante:

```

from qbank import problema_from_dict, save_problema, load_problema

d = {
    "version": "1",
    "tipo": "ProblemaTipo",
    "nombre": "matriz_rango",
    "setup": (
        "import random\n"
        "n = random.randint(3, 5)\n"
        "k = random.randint(2, n - 1)"
    ),
    "componentes": [
        "Considere una matriz  $A$  de  $n$  filas y  $k$  columnas.",
        [
            {
                "tipo": "Supuesto",
                "enunciado": "Suponga que  $A$  tiene rango máximo ( $\text{rg}(A) = k$ ).",
                "semantica": "v('MaxRk')",
                "precond": "True"
            },
            {
                "tipo": "Supuesto",
                "enunciado": "Suponga que  $A$  tiene rango deficiente ( $\text{rg}(A) < k$ ).",
                "semantica": "-v('MaxRk')",
                "precond": "True"
            }
        ],
        "Indique qué afirmación es correcta: ",
        [
            {
                "tipo": "Cuestion",
                "enunciado": "Las  $k$  columnas de  $A$  son linealmente independientes.",
                "semantica": "v('MaxRk')",
                "precond": "True",
                "exp": ""
            },
            {
                "tipo": "Cuestion",
                "enunciado": " $A^{\text{top}}$  es invertible.",
                "semantica": "v('MaxRk')",
                "precond": "True",
                "exp": ""
            }
        ]
    ]
}

```

```

}

p = problema_from_dict(d)
save_problema(p, "ejemplosManual/matriz_rango.json")

```

La estructura combina 2 supuestos \times 2 cuestiones = 4 variantes. En cada una, el `setup` llama a `random.randint` de nuevo, por lo que n y k toman valores distintos. Los marcadores `@n` y `@k` en los textos reciben esos valores; la semántica de las cuestiones (`v('MaxRk')`) la evalúa el motor proposicional en función del supuesto activo:

```

import random
random.seed(7)

p2 = load_problema("ejemplosManual/matriz_rango.json")
for etiqueta, enunciado, cuestiones in p2:
    print(f"Variante {etiqueta}: {enunciado}")
    for texto, correcto, activa, exp in cuestiones:
        marca = "✓" if correcto else "×"
        print(f" {marca} {texto}")

```

Variante 1: Considere una matriz A de 4 filas y 2 columnas. Suponga que A tiene rango máximo ($\mathrm{rang}(A) = 2$).
 ✓ Las 2 columnas de A son linealmente independientes.

Variante 2: Considere una matriz A de 4 filas y 2 columnas. Suponga que A tiene rango máximo ($\mathrm{rang}(A) = 2$).
 ✓ $A^{\top} A$ es invertible.

Variante 3: Considere una matriz A de 3 filas y 2 columnas. Suponga que A tiene rango deficiente ($\mathrm{rang}(A) < 2$).
 × Las 2 columnas de A son linealmente independientes.

Variante 4: Considere una matriz A de 4 filas y 2 columnas. Suponga que A tiene rango deficiente ($\mathrm{rang}(A) < 2$).
 × $A^{\top} A$ es invertible.

Nótese que 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 exclusivamente el supuesto elegido en cada variante. El `setup` solo aporta los números que aparecen en el texto mediante la interpolación `@variable`. Una vez guardado el fichero, `load_problema` reconstituye el `setup` desde la cadena de código y el problema se comporta exactamente igual que el original.

14.4. Campo `seed`: reproducibilidad independiente del orden

Cuando varios problemas con `setup` aleatorio se generan en secuencia (en un bucle o un `Makefile`), la realización numérica de cada problema depende del estado del generador en ese momento. Si se añade o reordena un problema, cambian las realizaciones de todos los que le siguen.

El campo opcional `seed` (entero) resuelve esto: antes de iterar las variantes del problema, `qbank` inicializa `numpy.random.seed(seed)`. Así cada problema tiene su propia secuencia reproducible, independiente del orden en que se generen los demás.

```

p = ProblemaTipo(..., setup=mi_setup, seed=42)

# Equivalente desde JSON:
# { "seed": 42, "setup": "...", "componentes": [...] }

```

El campo `seed` también actúa como semilla base en `por_partes(instances=N)` cuando no se pasa `base_seed` explícitamente.

Si el problema no usa `numpy` en el `setup`, `seed` se ignora sin error.

14.5. Limitaciones

- Los `setup` definidos como funciones Python (`def` o `lambda`) no se pueden guardar en JSON porque su código fuente no es recuperable. Si necesitas un `setup` paramétrico y también quieres persistirlo, escribe el código del `setup` como cadena de texto y usa `problema_from_dict` (o el editor visual, que gestiona esto automáticamente).
- `ProblemaVF` se puede guardar y cargar sin restricciones (sus datos son valores directos).

