

# Índice general

<b>I Creación de preguntas de opción múltiple</b>	<b>3</b>
<b>1. Clase Marcador</b>	<b>6</b>
1.1. Funcionamiento General . . . . .	6
1.2. Implementación . . . . .	6
1.2.1. Método <code>__init__(self, data)</code> . . . . .	7
1.2.2. Método <code>__iter__(self)</code> . . . . .	7
1.2.3. Método <code>__next__</code> . . . . .	7
<b>2. Clases Supuesto y Cuestion</b>	<b>9</b>
2.1. Clase <code>Supuesto(enunciado, semantica, precondition)</code> . . . . .	9
2.2. Clase <code>Cuestion(enunciado, semantica, precondition)</code> . . . . .	9
<b>3. Segmentación en partes y validación de homogeneidad</b>	<b>11</b>
3.1. Comprobación . . . . .	12
<b>4. Clase ProblemaTipo</b>	<b>13</b>
4.1. Implementación . . . . .	13
4.1.1. Generador interno <code>_variantes(self)</code> . . . . .	14
4.1.2. Generador <code>profe_variantes_profe(self)</code> . . . . .	15
4.1.3. Fachadas de iteración: <code>por_partes()</code> , <code>por_partes_profe()</code> , <code>__iter__</code> y <code>__next__</code> . . . . .	15
4.2. Ejemplo de funcionamiento . . . . .	16
4.3. Ejercicios multiparte: <code>por_partes()</code> . . . . .	19
<b>5. Clase ProblemaTipoProfe</b>	<b>20</b>
5.1. Implementación . . . . .	20
5.1.1. Método <code>__next__</code> para la clase <code>ProblemaTipoProfe</code> . . . . .	20
5.1.2. Función auxiliar <code>CuestionesJuntas</code> . . . . .	21
5.2. Ejemplo de funcionamiento . . . . .	22
<b>6. Clase ProblemaVF</b>	<b>24</b>
6.1. Implementación . . . . .	24
6.1.1. Métodos de la clase <code>ProblemaVF</code> . . . . .	24
6.2. Ejemplo de funcionamiento . . . . .	25
6.2.1. Generación de cuatro variantes cortas . . . . .	25
6.2.2. Generación de cinco variantes con presentación formateada . . . . .	25
<b>7. Clases <code>SubPregunta</code> y <code>ProblemaMultiParte</code> (deprecadas)</b>	<b>27</b>
7.1. Ejemplo . . . . .	27
<b>II Serialización y persistencia en formato JSON</b>	<b>29</b>
<b>8. Formato JSON</b>	<b>32</b>
8.1. Para un <code>ProblemaTipo</code> . . . . .	32
8.2. Para un <code>ProblemaVF</code> . . . . .	33

9. Evaluación de expresiones en el espacio de nombres de calcprop	34
10. Función de setup desde código Python	35
11. Deserialización de componentes	36
12. Serialización de componentes	37
13. Serialización de problemas completos	38
14. Lectura y escritura de ficheros	40
15. Ejemplo de uso	41
16. Exportación a código Python editable	42
16.1. <code>_slot_to_python</code> . . . . .	42
16.2. <code>problema_to_python</code> y <code>save_problema_py</code> . . . . .	43
16.3. Ejemplo . . . . .	44
III Editor visual de problemas en Jupyter	45
17. Dependencias e instalación	47
18. Widget de alternativa ( <code>_AltWidget</code> )	49
19. Widget de slot ( <code>_SlotWidget</code> )	51
20. Editor principal ( <code>ProblemaTipoEditor</code> )	54
IV ¿Y ahora qué?	59
20.1. ¿Qué tenemos? . . . . .	60
20.2. ¿Qué nos falta? . . . . .	60

## Parte I

# Creación de preguntas de opción múltiple

## Estructura del módulo `qbank._quiz`

El módulo completo está compuesto varios fragmentos de código se son descritos en los siguientes capítulos.

---

```
qbank/_quiz.py
from calccprop import *
from string import Template

<<Funciones auxiliares para setup paramétrico>>
<<Metodo auxiliar para juntar cuestiones en formato para profe>>
<<Definición de la clase Marcador>>
<<Definición de la clase Supuesto>>
<<Definición de la clase Cuestion>>
<<Segmentación en partes y validación de homogeneidad>>
<<Definición de la clase ProblemaTipo>>
<<Definición de la clase ProblemaTipoProfe>>
<<Definición de la clase ProblemaVF>>
<<Definición de SubPregunta y ProblemaMultiParte>>

<<Interfaz gráfica para el diseño visual de la lista de listas de ejercicios>>
```

---

## Uso del módulo

Para usar este módulo, primero se debe ejecutar

---

```
from qbank import *
```

---

El fichero `__init__.py` del paquete reúne la interfaz pública reexportando los submódulos. El editor visual (`_widgets`) se importa dentro de un `try/except` porque depende de `ipywidgets`, que es una dependencia opcional: si no está instalada, el resto del paquete sigue funcionando.

---

```
qbank/__init__.py
from qbank._quiz import *
from qbank._export import *
from qbank._json import *
try:
    from qbank._widgets import *
except ImportError:
    pass
```

---

# Funciones auxiliares para setup paramétrico

Las funciones `_ns_eval` y `_ns_interp` permiten que los atributos de `Supuesto` y `Cuestion` (enunciado, semántica, precondition) sean valores directos o **callable** que reciben el namespace generado por el `setup` de `ProblemaTipo`.

`_ns_eval(val, ns)` si `val` es callable, lo invoca con `ns` y devuelve el resultado; si no, devuelve `val` tal cual.

`_ns_interp(s, ns)` interpola en la cadena `s` los marcadores `@variable` usando el diccionario `ns`. Usa `string.Template.safe_substitute` con delimitador `@` (en lugar del `\$` estándar, para evitar conflictos con el modo matemático de  $\text{\LaTeX}$ ). Los marcadores desconocidos y las llaves `{...}` de  $\text{\LaTeX}$  no se ven afectados. Para obtener una arroba literal en el texto, escribe `@@`.

`_fuente_precond(componente)` devuelve una representación **legible** de la precondition de un `Supuesto` o `Cuestion` para los mensajes de rechazo. Si el componente se cargó desde JSON (atributo `_json`) usa la cadena de código original (p. ej. `lambda ns: ns['a'] > ns['b']`); en otro caso recurre a `repr()` (que para una lambda definida en Python da algo como `<function <lambda> at 0x...>`). Así los avisos «=... rechazado por ...=» muestran la condición y no la dirección de memoria de la función.

---

```
Funciones auxiliares para setup paramétrico
```

```
class _NsTemplate(Template):
    delimiter = '@'

def _ns_eval(val, ns):
    return val(ns) if callable(val) else val

def _ns_interp(s, ns):
    return _NsTemplate(s).safe_substitute(ns) if ns else s

def _fuente_precond(componente):
    """Representación legible de la precondition de un componente.

    Si el componente se cargó desde JSON, devuelve la cadena de código
    original (p. ej. ``lambda ns: ns['a'] > ns['b']``); en otro caso usa
    ``repr()`` , que para una lambda definida directamente en Python da algo
    como ``<function <lambda> at 0x...>``.
    """
    j = getattr(componente, '_json', None)
    if j and j.get('precond') is not None:
        return j['precond']
    return repr(componente.p)
```

---

# Capítulo 1

## Clase Marcador

Usaremos la clase `Marcador` como una herramienta auxiliar para construir el árbol con todas las combinaciones posibles entre un conjunto de proposiciones y textos (que componen un conjunto de posibles enunciados), y un conjunto de cuestiones relacionadas con dichos enunciados.

*(Posteriormente se descartarán de dicho árbol las combinaciones enunciado-cuestión en las que la veracidad o falsedad de la cuestión no se pueda deducir con las proposiciones declaradas en el enunciado.)*

### 1.1. Funcionamiento General

Invocamos esta clase del siguiente modo: `Marcador(data)`, donde `data` es una lista de números enteros mayores que cero. Por ejemplo, si invocamos `Marcador([2, 3, 1])`, estamos pidiendo todas las listas `[i, j, k]` tales que  $0 \leq i < 2$ ,  $0 \leq j < 3$  y  $0 \leq k < 1$  (es decir, la primera componente solo puede tomar dos valores (0 y 1), la segunda tres (0, 1 y 2) y la tercera solo puede valer 0).

---

```
for i in Marcador([2,3,1]):  
    print(i)
```

---

```
[0, 0, 0]  
[0, 1, 0]  
[0, 2, 0]  
[1, 0, 0]  
[1, 1, 0]  
[1, 2, 0]
```

### 1.2. Implementación

La clase `Marcador` es un [iterador](#). Para el argumento `data=[n_1, ..., n_k]` genera todas las listas de la forma `[m_1, ..., m_k]` tales que cada `m_i` es un número natural menor que `n_i`.