Capítulo 15

Exportar la lista de listas a un fichero .py editable

Además de guardar en JSON, es posible exportar cualquier `ProblemaTipo` como código Python puro: exactamente la lista de listas con `Supuesto` y `Cuestion` que se escribiría a mano. El fichero resultante puede abrirse con cualquier editor de texto y ejecutarse directamente con Python.

```
from qbank import *

p = ProblemaTipo([
    "Dado que ",
    [
        Supuesto("$A$ es verdadero, ", v('A')),
        Supuesto("$B$ es verdadero, ", v('B')),
    ],
    "indique qué opción es correcta: ",
    [
        Cuestion("$A$ es verdadero.", v('A')),
        Cuestion("$B$ es verdadero.", v('B')),
    ],
])

from qbank import problema_to_python
print(problema_to_python(p))
```

```
from qbank import Supuesto, Cuestion, ProblemaTipo
from calccprop import *
```

```
ejercicio = [
    'Dado que ',
    [
        Supuesto('$A$ es verdadero, ', v('A')),
        Supuesto('$B$ es verdadero, ', v('B')),
    ],
    'indique qué opción es correcta: ',
    [
        Cuestion('$A$ es verdadero.', v('A')),
        Cuestion('$B$ es verdadero.', v('B')),
    ],
]
```

```
p = ProblemaTipo(ejercicio)
```

Para guardar directamente en un fichero:

```
from qbank import save_problema_py
save_problema_py(p, "./tmp/mi_problema.py")
```

El fichero `mi_problema.py` es código Python válido que puede editarse y ejecutarse. Si el problema tiene `setup`, la función de generación de valores aleatorios aparece como `def _setup()`: en el fichero exportado, con su cuerpo exactamente como fue guardado en el JSON.

Parte IV

Ejemplos progresivos

Los cinco ejemplos de `ejemplosManual/` ilustran las mecánicas fundamentales de forma acumulativa. Cada ejemplo añade exactamente una idea nueva sobre el anterior.

```
from qbank import load_problema
```

Capítulo 16

Ejemplo A — T/F literales, tres sublistas (ej_a_literales.json)

La forma más simple de `ProblemaTipo`: sin `setup`, sin variables proposicionales. Los valores de verdad son literales `True` / `False`. Las tres sublistas de cuestiones generan variantes por combinación ($5 \times 5 \times 5 = 125$).

```
p = load_problema("ejemplosManual/ej_a_literales.json")
variantes = list(p.por_partes())
print(f"{len(variantes)} variantes")
etiqueta, partes = variantes[0]
enunciado, cuestiones = partes[0]
print(enunciado)
for texto, vf, *_ in cuestiones:
    print(f" [{ '✓' if vf else '×' }] {texto}")
```

Mecánica clave: el `Marcador` itera todas las combinaciones de una cuestión por sublista. Con tres sublistas de 5 elementos cada una se obtienen $5 \times 5 \times 5 = 125$ variantes válidas. El valor de verdad de cada cuestión es un literal y no depende de ningún contexto.

Capítulo 17

Ejemplo B — Supuesto en lista, dos mundos proposicionales (ej_b_mundos.json)

Una lista de `Supuesto` crea mundos proposicionales alternativos: el `Marcador` elige uno y añade su semántica al mundo. Las cuestiones cuya semántica depende de la variable proposicional resultante (`v('crec')` o `v('decr')`) tienen valores de verdad distintos en cada mundo.

```
p = load_problema("ejemplosManual/ej_b_mundos.json")
variantes = list(p.por_partes())
print(f"{len(variantes)} variantes")
for etiqueta, partes in variantes[:2]:
    enunciado, cuestiones = partes[0]
    print(f"\nV{etiqueta}: {enunciado.strip()}")
    for texto, vf, *_ in cuestiones:
        print(f" [{ '✓' if vf else '×' }] {texto}")
```

Mecánica clave: la lista de `Supuesto` genera un mundo por opción. La semántica de cada `Cuestion` es una fórmula proposicional evaluada en ese mundo, no un literal fijo. La misma cuestión puede ser verdadera en un mundo y falsa en otro.

Capítulo 18

Ejemplo C — Axioma vacío y precond por mundo (ej_c_precond.json)

Un Supuesto suelto con `enunciado:` actúa como axioma: añade su fórmula al mundo sin contribuir texto visible. Aquí establece que la matriz es invertible o singular (`v('inv')` | `v('sing')`). Las cuestiones usan `precond` proposicional para aparecer solo en el mundo donde son relevantes.

```
p = load_problema("ejemplosManual/ej_c_precond.json")
variantes = list(p.por_partes())
print(f"{len(variantes)} variantes")
for etiqueta, partes in variantes[:2]:
    enunciado, cuestiones = partes[0]
    print(f"\nV{etiqueta}: {enunciado.strip()}")
    for texto, vf, *_ in cuestiones:
        print(f" [{ '✓' if vf else '×' }] {texto}")
```

Mecánica clave: `precond: "v('inv')"` hace que una cuestión solo sea elegible en el mundo donde A es invertible. El Marcador descarta las combinaciones donde la precondición no se satisface, reduciendo el número de variantes válidas respecto al total de combinaciones.

Capítulo 19

Ejemplo D — setup paramétrico con interpolación @{var} (ej_d_setup_vars.json)

El campo `setup` genera valores aleatorios antes de cada variante. Esos valores se interpolan en los textos con `@{nombre}`. El valor de verdad de las cuestiones sigue siendo proposicional (depende del mundo elegido por el Marcador, no de los números concretos), pero el enunciado muestra los valores reales.

```
p = load_problema("ejemplosManual/ej_d_setup_vars.json")
variantes = list(p.por_partes(instances=3))
print(f"{len(variantes)} variantes con instances=3")
for etiqueta, partes in variantes[:4]:
    enunciado, cuestiones = partes[0]
    print(f"\nV{etiqueta}: {enunciado[50:].strip()}")
    for texto, vf, *_ in cuestiones:
        print(f" [{ '✓' if vf else '×' }] {texto}")
```

Mecánica clave: `instances=N` ejecuta el `setup` N veces con semillas distintas, multiplicando las variantes estructurales. El `@{var}` hace que cada instancia muestre valores concretos distintos en el enunciado, aunque la lógica T/F sea la misma.

Capítulo 20

Ejemplo E — Semántica lambda e inyección numérica (ej_e_lambda.json)

Cuando el valor de verdad depende de un valor numérico calculado en `setup` (no de variables proposicionales), se usa una **semántica lambda**. El `setup` calcula fórmulas proposicionales ($v('gt_u1')$ o $\neg v('gt_u1')$) según el valor de `x`, y el `Supuesto` con `lambda` las inyecta en el mundo. Los axiomas iniciales establecen la jerarquía $x > u_2 \rightarrow x > u_1$, que el sistema proposicional usa para inferir respuestas consistentes.

```
p = load_problema("ejemplosManual/ej_e_lambda.json")
variantes = list(p.por_partes(instances=5))
print(f"{len(variantes)} variantes con instances=5")
for etiqueta, partes in variantes:
    enunciado, cuestiones = partes[0]
    x_val = enunciado.split("x = ")[1].split(".")[0].strip()
    vfs = ['✓' if vf else '×' for _, vf, *_ in cuestiones]
    print(f" V{etiqueta}: x={x_val} {vfs}")
```

Mecánica clave: la `lambda` `lambda ns: ns['sup_u1'] & ns['sup_u2']` evalúa las fórmulas proposicionales precomputadas en `ns` y las inyecta en el mundo. Las cuestiones usan semánticas proposicionales puras ($v('gt_u1')$, $v('gt_u1') \& \neg v('gt_u2')$) que el sistema evalúa contra ese mundo. Los axiomas garantizan coherencia: si el sistema sabe que $x > u_2$, infiere automáticamente que $x > u_1$.

Parte V

Exportar a AMC ($\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$)

Las funciones AMC generan el código \LaTeX compatible con Auto Multiple Choice. Cada función devuelve una cadena de texto con el bloque $\text{\element}\{...\}$ correspondiente.

Capítulo 21

Funciones disponibles

Capítulo 22

Función unificada `AMCblock`

La función principal de exportación es `AMCblock`. Los comportamientos que antes requerían funciones separadas se controlan con parámetros:

Parámetro	Tipo	Descripción
<code>nombre</code>	<code>str</code>	Nombre del grupo (etiqueta \LaTeX). Los <code>:</code> se convierten en <code>-</code> automát.
<code>etiqueta</code>	<code>str</code>	Identificador único de esta variante.
<code>enunciado</code>	<code>str</code>	Texto del enunciado ($$$\dots$$$ se convierte a $\[...\]$).
<code>cuestiones</code>	<code>list</code>	Lista de tuplas (<code>texto</code> , <code>correcto</code> , <code>activa</code> [, <code>exp</code>]).
<code>last_choice</code>	<code>bool</code>	Si <code>True</code> , añade opción comodín con <code>\lastchoices</code> .
<code>cols</code>	<code>int</code>	Columnas (>1 activa <code>\begin{multicols}{N}\AMCBoxedAnswers</code>).
<code>profe</code>	<code>bool</code>	Si <code>True</code> , añade <code>\explain{}</code> con cuestiones rechazadas.
<code>aux_latex</code>	<code>str</code>	Instrucciones \LaTeX extra dentro del bloque <code>\element{}</code> .
<code>last_choice_text</code>	<code>str</code>	Texto de la opción comodín (solo si <code>last_choice=True</code>).

Para preguntas de tipo `ProblemaVF` existe `AMC_VF`, que sigue la misma estructura pero recibe tuplas (`texto`, `True/False`) sin el flag de descarte por precondition.

Capítulo 23

Firma

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

Capítulo 24

Ejemplo AMC básico

```
from qbank import *

p = ProblemaTipo([
    "Dado que ",
    [
        Supuesto("$\\mathcal{A}$ es verdadero. ", v("A")),
        Supuesto("$\\mathcal{B}$ es verdadero. ", v("B")),
    ],
    "indique qué opción es correcta: ",
    [
        Cuestion("$\\mathcal{A}$ es verdadero", v("A")),
        Cuestion("$\\mathcal{B}$ es verdadero", v("B")),
    ],
])

nombre = "MiCuestionario"
with open("preguntas.tex", "w") as f:
    for etiqueta, enunciado, cuestiones in p:
        f.write(AMCblock(nombre, etiqueta, enunciado, cuestiones))
```

Capítulo 25

Ejemplo AMC multicolumna con «Ninguna de las anteriores»

Usando el mismo p del ejemplo anterior, con `last_choice=True` y `cols=2`:

```
with open("preguntas.tex", "w") as f:
    for etiqueta, enunciado, cuestiones in p:
        f.write(AMCblock("MiCuestionario", etiqueta, enunciado, cuestiones,
                        last_choice=True, cols=2))
```

Los ítems se distribuyen en dos columnas y se añade automáticamente «Ninguna de las anteriores» como última opción.