Esta clase `Marcador` está formada por los siguientes fragmentos de código:

---

```
class Marcador:  
    <<Método __init__ para la clase Marcador>>  
    <<Método __iter__ para la clase Marcador>>  
    <<Método __next__ para la clase Marcador>>
```

---

**`__init__`** El método `__init__` es el inicializador de instancias en Python y actúa como el constructor de la clase; su función principal es configurar el estado inicial de un objeto recién creado asignándole valores a sus atributos mediante el argumento `self`. Este método especial se ejecuta de forma automática y obligatoria cada vez que se instancia una clase, permitiendo que cada objeto nazca con datos propios y personalizados, lo que evita tener que definir las variables de manera manual tras la creación del objeto. Para profundizar en los detalles técnicos de su funcionamiento, la herencia y las buenas prácticas, puede consultar la sección sobre clases en la [Documentación Oficial de Python](#).

Además, para construir un iterador necesitamos los métodos `__iter__` y `__next__`.

**\_\_iter\_\_** El método especial `__iter__` es el encargado de hacer que un objeto sea iterable en Python, permitiendo que pueda ser utilizado directamente en bucles `for`, comprensiones de listas o cualquier contexto que requiera recorrer elementos uno a uno. Su papel preciso es devolver un objeto iterador (que puede ser el propio objeto `self` si este implementa el método `__next__`, o un objeto iterador dedicado de otra clase), estableciendo así el protocolo de iteración de Python. Al invocar este método, Python sabe exactamente cómo comenzar el recorrido de la estructura de datos, ya sea una colección nativa como una lista o una clase personalizada que hayas diseñado. Para comprender a fondo cómo implementar este método junto con el protocolo de iteración y los generadores, puede consultar la sección sobre iteradores en la [Documentación Oficial de Python](#).

**\_\_next\_\_** El método especial `__next__` es la pieza fundamental que hace funcionar a un objeto **iterador** en Python, siendo el encargado de devolver el siguiente elemento disponible cada vez que se avanza en una secuencia. Este método es invocado automáticamente entre bastidores por la función nativa `next()` o al recorrer un bucle, y tiene la responsabilidad crucial de lanzar la excepción estándar `StopIteration` cuando se han agotado todos los elementos, lo que le indica a Python que la iteración ha terminado de forma segura. Junto con `__iter__`, conforma el protocolo de iteración completo de Python, permitiendo un control milimétrico sobre cómo se calculan y entregan los datos (incluso bajo demanda o de forma infinita). Para ver ejemplos prácticos de su implementación y entender cómo gestiona el flujo de datos, puede revisar la documentación sobre tipos iteradores en la [Referencia de la Biblioteca Estándar de Python](#).

### 1.2.1. Método `__init__(self, data)`

Este método inicializa la clase y almacena en el atributo `self.data` el argumento `data`; que es una lista con los topes de cada una de las componentes de las listas que se van a crear. En el ejemplo anterior, `[2,3,1]`.

---

Método `__init__` para la clase `Marcador`

---

```
def __init__(self, data):
    self.data = data
```

---

### 1.2.2. Método `__iter__(self)`

El método `__iter__` inicializa el estado del iterador. Este método genera la semilla del iterador. Dicha semilla es una lista de ceros (con tantos ceros como elementos hay en `self.data`). La lista `self.p` se entregará a `__next__` en la siguiente invocación al método.

---

Método `__iter__` para la clase `Marcador`

---

```
def __iter__(self):
    self.p = [0 for x in self.data]
    return self
```

---

### 1.2.3. Método `__next__`

Gestiona el avance del iterador.

- `__next__` devuelve `self.p` en su estado actual (`n`), y coloca en `self.p` la lista siguiente usando la función auxiliar `Siguiente`.
- Si `self.p` es vacía detiene la iteración.

---

Método `__next__` para la clase `Marcador`

---

```
def __next__(self):
    <<Definición de la función local Siguiente>>
    if self.p == []:
        raise StopIteration
    n = self.p
    self.p = Siguiente(self.p, self.data)
    return n
```

---

#### 1. Función auxiliar `Siguiente (x, y)`

Dentro del método `__next__` definimos de manera recursiva la función auxiliar local `Siguiente`, que es la función que calcula la lista siguiente.

La **función Siguiete(x, y)** es el corazón lógico de la clase. Utiliza **recursividad** para implementar un orden lexicográfico:

- **x**: lista de números naturales  $[m_1 \dots m_r]$  tales que  $m_i < n_i$  (es la lista de la que queremos calcular la siguiente).
- **y**: lista de números naturales  $[n_1 \dots n_r]$  mayores que cero (lista de topes).

La función **Siguiete** devuelve una lista  $[k_1 \dots k_r]$ ; que es la siguiente lista a  $[m_1 \dots m_r]$  (según el orden lexicográfico) que aún cumple que  $k_i < n_i$ . En caso de no existir tal lista devuelve  $[]$ .

a) Implementación:

- Intenta incrementar primero el último elemento de la lista (el de la derecha).
- Si el elemento de la derecha alcanza su límite ( $x[0]+1 == y[0]$ ), intenta incrementar el elemento anterior y resetea.<sup>a</sup> cero todos los elementos a su derecha.
- Si todos los elementos han alcanzado su límite, devuelve una lista vacía  $[]$ , lo que señala el fin de la iteración.

---

Definición de la función local Siguiete

```
def Siguiete(x,y):
    if x == [] :
        return []
    s = Siguiete(x[1:],y[1:])
    if s == []:
        if x[0]+1 == y[0]:
            return []
        else:
            return [x[0]+1] + [0 for i in x[1:]]
    else:
        return [x[0]] + s
```

---

## Capítulo 2

# Clases Supuesto y Cuestion

### 2.1. Clase Supuesto(enunciado, semantica, precondition)

La clase `Supuesto` define la estructura para almacenar tres aspectos (atributos) del planteamiento de un problema mediante su constructor `__init__`.

**enunciado** cadena de texto que presenta los supuestos en lenguaje humano.

**semantica** proposición que traduce parcialmente el significado lógico del **enunciado** (contiene lo necesario para deducir las respuestas).

**precond** es una proposición o bien cualquiera de los valores `True` o `False`. El objeto `Supuesto` será incluido en el problema solo si la evaluación de **precond** es `True`.

---

Definición de la clase `Supuesto`

```
class Supuesto:
    def __init__(self, enunciado, semantica, precondition=True):
        self.e = enunciado
        self.s = semantica
        self.p = precondition

    def __repr__(self):
        """Método de representación"""
        return f"Supuesto(enunciado={self.e!r}, semantica={self.s!r}, precondition={self.p!r})"
```

---

### 2.2. Clase Cuestion(enunciado, semantica, precondition)

De manera análoga, la clase `Cuestion` define la estructura para almacenar cuatro aspectos (atributos) del planteamiento de un problema mediante su constructor `__init__`.

**enunciado** string que presenta la pregunta en lenguaje humano.

**semantica** proposición  $P$  tal que `test(P, semantica_de_los_supuestos)` devuelve `True` si no es refutable a partir de las premisas, es decir, es imposible que  $P$  sea falso si las premisas son verdad.

**precond** es una proposición o bien cualquiera de los valores `True` o `False`. El objeto `Cuestion` será incluido en el problema solo si la evaluación de **precond** es `True`.

**exp** al exportar las preguntas al formato xml de Moodle, cabe la posibilidad de que Moodle ofrezca al alumno una explicación una vez ha contestado a las preguntas. En el atributo **exp** se especifica la cadena de texto con la explicación que debe mostrar Moodle.

---

Definición de la clase `Cuestion`

```
class Cuestion:
    def __init__(self, enunciado, semantica, precondition=True, exp=""):
        self.e = enunciado
        self.s = semantica
        self.p = precondition
        self.x = exp

    def __repr__(self):
        """Método de representación"""
```

---

```
return (f"Cuestion(enunciado={self.e!r}, semantica={self.s!r}, "  
        f"precond={self.p!r}, exp={self.x!r})")
```

---

## Capítulo 3

# Segmentación en partes y validación de homogeneidad

La lista de componentes de un `ProblemaTipo` codifica implícitamente la estructura de un ejercicio (posiblemente multiparte) mediante dos invariantes:

- **(I1) Homogeneidad:** cada sublista contiene un único tipo de elemento (solo cadenas, solo `Supuesto` o solo `Cuestion`). Los escalares sueltos se interpretan como una sublista de un único elemento.
- **(I2) Segmentación:** un slot es *material de enunciado* (cadena, sublista de cadenas o sublista de `Supuesto`) o *cuestiones* (sublista de `Cuestion`). Una **parte** del ejercicio es una tanda de material de enunciado seguida de una tanda de cuestiones. Cuando, tras una tanda de cuestiones, reaparece material de enunciado, comienza una **parte nueva**. Un ejercicio de una sola parte es el caso degenerado.