Capítulo 26

Vista aplanada del profesor: QuizAMCProfe

QuizAMCProfe es el exportador de alto nivel para la vista profe. A diferencia de la vista alumno (que muestra una cuestión por sublista), la vista aplanada muestra **todas** las cuestiones de cada sublista junto al enunciado, con marcas de correcto/incorrecto. Las cuestiones rechazadas por precondition aparecen en un bloque `\explain{}` al final de la pregunta.

```
QuizAMCProfe(nombre, directorio, problema, cols=1,  
             aux_latex="", last_choice_text="Ninguna de las anteriores")  
# Devuelve int: número de bloques AMC escritos
```

Parámetro	Descripción
nombre	Identificador del grupo AMC.
directorio	Ruta donde se guardará el <code>.tex</code> (debe terminar en <code>/</code>).
problema	Un <code>ProblemaTipo</code> , o un diccionario <code>{nombre: ProblemaTipo}</code> .
cols	Columnas (>1 activa <code>\begin{multicols}{N}\AMCBoxedAnswers</code>).
aux_latex	Instrucciones \LaTeX extra dentro del bloque <code>\element{}</code> .
last_choice_text	Texto de la opción comodín.

Genera `nombre_profe.tex` (con sufijo `_profe`). Se incluye en el documento AMC con:

```
\input{nombre_profe.tex}
```

Ejemplo con el mismo `p` de la sección anterior:

```
QuizAMCProfe("MiCuestionario", "exportaciones/", p)
```

Diferencia clave con QuizMoodleProfe: QuizAMCProfe usa `por_partes_profe()` y muestra la vista aplanada (todas las cuestiones). QuizMoodleProfe exporta a Moodle las mismas variantes que el alumno pero con fracciones de puntuación visibles. Las dos vistas son complementarias y tienen propósitos distintos.

Capítulo 27

Integración en el documento AMC

El código generado debe incluirse en el preámbulo del documento principal de AMC:

```
\input{preguntas.tex}
```

Consulta la documentación de AMC para el formato completo del documento.

Parte VI

Exportar a Moodle (vía L^AT_EX)

Las funciones `Quiz*` generan un fichero `.tex` que, al compilarlo con `LATEX` (usando el paquete `moodle`), produce un fichero `.xml` importable en Moodle.

Capítulo 28

Flujo de trabajo

Python (qbank) → .tex → pdflatex/xelatex → .xml → Moodle

El paquete \LaTeX `moodle` genera el `.xml` durante la compilación. Para instalarlo:

```
tlmgr install moodle
```

O en distribuciones Debian/Ubuntu:

```
sudo apt install texlive-latex-extra
```

Capítulo 29

Funciones disponibles

Función	Tipo de problema	<code>last_choice</code>	Notas
<code>QuizMoodle</code>	<code>ProblemaTipo</code>	parámetro	Función principal (vista alumno)
<code>QuizMoodleProfe</code>	<code>ProblemaTipo</code>	No	Vista alumno con fracciones de puntuación visibles
<code>QuizVFMoodle</code>	<code>ProblemaVF</code>	—	Verdadero/Falso

Para la vista aplanada (todas las cuestiones con marcas), usa `QuizAMCProfe` (sección [Vista aplanada del profesor](#)).

La **opción rescate** («Las demás opciones son falsas» por defecto) se añade automáticamente al final de cada bloque `\begin{multi}` y es correcta si y solo si ninguna otra opción lo es. Moodle rechaza la importación de preguntas sin ninguna opción correcta, por lo que `last_choice=True` es la elección segura salvo que el problema garantice siempre al menos una opción verdadera.

Capítulo 30

Firma

ProblemaTipo (una o varias partes):

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

ProblemaVF:

```
QuizVFMoodle(nombre, directorio, GenVar, num, opc=None)
```

Parámetro	Descripción
nombre	Nombre del cuestionario (aparece en Moodle como categoría).
directorio	Ruta donde se guardará el <code>.tex</code> (debe terminar en <code>/</code>).
problema	Un <code>ProblemaTipo</code> , o un diccionario <code>{nombre: ProblemaTipo}</code> .
last_choice	Si <code>True</code> , añade la opción comodín al final de cada cuestión.
aux_latex	Cabecera <code>L^AT_EX</code> adicional (p.ej. <code>\usepackage{nacal-moodle}</code>).
last_choice_text	Texto de la opción comodín.
instances	Número de instancias con semillas distintas (requiere <code>setup</code>).
GenVar	Iterador de variantes <code>ProblemaVF</code> .
num	Número de variantes a exportar.