Estas dos funciones implementan dichas reglas sin alterar todavía el comportamiento de `ProblemaTipo` (se usarán en fases posteriores):

- `validar_componentes(componentes)` comprueba I1 y lanza `ValueError` indicando el slot infractor.
- `segmentar(componentes)` valida y devuelve la lista de partes; cada parte es una tupla (`slots_enunciado`, `slots_cuestiones`).

---

```
Segmentación en partes y validación de homogeneidad
def _normaliza_slot(x):
    """Envuelve los escalares en una sublista; deja las listas tal cual."""
    return x if isinstance(x, list) else [x]

def _tipo_componente(e):
    if isinstance(e, str): return 'str'
    if isinstance(e, Supuesto): return 'Supuesto'
    if isinstance(e, Cuestion): return 'Cuestion'
    raise ValueError(
        f"Componente de tipo no soportado: {type(e).__name__} ({e!r}). "
        "Cada componente debe ser str, Supuesto o Cuestion.")

def _clasifica_slot(slot):
    """'cuestiones' si la sublista (ya normalizada y homogénea) es de Cuestion;
    'enunciado' en otro caso (cadenas o Supuesto)."""
    return 'cuestiones' if isinstance(slot[0], Cuestion) else 'enunciado'

def _une_enunciado(acc, pieza):
    """Concatena dos fragmentos de enunciado (de slots distintos) separándolos
    con un espacio, salvo que ya haya separación -uno termina o el otro empieza
    por espacio- o alguno esté vacío. Evita textos pegados sin romper el
    contenido que ya incluye sus propios espacios."""
    if acc and pieza and not acc[-1].isspace() and not pieza[0].isspace():
        return acc + ' ' + pieza
    return acc + pieza

def validar_componentes(componentes):
    """Comprueba la invariante de homogeneidad (I1).

    Tras normalizar, cada slot debe contener un único tipo entre
    {str, Supuesto, Cuestion}. Lanza ValueError indicando el slot infractor.
```

```

"""
for i, x in enumerate(componentes):
    slot = _normaliza_slot(x)
    if not slot:
        raise ValueError(f"El slot {i} está vacío.")
    tipos = {_tipo_componente(e) for e in slot}
    if len(tipos) > 1:
        raise ValueError(
            f"El slot {i} mezcla tipos {sorted(tipos)}; cada sublista debe "
            "contener un único tipo (solo cadenas, solo Supuesto o solo Cuestion).")

def segmentar(componentes):
    """Divide la lista de componentes en partes según las invariantes I1 e I2.

    Devuelve una lista de tuplas (slots_enunciado, [slots_cuestiones]).
    Una transición cuestiones->enunciado abre una parte nueva. Un ejercicio
    de una sola parte produce una lista de longitud 1.
    """
    validar_componentes(componentes)
    L = [_normaliza_slot(x) for x in componentes]
    partes, enun, cues, vistas = [], [], [], False
    for slot in L:
        if _clasifica_slot(slot) == 'cuestiones':
            cues.append(slot)
            vistas = True
        else:
            if vistas:
                # I2: cierra parte y abre una nueva
                partes.append((enun, cues))
                enun, cues, vistas = [], [], False
            enun.append(slot)
    partes.append((enun, cues))
    return partes

def _plan_partes(L):
    """Índice de parte (0, 1, 2, ...) de cada slot ya normalizado, según la
    regla I2. La transición cuestiones->enunciado incrementa el índice."""
    plan, parte, vistas = [], 0, False
    for slot in L:
        es_cues = _clasifica_slot(slot) == 'cuestiones'
        if not es_cues and vistas:
            parte += 1
            vistas = False
        if es_cues:
            vistas = True
            plan.append(parte)
    return plan

```

### 3.1. Comprobación

Verificamos la segmentación (I2) y la validación de homogeneidad (I1):

```

S = lambda e: Supuesto(e, 'True')
C = lambda e: Cuestion(e, 'True')

# Una sola parte: texto + supuesto + un bloque de cuestiones.
assert len(segmentar(['Sea A.', S('A es par'), [C('q1'), C('q2')]])) == 1

# Dos partes: la transición cuestiones -> enunciado abre parte nueva.
assert len(segmentar(['Stem', [C('p1')], '(parte 2)', [C('p2')]])) == 2

# Tres partes con escalares y sublistas como slots distintos.
assert len(segmentar(['t0', [C('a')], 't1', S('h'), [C('b')], 't2', [C('d')]])) == 3

# Puede arrancar directamente con cuestiones (enunciado vacío en la 1ª parte).
assert segmentar([[C('a')], 'txt', [C('b')]])[0][0] == []

# Sublistas de cuestiones consecutivas pertenecen a la misma parte.
assert len(segmentar(['t', [C('a')], [C('b')], [C('c')]])[0][1]) == 3

# I1: una sublista que mezcla tipos se rechaza.
for bad in [[S('h'), C('q')], ['txt', C('q')], [[S('a'), 'c']], ['t', []], [42]]:
    try:
        validar_componentes(bad); raise AssertionError(f"no rechazó {bad!r}")
    except ValueError:
        pass

print("Fase 1 OK: segmentar + validar_componentes")

```

# Capítulo 4

## Clase ProblemaTipo

La clase `ProblemaTipo` permite, a partir de una lista de posibles enunciados y varias listas de posibles preguntas, generar muchas variantes de ejercicios de opción múltiple por simple combinatoria. Combina enunciados con una sublista de preguntas y, para cada pregunta calcula su veracidad o falsedad dados los supuestos del enunciado. Si no es posible decidir la veracidad o falsedad de una pregunta para un enunciado concreto, esa combinación enunciado-pregunta es descartada.

### 4.1. Implementación

La clase `ProblemaTipo` es un iterador que genera todas las versiones aceptables de un problema. Cada variante resulta de tomar una de las opciones de cada una de las listas en `supuestos_y_cuestiones`. Si la precondition de alguna de las opciones elegidas es `False` (o resulta refutable), la variante entera del problema es rechazada y se evalúa la siguiente combinación disponible.

- El atributo `self.e` contiene la lista `supuestos_y_cuestiones`. El argumento `supuestos_y_cuestiones` es una lista de listas de strings, objetos `Supuesto` o objetos `Cuestion`. Cada lista representa un conjunto de opciones excluyentes.

Por tanto, cada elemento de `supuestos_y_cuestiones` es a su vez una lista de opciones (excluyentes) correspondiente a cada componente (cada parte) del texto del problema.

Por comodidad, en el caso de que una de esas listas consista en un único objeto (una única opción), también se permite escribir directamente dicho objeto sin encerrarlo en una lista; es decir, la lista `supuestos_y_cuestiones` también admite directamente objetos de tipo string, `Supuesto` o `Cuestion`.

La combinación de distintas opciones crea el texto completo de una variante del problema.

---

Definición de la clase `ProblemaTipo`

```
class ProblemaTipo:
    def __init__(self, supuestos_y_cuestiones, setup=None):
        self.e = supuestos_y_cuestiones
        self.setup = setup

    <<Generador interno de variantes por partes>>
    <<Generador profe de variantes por partes>>
    <<Iteración pública de ProblemaTipo>>
```

---

Internamente, `ProblemaTipo` calcula las *partes* del ejercicio (regla I2, vía `_plan_partes`) y genera, por cada combinación válida del marcador, la estructura `[(enunciado, cuestiones), ...]` con una entrada por parte. Las hipótesis se acumulan a lo largo de *todas* las partes: una cuestión de la parte *k* ve los `Supuesto` de las partes anteriores.

Sobre ese generador interno se ofrecen tres fachadas:

- el protocolo de iteración estándar (`for ... in problema`), que mantiene el contrato histórico (`etiqueta, enunciado, cuestiones`) para los ejercicios de **una sola parte** (lo habitual en docencia);

- el método `por_partes()`, que devuelve (etiqueta, [(enunciado, cuestiones), ...]) y es el que debe usarse con ejercicios **multiparte**;
- el método `por_partes_profe()`, vista de **revisión del profesor**: mantiene el mismo formato por partes, pero muestra *todas* las cuestiones de cada sublista (no una por variante) y marca las que no superan su precondition como rechazadas, en vez de descartar la variante. No es la salida que ve el alumno ni la que se exporta.

Si un ejercicio multiparte se itera por la vía histórica de tres elementos, se lanza un `ValueError` que remite a `por_partes()`, para no devolver una salida silenciosamente degradada (enunciados concatenados y cuestiones de distintas partes mezcladas).

#### 4.1.1. Generador interno `_variantes(self)`

Normaliza la lista, calcula el plan de partes (`_plan_partes`) y recorre el marcador combinatorio. Para cada variante construye un enunciado y una lista de cuestiones **por parte**, compartiendo una única lista `hipotesis` que crece a medida que se aceptan **Supuesto** (de ahí la acumulación entre partes). Si algún **Supuesto** o **Cuestion** resulta refutado, la variante completa se descarta. El contador `c` numera solo las variantes efectivamente emitidas.

---

Generador interno de variantes por partes

---

```
def _variantes(self):
    L = [_normaliza_slot(x) for x in self.e]
    plan = _plan_partes(L)
    npar = (plan[-1] + 1) if plan else 1
    c = 0
    for variante in Marcador([len(x) for x in L]):
        ns = self.setup() if self.setup else {}
        hipotesis = []
        enunciados = ['' for _ in range(npar)]
        cuestiones = [[] for _ in range(npar)]
        descartada = False
        for n in range(len(L)):
            parte = plan[n]
            componente = L[n][variante[n]]
            <<Tratamiento del componente por partes>>
        if not descartada:
            c += 1
            yield (str(c), list(zip(enunciados, cuestiones)))
```

---

El fragmento `<<Tratamiento del componente por partes>>` procesa cada componente según su tipo, dirigiéndolo al enunciado o a las cuestiones de su parte (`parte`):

- Una cadena se concatena al enunciado de su parte.
- Un **Supuesto** testea su precondition contra las `hipotesis` acumuladas; si pasa, su texto se añade al enunciado de su parte y su semántica se incorpora a las `hipotesis` (visibles para las partes siguientes). Si se refuta, se descarta la variante.
- Una **Cuestion** testea su precondition; si pasa, el par (texto, veracidad) se añade a las cuestiones de su parte. Si se refuta, se descarta la variante.

---

Tratamiento del componente por partes

---

```
if isinstance(componente, str):
    enunciados[parte] = _une_enunciado(enunciados[parte], _ns_interp(componente, ns))

elif isinstance(componente, Supuesto):
    precond = _ns_eval(componente.p, ns)
    if test(precond, hipotesis):
        enunciados[parte] = _une_enunciado(
            enunciados[parte], _ns_interp(_ns_eval(componente.e, ns), ns))
        hipotesis = hipotesis + [_ns_eval(componente.s, ns)]
    else:
        print('\n Supuesto: ' + str(componente.e) \
            + ' rechazado por ' + _fuente_precond(componente) + '\n')
        descartada = True
        break

elif isinstance(componente, Cuestion):
    precond = _ns_eval(componente.p, ns)
    semantica = _ns_eval(componente.s, ns)
    texto = _ns_interp(_ns_eval(componente.e, ns), ns)
    if test(precond, hipotesis):
        cuestiones[parte].append(
            (texto, (True if test(semantica, hipotesis) else False), 1, componente.x))
    else:
        print('\n Cuestion: ' + str(componente.e) \
```

```

    + ' rechazada por ' + _fuente_precond(componente) + '\n')
descartada = True
break

```

---

#### 4.1.2. Generador profe \_variantes\_profe(self)

Variante de revisión de \_variantes. Difiere en dos puntos:

- Aplica CuestionesJuntas a la lista de componentes, que **aplana** las sublistas de cuestiones en cuestiones sueltas. Como cada cuestión pasa a ser su propio slot (de una sola opción), el marcador combinatorio ya no elige una cuestión por sublista: se muestran **todas**. Las variantes quedan gobernadas solo por las alternativas de textos y supuestos.
- Una Cuestion refutada por su precondition no descarta la variante: se incluye marcada como rechazada mediante la tupla (texto, rechazada por ..., 0) (3 elementos, frente a los 4 de una cuestión aceptada (texto, veracidad, 1, exp)). Un Supuesto refutado sí descarta la variante (e imprime el aviso), igual que en \_variantes.

Conserva el layout por partes (\_plan\_partes) y la unión de enunciados con \_une\_enunciado, de modo que la salida encaja con el mismo renderizado por partes que por\_partes().

---

Generador profe de variantes por partes

---

```

def _variantes_profe(self):
    L = [_normaliza_slot(x) for x in CuestionesJuntas(self.e)]
    plan = _plan_partes(L)
    npar = (plan[-1] + 1) if plan else 1
    c = 0
    for variante in Marcador([len(x) for x in L]):
        ns = self.setup() if self.setup else {}
        hipotesis = []
        enunciados = ['' for _ in range(npar)]
        cuestiones = [[] for _ in range(npar)]
        descartada = False
        for n in range(len(L)):
            parte = plan[n]
            componente = L[n][variante[n]]
            if isinstance(componente, str):
                enunciados[parte] = _une_enunciado(enunciados[parte], _ns_interp(componente, ns))
            elif isinstance(componente, Supuesto):
                precond = _ns_eval(componente.p, ns)
                if test(precond, hipotesis):
                    enunciados[parte] = _une_enunciado(
                        enunciados[parte], _ns_interp(_ns_eval(componente.e, ns), ns))
                    hipotesis = hipotesis + [_ns_eval(componente.s, ns)]
                else:
                    print('\n Supuesto: ' + str(componente.e) \
                          + ' rechazado por ' + _fuente_precond(componente) + '\n')
                    descartada = True
                    break
            elif isinstance(componente, Cuestion):
                precond = _ns_eval(componente.p, ns)
                texto = _ns_interp(_ns_eval(componente.e, ns), ns)
                if test(precond, hipotesis):
                    semantica = _ns_eval(componente.s, ns)
                    cuestiones[parte].append(
                        (texto, (True if test(semantica, hipotesis) else False), 1, componente.x))
                else:
                    cuestiones[parte].append(
                        (texto, 'rechazada por ' + _fuente_precond(componente), 0))
        if not descartada:
            c += 1
            yield (str(c), list(zip(enunciados, cuestiones)))

```

---

#### 4.1.3. Fachadas de iteración: por\_partes(), por\_partes\_profe(), \_\_iter\_\_ y \_\_next\_\_

---

Iteración pública de ProblemaTipo

---

```

def por_partes(self):
    """Itera las variantes válidas como (etiqueta, [(enunciado, cuestiones), ...]).

    Es la vía recomendada para ejercicios multiparte. En un ejercicio de una
    sola parte cada elemento es una lista de longitud 1.
    """
    return self._variantes()

def por_partes_profe(self):
    """Vista de revisión del profesor, en el mismo formato por partes que
    por_partes(): muestra TODAS las cuestiones de cada sublista (no una por

```

```

    variante) y marca las que no superan su precondición como rechazadas en vez
    de descartar la variante. No es la salida que ve el alumno ni la exportada."""
    return self._variantes_profe()

def __iter__(self):
    self._gen = self._variantes()
    return self

def __next__(self):
    etiqueta, partes = next(self._gen) # propaga StopIteration al agotarse
    if len(partes) > 1:
        raise ValueError(
            "Este ProblemaTipo tiene varias partes; itéralo con .por_partes(), "
            "que devuelve (etiqueta, [(enunciado, cuestiones), ...])."
        )
    (enunciado, cuestiones), = partes
    return (etiqueta, enunciado, cuestiones)

```

## 4.2. Ejemplo de funcionamiento

Escribamos un hipotético ejercicio (es decir una lista que contiene cadenas de caracteres, supuestos, cuestiones, sublistas de supuestos o sublistas de cuestiones):

---

Ejemplo de programación de un ejercicio

---

```

ejercicio = [ "Considerere ",
    [
        Supuesto("A ", v("A")),
        Supuesto("B ", v("B")),
    ],
    [
        Supuesto("y que A implica C. ", v("A") >> v("C")),
        Supuesto("y que B equivale a D. ", v("B") ** v("D")),
    ],
    "Conteste: ",
    [
        Cuestion("¿Se da C?", v("C"), exp="Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A"),
        Cuestion("¿Se da D?", v("D")),
        Cuestion("¿Se da E?", v("E"), v("D")),
    ],
    [
        Cuestion("¿Se dan A y C?", v("A") & v("C")),
        Cuestion("¿Se dan C y D?", v("C") & v("D")),
    ],
]

```

---

A continuación veamos lo que hace `ProblemaTipo` con la lista `ejercicio`. Observe que en cada iteración se toma un elemento de cada lista de supuestos y un elemento de cada lista de cuestiones para formar la combinación de un enunciado (compuesto por supuestos) y un subconjunto de preguntas (tantas como sublistas de cuestiones). Si alguna cuestión es descartada en una combinación, se salta a la siguiente combinación (indicando la cuestión descartada y el motivo). Así, `ProblemaTipo` muestra todas las combinaciones posibles de cuestiones que se pueden responder para cada combinación de supuestos. Fíjese que los elementos que no son listas (aquí las dos cadenas de caracteres) funcionan como sublistas con un único elemento (por eso en todas las iteraciones aparece tanto 'Considerere' como 'Conteste:').