Capítulo 31

Ejemplo Moodle completo

Creamos el directorio donde vamos a guardar el ejemplo:

```
mkdir -p ejemplosManual
```

El siguiente código genera el fichero de L^AT_EX:

```
from qbank import *

p = ProblemaTipo([
    "Considere ",
    [
        Supuesto("$\\mathcal{A}$ ", v("A")),
        Supuesto("$\\mathcal{B}$ ", v("B")),
    ],
    [
        Supuesto("y que $\\mathcal{A}\\rightarrow\\mathcal{C}$.", v("A") >> v("C")),
        Supuesto("y que $\\mathcal{B}\\rightarrow\\mathcal{D}$.", v("B") ** v("D")),
    ],
    "Indique qué opción es correcta: ",
    [
        Cuestion("Entonces $\\mathcal{C}$ es verdadero", v("C")),
        Cuestion("Entonces $\\mathcal{D}$ es verdadero", v("D")),
    ],
])

QuizMoodle("EjemploMoodle", "ejemplosManual/", p, last_choice=True)
```

Esto crea salida/EjemploMoodle.tex. Para obtener el .xml y una versión .pdf del conjunto de preguntas basta ejecutar:

```
cd ejemplosManual
pdflatex EjemploMoodle.tex
```

Se generan [EjemploMoodle-moodle.xml](#) (para importar en Moodle) y [EjemploMoodle.pdf](#) (vista previa del aspecto de las preguntas).

Capítulo 32

Ejemplo con ProblemaVF para Moodle

El siguiente código genera el fichero de L^AT_EX:

```
from qbank import *

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),
    ("Las eñes son letras frecuentes en inglés", False),
]

b = [(codchar(texto), correcto) for texto, correcto in banco]
GenVar = iter(ProblemaVF(codchar(enunciado), b, 2)) # 2 items por variante

QuizVFMoodle("EjemploVF", "ejemplosManual/", GenVar, num=5) # 5 variantes
```

Para obtener el .xml y una versión .pdf del conjunto de preguntas basta ejecutar:

```
cd ejemplosManual
pdflatex EjemploVF.tex
```

Nota: `codchar()` convierte los caracteres con tilde y la ñ a secuencias L^AT_EX (p.ej. á → \'a). Es necesario para Moodle porque el .tex usa codificación OT1.

Parte VII

Preguntas multi-parte

`ProblemaTipo` admite ejercicios con varias partes. La lista plana de componentes determina la estructura: una transición de un bloque de cuestiones a un bloque de texto o supuestos abre una parte nueva. El método `por_partes()` devuelve `(etiqueta, [(enunciado, cuestiones), ...])` con un par por parte.

Capítulo 33

Ejemplo

El fichero `ejemplosManual/superheroes.json` contiene un ejercicio de tres partes sobre distintos superhéroes.

```
from qbank import *
import itertools

p = load_problema("ejemplosManual/superheroes.json")

for etiqueta, partes in itertools.islice(p.por_partes(), 2):
    print(f"=== Variante {etiqueta} ===")
    for enunciado, cuestiones in partes:
        if enunciado:
            print(f" {enunciado}")
        for texto, correcto, _, exp in cuestiones:
            marca = '✓' if correcto else '×'
            print(f" {marca} {texto}")
    print()
```

Capítulo 34

Exportar a AMC

```
with open("preguntas_multi.tex", "w") as f:
    for etiqueta, partes in p.por_partes():
        f.write(AMC_multipart("Superheroes", etiqueta, "", partes))

with open("preguntas_multi.tex") as f:
    print(f.read())
```

Capítulo 35

Exportar a Moodle (cloze)

```
import os
os.makedirs("ejemplosManual", exist_ok=True)
QuizMoodle("Superheroes", "ejemplosManual/", p, last_choice=True)
```

Esto genera `ejemplosManual/Superheroes.tex`. Compila con `xelatex` para obtener el `.xml`:

```
cd ejemplosManual
xelatex Superheroes.tex
```

El fichero `Superheroes.xml` puede importarse en Moodle (Banco de preguntas → Importar → Formato Moodle XML). Cada pregunta cloze contiene tantos bloques `\begin{multi}` como partes tenga el ProblemaTipo.

Capítulo 36

Persistencia JSON

ProblemaTipo (también multiparte) se guarda y carga con `save_problema` y `load_problema`:

```
from qbank import save_problema, load_problema

save_problema(p, "ejemplosManual/superheroes.json")
p2 = load_problema("ejemplosManual/superheroes.json")
```

Capítulo 37

Exportar a código Python editable

```
from qbank import problema_to_python
print(problema_to_python(p, varname="superheroes"))
```

Capítulo 38

Compatibilidad: ProblemaMultiParte y SubPregunta (deprecados)

`ProblemaMultiParte` y `SubPregunta` siguen disponibles pero están *deprecados*: al construir un `ProblemaMultiParte` se emite un `DeprecationWarning` y el objeto devuelto es un `ProblemaTipo` equivalente. Para código nuevo, define directamente un `ProblemaTipo` con la lista plana de componentes y recórrelo con `.por_partes()`.

Parte VIII

Editor visual en Jupyter (ProblemaTipoEditor)

`ProblemaTipoEditor` es un formulario interactivo para diseñar problemas de tipo `ProblemaTipo` desde un notebook de Jupyter, sin necesidad de escribir la estructura de listas a mano.

Capítulo 39

Instalación

El editor requiere `ipywidgets`. Instálalo junto con `calcprop-qbank`:

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

En **JupyterLab 4** es necesario que la extensión `@jupyter-widgets/jupyterlab-manager` esté habilitada. Compruébalo con:

```
jupyter labextension list
```

Si la extensión no aparece (síntoma: el widget se muestra como texto `VBox(children...=)`), ejecuta una sola vez desde el terminal:

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

Reinicia JupyterLab para que el cambio surta efecto. (En Jupyter Notebook clásico no hace falta este paso.)