---

Ejemplo de ProblemaTipo

---

```

for i in ProblemaTipo( ejercicio ):
    print(i)

```

---

```

('1', 'Considerere A y que A implica C. Conteste: ', [(('¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y
('2', 'Considerere A y que A implica C. Conteste: ', [(('¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y
('3', 'Considerere A y que A implica C. Conteste: ', [(('¿Se da D?', False, 1, ''), ('¿Se dan A y C?', True, 1, ''))])
('4', 'Considerere A y que A implica C. Conteste: ', [(('¿Se da D?', False, 1, ''), ('¿Se dan C y D?', False, 1, ''))])

Question: ¿Se da E? rechazada por v('D')

Question: ¿Se da E? rechazada por v('D')

('5', 'Considerere A y que B equivale a D. Conteste: ', [(('¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica
('6', 'Considerere A y que B equivale a D. Conteste: ', [(('¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica
('7', 'Considerere A y que B equivale a D. Conteste: ', [(('¿Se da D?', False, 1, ''), ('¿Se dan A y C?', False, 1, ''))])
('8', 'Considerere A y que B equivale a D. Conteste: ', [(('¿Se da D?', False, 1, ''), ('¿Se dan C y D?', False, 1, ''))])

Question: ¿Se da E? rechazada por v('D')

```

Question: ¿Se da E? rechazada por v('D')

```
('9', 'Considere B y que A implica C. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')])
('10', 'Considere B y que A implica C. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')])
('11', 'Considere B y que A implica C. Conteste: ', [( '¿Se da D?', False, 1, ''), ( '¿Se dan A y C?', False, 1, '')])
('12', 'Considere B y que A implica C. Conteste: ', [( '¿Se da D?', False, 1, ''), ( '¿Se dan C y D?', False, 1, '')])
```

Question: ¿Se da E? rechazada por v('D')

Question: ¿Se da E? rechazada por v('D')

```
('13', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')])
('14', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')])
('15', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da D?', True, 1, ''), ( '¿Se dan A y C?', False, 1, '')])
('16', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da D?', True, 1, ''), ( '¿Se dan C y D?', False, 1, '')])
('17', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da E?', False, 1, ''), ( '¿Se dan A y C?', False, 1, '')])
('18', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da E?', False, 1, ''), ( '¿Se dan C y D?', False, 1, '')])
```

Cuando la lista de cuestiones es extensa (o incluye aclaraciones), la representación en una tupla puede extenderse más allá de los márgenes de este documento al utilizar `print`. Para mejorar la claridad y facilitar la comprensión del funcionamiento de `ProblemaTipo`, desempaquetaremos los componentes devueltos por el iterador y los presentaremos de forma estructurada. Esto permitirá visualizar mejor los resultados.

En cada iteración, presentaremos el enunciado completo correspondiente, junto con las cuestiones incluidas, cada una en una línea separada (indicando su veracidad y la explicación incluida en el atributo `self.x` si no la hemos dejado vacía). Si la variante fue abortada, se indicará cuál fue la cuestión desencadenante y el motivo de la interrupción.

---

```
for i in ProblemaTipo( ejercicio ):
    # Desempaquetamos la tupla en variables
    id_pregunta, EnunciadoCompleto, lista_Cuestiones = i

    # Imprimimos los primeros datos
    print(f"ID: {id_pregunta}")
    print(f"Enunciado completo: {EnunciadoCompleto}")

    # Imprimimos la lista de cuestiones, cada cuestión en una línea
    print(*lista_Cuestiones, sep='\n')
    print('\n')
```

---

ID: 1

Enunciado completo: Considere A y que A implica C. Conteste:  
( '¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan A y C?', True, 1, '' )

ID: 2

Enunciado completo: Considere A y que A implica C. Conteste:  
( '¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan C y D?', False, 1, '' )

ID: 3

Enunciado completo: Considere A y que A implica C. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan A y C?', True, 1, '' )

ID: 4

Enunciado completo: Considere A y que A implica C. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan C y D?', False, 1, '' )

Question: ¿Se da E? rechazada por v('D')

Question: ¿Se da E? rechazada por v('D')

ID: 5

Enunciado completo: Considere A y que B equivale a D. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan A y C?', False, 1, '' )

ID: 6

Enunciado completo: Considere A y que B equivale a D. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan C y D?', False, 1, '' )

ID: 7

Enunciado completo: Considere A y que B equivale a D. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan A y C?', False, 1, '' )

ID: 8

Enunciado completo: Considere A y que B equivale a D. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan C y D?', False, 1, '' )

Question: ¿Se da E? rechazada por v('D')

Question: ¿Se da E? rechazada por v('D')

ID: 9

Enunciado completo: Considere B y que A implica C. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan A y C?', False, 1, '' )

ID: 10

Enunciado completo: Considere B y que A implica C. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan C y D?', False, 1, '' )

ID: 11

Enunciado completo: Considere B y que A implica C. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan A y C?', False, 1, '' )

ID: 12

Enunciado completo: Considere B y que A implica C. Conteste:  
( '¿Se da D?', False, 1, '' )  
( '¿Se dan C y D?', False, 1, '' )

Question: ¿Se da E? rechazada por v('D')

Question: ¿Se da E? rechazada por v('D')

ID: 13

Enunciado completo: Considere B y que B equivale a D. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan A y C?', False, 1, '' )

ID: 14

Enunciado completo: Considere B y que B equivale a D. Conteste:  
( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')  
( '¿Se dan C y D?', False, 1, '' )

ID: 15

Enunciado completo: Considere B y que B equivale a D. Conteste:

```
('¿Se da D?', True, 1, '')
('¿Se dan A y C?', False, 1, '')
```

ID: 16

Enunciado completo: Considere B y que B equivale a D. Conteste:

```
('¿Se da D?', True, 1, '')
('¿Se dan C y D?', False, 1, '')
```

ID: 17

Enunciado completo: Considere B y que B equivale a D. Conteste:

```
('¿Se da E?', False, 1, '')
('¿Se dan A y C?', False, 1, '')
```

ID: 18

Enunciado completo: Considere B y que B equivale a D. Conteste:

```
('¿Se da E?', False, 1, '')
('¿Se dan C y D?', False, 1, '')
```

En el ejemplo anterior, se observa que, para cada combinación de supuestos, en cada iteración solo se presenta una cuestión de cada sublista de cuestiones. Dado que hay solo dos sublistas, esto resulta en la visualización de dos preguntas por iteración, esto dificulta la revisión y mantenimiento del banco de preguntas.

En el anterior ejemplo, y aunque existen únicamente cuatro enunciados posibles y cinco preguntas en total, la combinatoria ha generado 18 variantes no descartables, cada una con un subconjunto de solo dos preguntas. Esta complejidad complica la tarea de revisar y mantener un banco de ejercicios eficaz.

### 4.3. Ejercicios multiparte: `por_partes()`

Un ejercicio es *multiparte* cuando, tras una tanda de cuestiones, su lista de componentes vuelve a introducir material de enunciado (regla I2). En ese caso la vía de iteración histórica (`etiqueta`, `enunciado`, `cuestiones`) no basta —no podría separar las partes— y debe usarse `por_partes()`, que devuelve (`etiqueta`, `[(enunciado, cuestiones), ...]`) con una entrada por parte. Las hipótesis se acumulan entre partes: una cuestión de una parte posterior «ve» los `Supuesto` de las anteriores.

Iterar un ejercicio multiparte por la vía histórica lanza un `ValueError` que remite a `por_partes()`.

---

```
multi = [
    "Sea A cierto. ", Supuesto("", v("A")),
    [ Question("¿A?", v("A")) ],           # parte 0
    "Sea además A→B. ", Supuesto("", v("A") >> v("B")),
    [ Question("¿B?", v("B")) ],         # parte 1 (B es cierto gracias a A acumulada)
]

etiqueta, partes = next(iter(ProblemaTipo(multi).por_partes()))
assert len(partes) == 2
(e0, c0), (e1, c1) = partes
assert c0 == [('¿A?', True, 1, '')]
assert c1 == [('¿B?', True, 1, '')] # la hipótesis A de la parte 0 se acumula en la parte 1

# Por la vía histórica de 3 elementos, un multiparte avisa:
try:
    list(ProblemaTipo(multi))
except ValueError as ex:
    assert "por_partes()" in str(ex)

print("Fase 2 OK: por_partes() separa partes y acumula hipótesis entre ellas")
```

---

# Capítulo 5

## Clase ProblemaTipoProfe

Para facilitar la revisión y el mantenimiento de cada ejercicio, es preferible ver el conjunto completo de cuestiones para cada uno de los enunciados. La clase `ProblemaTipoProfe` lo hace, indicando en cada caso si son `True` o `False` (o si serán descartadas). Para ello utiliza el procedimiento auxiliar `CuestionesJuntas`, que crea una sublista con cada `Cuestion`. De este modo, como al iterar se toma un elemento de cada una de las sublistas, se mostrarán todas las cuestiones a la vez.

### 5.1. Implementación

Hay dos diferencias entre el código de la clase `ProblemaTipoProfe` y el la clase `ProblemaTipo`.

- La primera es que en `self.e` cada `Cuestion` aparece como único elemento de una sublista, para así lograr que se ven todas las cuestiones en cada iteración.
- La segunda diferencia se produce en la definición del método `__next__`, que aquí no oculta las cuestiones que sí son rechazadas cuando usamos `ProblemaTipo`.

`ProblemaTipoProfe` conserva el esquema clásico de iteración (`self.l`, `self.long`, `self.i`, `self.c`) y su propio `__iter__`, ya que la vista del profesor es de una sola parte y no comparte el motor por partes de `ProblemaTipo`.

```
----- Método __iter__ para la clase ProblemaTipoProfe -----
def __iter__(self):
    self.l = [x if isinstance(x,list) else [x] for x in self.e]
    self.long = len(self.l)
    self.i = iter(Marcador([len(x) for x in self.l]))
    self.c = 0
    return self
-----

----- Definición de la clase ProblemaTipoProfe -----
class ProblemaTipoProfe:
    def __init__(self, supuestos_y_cuestiones, setup=None):
        self.e = CuestionesJuntas(supuestos_y_cuestiones)
        self.setup = setup

    <<Método __iter__ para la clase ProblemaTipoProfe>>
    <<Método __next__ para la clase ProblemaTipoProfe>>
-----
```

#### 5.1.1. Método `__next__` para la clase `ProblemaTipoProfe`

La diferencia con respecto al <<Método `__next__` para la clase `ProblemaTipo`>> es el tratamiento de las componentes; no se excluyen las preguntas que serán rechazadas en la versión para el estudiante, se añade la indicación `'rechazada por'` y la precondition que ocasionará el rechazo con la clase `ProblemaTipo`.

```
----- Método __next__ para la clase ProblemaTipoProfe -----
def __next__(self):
    self.c += 1
    while True:
        try:
            variante = next(self.i)
        except StopIteration:
-----
```

```

        raise StopIteration

    ns = self.setup() if self.setup else {}
    enunciado = ""
    hipotesis = []
    cuestiones = []

    for n in range(self.long+1):
        if n == self.long:
            return (str(self.c), enunciado, cuestiones)

    componente = self.l[n][variante[n]]
    <<Tratamiento en función del tipo de componente para la versión del Profe>>

```

---

## Tratamiento en función del tipo de componente en la versión del Profe

Tratamiento en función del tipo de componente para la versión del Profe

```

if isinstance(componente, str):
    enunciado = _une_enunciado(enunciado, _ns_interp(componente, ns))

elif isinstance(componente, Supuesto):
    precond = _ns_eval(componente.p, ns)
    if test(precond, hipotesis):
        enunciado = _une_enunciado(
            enunciado, _ns_interp(_ns_eval(componente.e, ns), ns))
        hipotesis = hipotesis + [_ns_eval(componente.s, ns)]
    else:
        print('\n Supuesto: ' + str(componente.e) \
            + ' rechazado por ' + _fuente_precond(componente) + '\n')
        break

elif isinstance(componente, Cuestion):
    precond = _ns_eval(componente.p, ns)
    semantica = _ns_eval(componente.s, ns)
    texto = _ns_interp(_ns_eval(componente.e, ns), ns)
    if test(precond, hipotesis):
        cuestiones = cuestiones + \
            [(texto, (True if test(semantica, hipotesis) else False), 1, componente.x)]
    else:
        cuestiones = cuestiones + \
            [(texto, 'rechazada por ' + _fuente_precond(componente), 0)]

```

---

### 5.1.2. Función auxiliar CuestionesJuntas

Este bloque de código define la función `CuestionesJuntas(lista)`, cuyo objetivo principal es **aislar cada objeto de la clase `Cuestion` dentro de su propia sublista individual**, manteniendo otros tipos de datos (como cadenas o listas de supuestos) intactos.

En detalle:

1. **CreaLista(t)**: Una función auxiliar que asegura que cualquier elemento sea tratado como una lista (si no lo es, lo envuelve en una para poder indexarlo).
2. **Iteración**: Recorre cada elemento `e` de la lista de entrada `lista`.
3. **Strings y No-Cuestiones**: Si el elemento es una cadena o si el primer elemento de su contenido (tras pasarlo por `CreaLista`) no es de tipo `Cuestion` (por ejemplo, un objeto de la clase `Supuesto`), se añade directamente a la lista resultante `p` mediante `append`.
4. **Aislamiento de Cuestion**: Si el elemento es una instancia de `Cuestion` o pertenece a una lista de cuestiones, se utiliza `extend(CreaLista(e))`, lo que descompone cualquier agrupación previa de `cuestiones`, añadiendo cada `Cuestion` como una sublista independiente (y con un único elemento) dentro de `p`.

**En resumen:** Transforma una lista mixta de modo que cada objeto `Cuestion` acabe confinado en una sublista propia. De este modo, el generador multidimensional las procesa de forma independiente y simultánea.

**Objetivo:** Esto permite que la clase `ProblemaTipoProfe` muestre todas las cuestiones posibles para cada enunciado en lugar de una sola.

Metodo auxiliar para juntar cuestiones en formato para profe

```

def CuestionesJuntas(lista):
    def CreaLista(t):
        return t if isinstance(t, list) else [t]

```

```

p = []
for e in lista:
    if isinstance(e, str):
        p.append(e)
    elif not isinstance(CreaLista(e)[0], Cuestion):
        p.append(e)
    else:
        p.extend(CreaLista(e))
return p

```

---

## 5.2. Ejemplo de funcionamiento

Usaremos la misma lista ejercicio que usamos en el capítulo anterior

---

```

for i in ProblemaTipoProfe( ejercicio ):
    print(i)

```

---

```

('1', 'Considere A y que A implica C. Conteste: ', [( '¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y
('2', 'Considere A y que B equivale a D. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica
('3', 'Considere B y que A implica C. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y
('4', 'Considere B y que B equivale a D. Conteste: ', [( '¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica

```

Como la vez anterior la representación en una tupla usando `print` se ha extendido más allá de los márgenes de este documento. Para evitar este problema y que el lector vea mejor el funcionamiento de `ProblemaTipoProfe`, vamos usar el mismo truco que en el capítulo anterior: Debajo de cada variante del enunciado completo, mostraremos todas las cuestiones de manera individual, cada una en una línea separada, junto con el cálculo de su veracidad o la indicación de si serán descartadas al emplear la clase `ProblemaTipo`. Si la cuestión incluye el atributo `self.x` con una explicación, también se muestra la explicación (si no, aparece una cadena vacía `' '`).

---

```

for i in ProblemaTipoProfe( ejercicio ):
    # Desempaquetamos la tupla en variables
    id_pregunta, EnunciadoCompleto, lista_Cuestiones = i

    # Imprimimos los primeros datos
    print(f"ID: {id_pregunta}")
    print(f"Enunciado completo: {EnunciadoCompleto}")

    # Imprimimos la lista de cuestiones, cada cuestión en una línea
    print(*lista_Cuestiones, sep='\n')
    print('\n')

```

---

```

ID: 1
Enunciado completo: Considere A y que A implica C. Conteste:
(¿Se da C?', True, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')
(¿Se da D?', False, 1, ' ')
(¿Se da E?', "rechazada por v('D')", 0)
(¿Se dan A y C?', True, 1, ' ')
(¿Se dan C y D?', False, 1, ' ')

```

```

ID: 2
Enunciado completo: Considere A y que B equivale a D. Conteste:
(¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')
(¿Se da D?', False, 1, ' ')
(¿Se da E?', "rechazada por v('D')", 0)
(¿Se dan A y C?', False, 1, ' ')
(¿Se dan C y D?', False, 1, ' ')

```

```

ID: 3
Enunciado completo: Considere B y que A implica C. Conteste:
(¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')
(¿Se da D?', False, 1, ' ')
(¿Se da E?', "rechazada por v('D')", 0)
(¿Se dan A y C?', False, 1, ' ')
(¿Se dan C y D?', False, 1, ' ')

```

```

ID: 4
Enunciado completo: Considere B y que B equivale a D. Conteste:
(¿Se da C?', False, 1, 'Solo sabemos que C es cierto y si A implica C y además sabemos que ocurre A')

```

```
('¿Se da D?', True, 1, '')  
( '¿Se da E?', False, 1, '' )  
( '¿Se dan A y C?', False, 1, '' )  
( '¿Se dan C y D?', False, 1, '' )
```

Vemos que son cuatro enunciados en total, y para cada uno de ellos vemos lo que pasa con cada una de las cinco cuestiones posibles: si son verdaderas, si son falsas o si serán descartadas (y el motivo). Ahora es mucho más fácil mejorar y depurar el **ejercicio** que estamos diseñando para nuestros exámenes de opción múltiple.

# Capítulo 6

## Clase ProblemaVF

A diferencia de las clases anteriores, que exploran de forma sistemática y secuencial todas las combinaciones posibles de un espacio combinatorio, la clase `ProblemaVF` está diseñada para generar ejercicios basados en un **banco de cuestiones aleatorizado**.

Su objetivo primordial es construir variantes de exámenes del tipo **Verdadero/Falso**, donde el enunciado permanece fijo y cada variante ofrece un subconjunto único y desordenado de preguntas seleccionadas al azar a partir de un repertorio común.