Capítulo 40

Uso básico

Abre un notebook con el kernel del entorno donde instalaste `calcprop-qbank`:

```
from qbank import ProblemaTipoEditor
```

Para empezar un problema nuevo:

```
editor = ProblemaTipoEditor()
```

Para cargar un problema desde un fichero JSON existente:

```
editor = ProblemaTipoEditor('mi_problema.json')
```

Capítulo 41

Descripción del formulario

El editor tiene tres zonas:

- **Cabecera:** campo **Nombre** (identificador del problema) y campo **Setup** (código Python del setup paramétrico; déjalo vacío si no necesitas valores variables).
- **Slots:** cada slot es un «componente» del problema. Pulsa + **slot** para añadir slots. Cada slot puede ser de tipo:
 - **texto:** texto fijo que aparece igual en todas las variantes.
 - **alternativas:** una o más filas, cada una con tipo (**Supuesto / Cuestion**), enunciado, semántica, precondition y, para **Cuestion**, un campo de explicación/feedback. Pulsa + **alt** para añadir filas.
- **Barra inferior:** controles de acción.

Botón	Acción
Preview	Genera hasta N variantes y las muestra en el área de salida.
Guardar	Serializa el estado actual y lo guarda en el fichero indicado (campo de texto JSON).
Cargar	Lee el fichero indicado y rellena el editor con su contenido.
{ } JSON	Muestra el JSON equivalente al estado actual del editor en el área de salida.

Capítulo 42

Ejemplo sin setup

Construir desde el editor el mismo problema del ejemplo de ProblemaTipo:

1. Ejecutar `ProblemaTipoEditor()` en una celda.
2. Dejar el campo `Setup` vacío.
3. + slot → tipo **texto** → escribir "Dado que ".
4. + slot → tipo **alternativas** → + alt dos veces:
 - fila 1: tipo **Supuesto**, enunciado \mathcal{A} es verdadero, =, semántica $v('A')$
 - fila 2: tipo **Supuesto**, enunciado \mathcal{B} es verdadero, =, semántica $v('B')$
5. + slot → tipo **texto** → escribir indique qué opción es correcta: ".
6. + slot → tipo **alternativas** → + alt dos veces:
 - fila 1: tipo **Cuestion**, enunciado \mathcal{A} es verdadero, semántica $v('A')$
 - fila 2: tipo **Cuestion**, enunciado \mathcal{B} es verdadero, semántica $v('B')$
7. Pulsar `Preview` (4 variantes esperadas).
8. Escribir `mi_problema.json` en el campo de ruta y pulsar `Guardar`.

Una vez guardado, el fichero puede cargarse desde código Python:

```
from qbank import load_problema
p = load_problema('mi_problema.json')
for etiqueta, enunciado, cuestiones in p:
    print(etiqueta, enunciado)
```

Capítulo 43

Ejemplo con setup paramétrico

Para un problema con valores variables, escribe en el campo **Setup** el código que genera los valores por variante:

```
import random
a = random.randint(1, 9)
b = random.randint(1, 9)
suma = a + b
```

Luego, en los textos de los slots, usa **@a**, **@b** y **@suma** donde corresponda. Por ejemplo, un slot de tipo **texto** con el contenido "**Sean \$a = @a\$ y \$b = @b\$.** " mostrará los valores aleatorios de cada variante.

Las semánticas que dependan de los valores del setup deben escribirse como lambdas:

```
lambda ns: ns['a'] + ns['b'] > 10
```

Capítulo 44

Obtener el problema desde el editor

Una vez diseñado el problema, puedes obtener el objeto Python directamente desde otra celda:

```
p = editor.to_problema() # ProblemaTipo listo para iterar  
d = editor.to_dict()    # dict JSON equivalente
```

Con `p` ya puedes llamar a las funciones de exportación (`AMCblock`, `QuizMoodle`, etc.) como si hubieras definido el problema a mano.

Capítulo 45

Flujo completo: editor → JSON → AMC y Moodle

Este ejemplo muestra el recorrido completo desde el editor visual hasta los ficheros de exportación, pasando por el JSON como punto de guardado intermedio.

45.1. Paso 1 — Diseñar y guardar desde el editor

Crea el problema en el editor y guárdalo en disco (botón `Guardar` o desde código):

```
from qbank import ProblemaTipoEditor

editor = ProblemaTipoEditor()
editor # muestra el formulario
```

Una vez rellenado el formulario, guarda el fichero:

```
editor.save("mi_problema.json")
```

O bien, si prefieres guardarlo a mano:

```
from qbank import save_problema
save_problema(editor.to_problema(), "mi_problema.json")
```

45.2. Paso 2 — Cargar el JSON

En cualquier sesión posterior (o en el mismo notebook, en otra celda):

```
from qbank import *

p = load_problema("mi_problema.json")
```

`p` es un `ProblemaTipo` completamente funcional, idéntico al que se habría obtenido definiendo el problema a mano en Python.

45.3. Paso 3a — Exportar a AMC

```
nombre = "MiCuestionario"

with open("preguntas_amc.tex", "w") as f:
    for etiqueta, enunciado, cuestiones in p:
        f.write(AMCblock(nombre, etiqueta, enunciado, cuestiones))
```

Variantes habituales de `AMCblock`:

Parámetros de AMCblock	Cuándo usarla
(por defecto)	Pregunta estándar (solo ítems activos).
<code>last_choice=True</code>	Añade «Ninguna de las anteriores» al final.
<code>cols=N</code>	Ítems en N columnas.
<code>last_choice=True, cols=N</code>	Multicolumna con «Ninguna de las anteriores».

El fichero generado se incluye en el documento principal de AMC con `\input{preguntas_amc.tex}`.

45.4. Paso 3b — Exportar a Moodle

```
QuizMoodle("MiCuestionario", "salida/", p, last_choice=True)
```

Esto genera `salida/MiCuestionario.tex`. A continuación, compila con \LaTeX para obtener el `.xml` importable en Moodle:

```
cd salida
xelatex MiCuestionario.tex
```

El paquete \LaTeX `moodle` produce `MiCuestionario.xml` durante la compilación. Ese fichero es el que se importa en Moodle (Banco de preguntas → Importar → Formato Moodle XML).

Variantes habituales de `QuizMoodle`:

Parámetros / función	Cuándo usarla
(por defecto)	Pregunta estándar.
<code>last_choice=True</code>	Añade «Las demás opciones son falsas».
<code>instances=N</code>	N realizaciones numéricas distintas (problemas con <code>setup</code>).
<code>QuizMoodleProfe(...)</code>	Mismas variantes que el alumno, con fracciones visibles.
<code>QuizAMCProfe(...)</code>	Vista aplanada: todas las cuestiones con marcas (PDF AMC).
<code>QuizVFMoodle(...)</code>	Para <code>ProblemaVF</code> (verdadero/falso).

45.5. Paso 4 — Generar múltiples instancias numéricas (problemas con `setup`)

Cuando el problema tiene `setup` con valores aleatorios, iterar una sola vez el problema genera tantas variantes como combinaciones estructurales existan (por ejemplo, 2 supuestos \times 2 cuestiones = 4). Cada variante llama a `setup()` de forma independiente, por lo que cada una puede tener distintos valores numéricos.

Hay dos patrones de exportación masiva:

Opción A — variantes totalmente independientes

Cada una de las 40 variantes tiene sus propios valores numéricos. Útil cuando el banco de preguntas debe ser lo más variado posible.

```
from qbank import *

p = load_problema("mi_problema.json")

with open("preguntas.tex", "w") as f:
    for instancia in range(10):
        for etiqueta, enunciado, cuestiones in p:
            f.write(AMCblock("MiCuestionario", f"{instancia+1}-{etiqueta}",
                             enunciado, cuestiones))
```

Las 4 variantes estructurales de cada vuelta del bucle externo no comparten necesariamente los mismos valores numéricos entre sí.

Opción B — lotes: las variantes estructurales comparten los mismos valores

Dentro de cada lote, las 4 variantes estructurales usan los mismos valores numéricos. El `setup` se reemplaza temporalmente por un callable que devuelve un dict fijo calculado al inicio del lote.

```

from qbank import *
import random

p = load_problema("mi_problema.json")
d = problema_to_dict(p)          # dict reutilizable

with open("preguntas.tex", "w") as f:
    for instancia in range(10):
        n = random.randint(3, 9)
        k = random.randint(2, n - 1)
        ns = {'n': n, 'k': k}

        p_lote = problema_from_dict(d)
        p_lote.setup = lambda ns=ns: ns    # fija n,k para todo el lote

        for etiqueta, enunciado, cuestiones in p_lote:
            f.write(AMCblock("MiCuestionario", f"{instancia+1}-{etiqueta}",
                             enunciado, cuestiones))

```

El lambda `ns=ns: ns` captura el valor del dict en el momento de creación mediante argumento por defecto, evitando que todas las lambdas apunten al mismo último valor de la variable de bucle.

Opción	Patrón	Valores por variante
A	<code>for i in range(10): for v in p:</code>	Cada variante tiene los suyos
B	<code>p_lote.setup = lambda ns=ns: ns</code>	Las 4 del lote comparten (n, k)

Ambos patrones funcionan igual para Moodle: sustituye el bloque `with open(...)` por `QuizMoodle("MiCuestionario", "salida/", p, last_choice=True)` y adapta el nombre.

Parte IX

Referencia rápida de la API

Capítulo 46

Clases

```
Supuesto(enunciado, semantica, precond=True)
Cuestion(enunciado, semantica, precond=True, exp="")
ProblemaTipo(lista, setup=None, export=None, seed=None)
    → itera (etiqueta, enunciado, cuestiones)
    → .por_partes(instances=1, base_seed=None, verbose=False) - vista alumno; multiparte e instancias numérica
    → .por_partes_profe() - vista aplanada: todas las cuestiones con marcas
ProblemaTipoProfe(lista, setup=None) [DEPRECADO → usar ProblemaTipo + .por_partes_profe()]
    → como ProblemaTipo, sin descartar ítems rechazados
ProblemaVF(enunciado, lista, n)
    → itera (etiqueta, enunciado, cuestiones)
SubPregunta(intro, cuestiones) [DEPRECADO → devuelve ProblemaTipo]
ProblemaMultiParte(componentes, subpreguntas, setup=None) [DEPRECADO → devuelve ProblemaTipo]
ProblemaTipoEditor(source=None)
    → editor visual en Jupyter (requiere ipywidgets; solo ProblemaTipo)
```

Los campos `semantica` y `precond` de `Supuesto` y `Cuestion` admiten, además de fórmulas `calcprop`, callables de la forma `lambda ns: ...` para problemas con `setup` paramétrico. Los textos (enunciados) admiten `@variable` para sustituir valores del namespace del `setup`.