### 6.1. Implementación

El núcleo de esta clase radica en el uso de la función `sample` del módulo `random` de Python, lo que permite extraer muestras sin repetición del banco de preguntas en cada iteración.

Una particularidad fundamental de este diseño es que el método `__next__` es que no sigue un barrido secuencial de todas las combinaciones posibles. Aquí el iterador generará una variante al azar cada vez que se invoque el método `next()` (obviamente, dado que el banco de cuestiones es finito, aparecerán variantes repetidas de cuando en cuando).

---

Definición de la clase `ProblemaVF`

```
from random import sample
class ProblemaVF():
    def __init__(self, enunciado, cuestiones, NumPreguntas):
        self.e = enunciado
        self.c = cuestiones
        self.NumPreguntas = NumPreguntas

    def __iter__(self):
        self.contador = 0
        return self

    def __next__(self):
        cuestiones = sample(self.c, self.NumPreguntas)
        self.contador += 1
        return (str(self.contador), self.e, cuestiones)
```

---

#### 6.1.1. Métodos de la clase `ProblemaVF`

- `__init__(self, enunciado, cuestiones, NumPreguntas)`: Inicializa el generador almacenando el texto base del problema (`enunciado`), la lista completa de tuplas con las preguntas y sus valores de verdad (`cuestiones`), y la cantidad exacta de ítems que se deben extraer para cada ejercicio (`NumPreguntas`).
- `__iter__(self)`: Prepara el objeto para ser iterado, inicializando un `self.contador` interno a cero que servirá para identificar unívocamente el ID de cada variante generada.
- `__next__(self)`: Constituye el motor de la clase. En cada invocación ejecuta tres pasos:
  1. Utiliza `sample(self.c, self.NumPreguntas)` para obtener una lista con el número de preguntas solicitado, garantizando que en una misma variante no se repita ninguna cuestión.

2. Incrementa el contador de variantes.
3. Devuelve una tupla con el identificador en formato de cadena, el enunciado estático y la lista aleatoria de preguntas seleccionadas con sus respectivos estados lógicos (`True/False`).

**Nota de diseño:** Dado que no existe una condición de parada intrínseca (como un extremo de lista), el bucle que consuma este iterador debe limitarse explícitamente desde el exterior (por ejemplo, mediante un bucle `for` con un rango acotado o controlando el número de peticiones).

## 6.2. Ejemplo de funcionamiento

A continuación se ilustra el comportamiento de la clase utilizando un banco de diez enunciados cotidianos relacionados con la vida académica.

---

```
banco = [
    ("Todo alumno que va a clase aprueba", False),
    ("Todo alumno que suspende va a clase", False),
    ("Todo alumno que sabe aprueba", True ),
    ("Todo alumno que NO sabe suspende", False ),
    ("Algunos alumnos aprueban copiando", True ),
    ("Copiar es intolerable", True ),
    ("Si aprueban todos es estupendo", True ),
    ("Si aprueban todos eres buen profesor", False),
    ("Si suspenden todos eres mal profesor", False),
    ("Si suspenden todos es frustrante", True),]
```

---

### 6.2.1. Generación de cuatro variantes cortas

En este primer ensayo se configuran cuatro ejercicios, solicitando únicamente dos preguntas tipo verdadero/falso para cada uno.

---

```
Ejemplo de ProblemaVF
enunciado = "Marque las verdaderas:"

p = ProblemaVF (enunciado, banco, 2)      # Dos preguntas del banco por variante
p0 = iter(p)

for i in range(4):                       # Generamos cuatro variantes y paramos
    print(next(p0))

('1', 'Marque las verdaderas:', [('Si suspenden todos es frustrante', True), ('Si aprueban todos es estupendo', True)])
('2', 'Marque las verdaderas:', [('Copiar es intolerable', True), ('Todo alumno que suspende va a clase', False)])
('3', 'Marque las verdaderas:', [('Todo alumno que suspende va a clase', False), ('Si suspenden todos eres mal profesor', Fa
('4', 'Marque las verdaderas:', [('Todo alumno que suspende va a clase', False), ('Todo alumno que sabe aprueba', True)])
```

---

### 6.2.2. Generación de cinco variantes con presentación formateada

Como la representación en una sola tupla puede dificultar la lectura a medida que el número de ítems crece, vamos a desempaquetar los componentes devueltos por el iterador para volcar los datos de manera estructurada para que se vea mejor el resultado, pues vamos a mostrar cada cuestión en una línea independiente.

Dado que ahora lo vamos a ver mejor, aumentamos la complejidad solicitando tres preguntas por variante para un total de cinco ejercicios:

---

```
p = ProblemaVF (enunciado, banco, 3)      # Tres preguntas por variante
p0 = iter(p)

for i in range(5):
    # Desempaquetamos la tupla en variables
    id_pregunta, EnunciadoCompleto, lista_Cuestiones = next(p0)

    # Imprimimos los primeros datos
    f"ID: {id_pregunta}"
    print(f"Enunciado completo: {EnunciadoCompleto}")

    # Imprimimos la lista de cuestiones, cada cuestión en una línea
    print(*lista_Cuestiones, sep='\n')
    print('\n')
```

---

```
Enunciado completo: Marque las verdaderas:
('Copiar es intolerable', True)
```

('Si suspenden todos eres mal profesor', False)  
( 'Si suspenden todos es frustrante', True)

Enunciado completo: Marque las verdaderas:  
( 'Todo alumno que suspende va a clase', False)  
( 'Si aprueban todos es estupendo', True)  
( 'Si aprueban todos eres buen profesor', False)

Enunciado completo: Marque las verdaderas:  
( 'Si suspenden todos eres mal profesor', False)  
( 'Copiar es intolerable', True)  
( 'Si aprueban todos es estupendo', True)

Enunciado completo: Marque las verdaderas:  
( 'Si suspenden todos es frustrante', True)  
( 'Todo alumno que suspende va a clase', False)  
( 'Todo alumno que va a clase aprueba', False)

Enunciado completo: Marque las verdaderas:  
( 'Todo alumno que NO sabe suspende', False)  
( 'Todo alumno que suspende va a clase', False)  
( 'Todo alumno que sabe aprueba', True)

Como se puede apreciar en el resultado impreso, cada uno de los cinco bloques representa un ejercicio completamente autónomo con un identificador secuencial, conservando el enunciado general e integrando un muestreo dinámico y heterogéneo extraído directamente de la lista de afirmaciones almacenada con el nombre de **banco**.

## Capítulo 7

# Clases SubPregunta y ProblemaMultiParte (deprecadas)

Los formatos AMC y Moodle permiten agrupar varias sub-preguntas de opción múltiple bajo un mismo enunciado general (pregunta *multi-parte* en AMC, *cloze* en Moodle). En qbank esto ya **no es un tipo aparte**: una pregunta multiparte es un `ProblemaTipo` cuya lista plana de componentes intercala el enunciado común y, por cada parte, material de enunciado seguido de una sublista de `Cuestion`. Las partes se infieren con la regla I2 (`segmentar`) y se recorren con `por_partes()`.

`SubPregunta` y `ProblemaMultiParte` se conservan **solo por compatibilidad** y están **deprecadas**:

- `SubPregunta(intro, cuestiones)` es un contenedor simple (un `intro` y una lista de `Cuestion`).
- `ProblemaMultiParte(componentes, subpreguntas, setup)` no construye un objeto propio: **devuelve un `ProblemaTipo` equivalente** (aplanando `componentes` + cada `SubPregunta`) y emite un `DeprecationWarning`. El resultado se itera con `por_partes()` como cualquier `ProblemaTipo` multiparte.

---

Definición de `SubPregunta` y `ProblemaMultiParte`

---

```
class SubPregunta:
    """DEPRECADO. Contenedor (intro, lista de Cuestion) del antiguo
    ProblemaMultiParte. Se conserva solo para compatibilidad; usa ProblemaTipo
    con su lista plana de componentes."""
    def __init__(self, intro, cuestiones):
        self.intro = intro
        self.cuestiones = cuestiones if isinstance(cuestiones, list) else [cuestiones]

class ProblemaMultiParte:
    """DEPRECADO. Devuelve un ProblemaTipo equivalente.

    La estructura multiparte ya no es un tipo propio: se representa como un
    ProblemaTipo cuya lista plana intercala el enunciado común y, por cada
    SubPregunta, su intro seguido de su sublista de Cuestion. Las partes se
    infieren con la regla I2 y se recorren con por_partes().
    """
    def __new__(cls, componentes, subpreguntas, setup=None):
        import warnings
        warnings.warn(
            "ProblemaMultiParte y SubPregunta están deprecados: usa ProblemaTipo "
            "con su lista plana de componentes (las partes se infieren) y recórrelo "
            "con .por_partes().",
            DeprecationWarning, stacklevel=2)
        flat = list(componentes)
        for sp in subpreguntas:
            flat.append(sp.intro)
            flat.append(list(sp.cuestiones))
        return ProblemaTipo(flat, setup=setup)
```

---

### 7.1. Ejemplo

---

```
from qbank import *
import itertools
```

```
p = load_problema('ejemplosManual/superheroes.json')

# ProblemaTipo multiparte: se recorre con por_partes().
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, _, _ in cuestiones:
            print(f'    {" " if correcto else " "} {texto}')
    print()
```

---

## Parte II

# Serialización y persistencia en formato JSON

El módulo `qbank._json` permite guardar y cargar objetos `ProblemaTipo` y `ProblemaVF` en ficheros JSON. Esto facilita compartir bancos de preguntas, o modificarlos en distintas mediante un editor visual sin tener que reescribir el código Python en cada ocasión.