Capítulo 47

Funciones JSON

```
problema_from_dict(d)           → ProblemaTipo o ProblemaVF
problema_to_dict(problema)     → dict
load_problema(filepath_o_dict) → ProblemaTipo o ProblemaVF # acepta ruta o dict
save_problema(problema, filepath)
load_banco(filepath)           → lista de problemas
save_banco(problemas, filepath) # acepta lista o dict
problema_to_python(problema, varname="ejercicio") → str (ProblemaTipo)
save_problema_py(problema, filepath, varname="ejercicio")
```

Capítulo 48

Funciones AMC

```
AMCblock      (nombre, etiqueta, enunciado, cuestiones,
               last_choice=False, cols=1, profe=False, *,
               aux_latex="", last_choice_text="Ninguna de las anteriores")
    # bajo nivel: devuelve str; el usuario escribe al fichero manualmente
AMC_VF        (nombre, etiqueta, enunciado, cuestiones, opc=None)
AMC_multipart(nombre, etiqueta, enunciado, partes, opc=None)
    # para ProblemaTipo multiparte; partes = [(enunciado, cuestiones), ...]
QuizAMCProfe(nombre, directorio, problema, cols=1,
               aux_latex="", last_choice_text="Ninguna de las anteriores")
    # alto nivel: vista aplanada profe → escribe nombre_profe.tex; devuelve int
    # usa por_partes_profe(); muestra todas las cuestiones con marcas y \explain{}
```

Capítulo 49

Funciones Moodle

```
QuizMoodle      (nombre, directorio, problema, last_choice=False,
                aux_latex="", last_choice_text="Las demás opciones son falsas",
                instances=1)
    # vista alumno; instances>1 requiere numpy y setup con aleatorios; devuelve int
QuizMoodleProfe (nombre, directorio, problema, aux_latex="")
    # mismas variantes que el alumno, con fracciones de puntuación visibles; devuelve int
QuizVFMoodle    (nombre, directorio, GenVar, num, opc=None)
```

Capítulo 50

Operadores de `calccprop` (disponibles tras `from qbank import *`)

```
v("X")           variable proposicional X
~v("X")          negación de X
v("X") & v("Y")  conjunción (X Y)
v("X") | v("Y")  disyunción (X Y)
v("X") >> v("Y") implicación (X → Y)
v("X") ** v("Y") bicondicional (X ↔ Y)
alguno(P, Q, R) al menos una de la lista es verdadera
unoDe(P, Q, R)  exactamente una de la lista es verdadera
True           tautología (siempre verdadero)
False          contradicción (siempre falso)
```

```
test(formula, lista_de_formulas) → bool
```

Devuelve True si formula es consecuencia lógica de la lista de hipótesis.

Parte X

Preguntas frecuentes

Capítulo 51

¿Por qué algunas variantes no se generan?

Si un Supuesto 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.

Capítulo 52

¿Cómo incluyo texto L^AT_EX en los enunciados?

Los campos `enunciado` de `Supuesto` y `Cuestion` son cadenas Python ordinarias que se insertan tal cual en el `.tex`. Usa cadenas crudas (`r"...`) para evitar problemas con las barras invertidas:

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

Capítulo 53

¿Cómo escribo `display math` en enunciados para AMC?

Usa `\[...]` en lugar de `$$...$$`. El exportador `AMCblock` convierte `$$...$$` a `\[...]` automáticamente en el enunciado y en las opciones, pero el paquete `automultiplechoice` a veces produce errores «Missing \$» con `$$` y los corrige de forma no fatal. Usar `\[...]` directamente evita el problema por completo.

Además, dos expresiones inline consecutivas sin espacio (`\$A\$\$B\$\$`) pueden confundirse con `display math` durante la conversión. Separa siempre expresiones inline contiguas con un espacio: `\$A\$ \$B\$\$` en lugar de `\$A\$\$B\$\$`.

Capítulo 54

¿Para qué sirve `codchar`?

La función `codchar(s)` convierte vocales acentuadas y la ñ a secuencias \LaTeX :

```
codchar("á é í ó ú ñ") # + "\\ 'a \\ 'e} \\ 'i} \\ 'o} \\ 'u} \\ -{n}"
```

Es necesaria solo para las funciones Moodle, que generan un `.tex` con codificación OT1. Para AMC no suele hacer falta si el documento principal usa `inputenc` con UTF-8.

Capítulo 55

¿Cómo genero un diccionario de preguntas de varias secciones?

Pasa un diccionario a las funciones Quiz*:

```
problema = {  
    "Sección-A": ProblemaTipo(lista_A),  
    "Sección-B": ProblemaTipo(lista_B),  
}  
QuizMoodle("MiExamen", "salida/", problema, last_choice=True)
```

Cada clave del diccionario se convierte en una categoría dentro del cuestionario Moodle.