# Estructura del módulo qbank.\_json

El módulo completo está compuesto por varios fragmentos de código que se describen en las subsecciones siguientes.

---

```
qbank/_json.py
import json as _json
import calcprop as _calcprop_mod
from qbank._quiz import Supuesto, Cuestion, ProblemaTipo, ProblemaVF

__all__ = [
    'problema_from_dict',
    'problema_to_dict',
    'load_problema',
    'save_problema',
    'load_banco',
    'save_banco',
    'problema_to_python',
    'save_problema_py',
]

<<Espacio de nombres calcprop para JSON>>
<<Evaluación de expresiones JSON>>
<<Función setup desde string>>
<<Componente desde dict JSON>>
<<Slot desde JSON>>
<<Slot a JSON>>
<<Problema desde dict JSON>>
<<Problema a dict JSON>>
<<Cargar y guardar problema JSON>>
<<Cargar y guardar banco JSON>>
<<Exportación a código Python>>
```

---

# Capítulo 8

## Formato JSON

### 8.1. Para un ProblemaTipo

Un ProblemaTipo se representa con la siguiente estructura:

```
{
  "version": "1",
  "tipo": "ProblemaTipo",
  "nombre": "identificador_opcional",
  "setup": null,
  "componentes": [
    "texto fijo del enunciado",
    {"tipo": "Supuesto", "enunciado": "A ", "semantica": "v('A')", "precond": "True"},
    [
      {"tipo": "Supuesto", "enunciado": "alt1 ", "semantica": "v('A')"},
      {"tipo": "Supuesto", "enunciado": "alt2 ", "semantica": "v('B')"}
    ],
    [
      {"tipo": "Cuestion", "enunciado": "¿A?", "semantica": "v('A')",
       "precond": "True", "exp": ""}
    ]
  ]
}
```

El campo `componentes` es una lista cuyos elementos pueden ser:

- **Cadena de texto:** texto fijo que aparece en todas las variantes.
- **Dict:** un único `Supuesto` o `Cuestion` (equivale a una lista de un elemento).
- **Lista de dicts:** varias alternativas de las que `ProblemaTipo` elige una por variante.

Los campos `semantica` y `precond` se almacenan como **cadena de texto**. Al deserializar<sup>1</sup> se evalúan con `eval()` en el espacio de nombres (*namespace*)<sup>2</sup> de `calccprop`, lo que permite expresiones como `"v('A') & v('B')"`, los booleanos literales `True` y `False`, o lambdas para semánticas paramétricas (`"lambda ns: ns['x'] > 0"`).

<sup>1</sup>«Serializar» significa convertir un objeto Python en memoria (un `ProblemaTipo`, por ejemplo) a un formato que se puede guardar o transmitir —en este caso, JSON. «Deserializar» es la operación inversa: leer ese JSON y reconstruir el objeto Python.

<sup>2</sup>«Espacio de nombres» es la traducción literal de *namespace*, que en Python significa simplemente el conjunto de variables (nombres y valores) disponibles en un momento dado. Aquí los usaremos en dos contextos distintos: Por una parte en `_EVAL_NS`; es decir, el diccionario que contiene todas las funciones y variables del módulo `calccprop` (`v`, `test`, etc.). Cuando se evalúa la cadena `"v('A') & v('B')"` con `eval()`, se le pasa ese diccionario para que `v` esté disponible. Aquí «espacio de nombres» podría sustituirse por «entorno de evaluación» o simplemente «las funciones de `calccprop` disponibles para `eval()`». Y por otra parte con el `dict` que devuelve `setup()` — por ejemplo `{'a': 3, 'b': 7, 'suma': 10}`. Se llamará «espacio de nombres de la variante» porque es el conjunto de variables que ese problema concreto pone a disposición de los textos (`@a`, `@b`) y las semánticas (`lambda ns: ns['a'] + ns['b'] > 10`). Aquí el término más claro sería simplemente «diccionario de variables» o «variables de la variante».

## 8.2. Para un ProblemaVF

Un ProblemaVF tiene un formato más simple:

```
{
  "version": "1",
  "tipo": "ProblemaVF",
  "nombre": "identificador_opcional",
  "enunciado": "Marque las verdaderas:",
  "NumPreguntas": 3,
  "cuestiones": [
    {"texto": "Afirmación 1", "respuesta": true},
    {"texto": "Afirmación 2", "respuesta": false}
  ]
}
```

Un banco con múltiples problemas agrupa varios dicts bajo la clave "banco":

```
{
  "version": "1",
  "banco": [ {problema1}, {problema2}, ... ]
}
```

## Capítulo 9

# Evaluación de expresiones en el espacio de nombres de calcprop

Las semánticas y precondiciones se guardan en JSON como cadenas de texto y se recuperan evaluándolas con `eval()` en el espacio de nombres del módulo `calcprop`. Así, expresiones como `v('A') & v('B')` se resuelven correctamente al deserializar.

El espacio de nombres se construye una sola vez al importar el módulo:

---

```
_EVAL_NS = vars(_calcprop_mod).copy()
```

---

La función `_eval_expr` gestiona los casos especiales (booleanos literales) antes de delegar en `eval`, evitando así que `"True"` se evalúe como el nombre Python `True` solo si el usuario ha escrito exactamente esa cadena:

---

```
def _eval_expr(expr):
    """Evalúa una expresión (string o bool) en el namespace de calcprop."""
    if isinstance(expr, bool):
        return expr
    if expr == "True":
        return True
    if expr == "False":
        return False
    return eval(expr, _EVAL_NS)
```

---

## Capítulo 10

# Función de `setup` desde código Python

El campo `setup` permitirá realizar cálculos con python para dar valores a preguntas con parámetros.

Cuando el campo `setup` no es `null`, contiene el código Python (como cadena) que se ejecuta al comienzo de cada iteración de `ProblemaTipo`. La función `_make_setup_fn` lo convierte en un callable que devuelve el diccionario de variables definidas, excluyendo las que empiezan por guión bajo (convención para internas de Python):

---

Función `setup` desde string

```
def _make_setup_fn(code_str):
    """Crea un callable setup() a partir de código Python en string."""
    def setup():
        ns = {}
        exec(code_str, ns)
        return {k: v for k, v in ns.items() if not k.startswith('_')}
    return setup
```

---

Por ejemplo, si el JSON contiene:

```
"setup": "import random\nx = random.randint(1, 9)\ny = random.randint(1, 9)"
```

en cada iteración se ejecuta ese código y las variables `x` e `y` quedan disponibles para interpolar en los textos del problema mediante `@x` e `@y`.

En el ejemplo, `\n` indica un salto de línea en el script de python; es decir. el `setup` anterior se despliega en el script:

```
import random
x = random.randint(1, 9)
y = random.randint(1, 9)
```

# Capítulo 11

## Deserialización de componentes

La función `_componente_from_dict` crea un objeto `Supuesto` o `Cuestion` a partir de un dict JSON. Además de evaluar la semántica y la precondition, adjunta al objeto un atributo `_json` con los datos originales en formato de cadena. Ese atributo permite la serialización inversa (`problema_to_dict`) sin pérdida de información: el valor de cadena de la expresión se conserva tal como fue escrito por el usuario, sin que la evaluación lo destruya.

---

```
Componente desde dict JSON
def _componente_from_dict(d):
    tipo = d.get("tipo")
    enunciado = d["enunciado"]
    semantica_str = d["semantica"]
    precond_str = d.get("precond", "True")

    semantica = _eval_expr(semantica_str)
    precond = _eval_expr(precond_str)

    if tipo == "Supuesto":
        obj = Supuesto(enunciado, semantica, precond)
        obj._json = {"tipo": "Supuesto", "enunciado": enunciado,
                    "semantica": semantica_str, "precond": precond_str}
    elif tipo == "Cuestion":
        exp = d.get("exp", "")
        obj = Cuestion(enunciado, semantica, precond, exp)
        obj._json = {"tipo": "Cuestion", "enunciado": enunciado,
                    "semantica": semantica_str, "precond": precond_str, "exp": exp}
    else:
        raise ValueError(f"Tipo de componente desconocido: {tipo!r}")
    return obj
```

---

La función `_slot_from_json` despacha en función del tipo de cada elemento del slot (cadena, dict único o lista de dicts):

---

```
Slot desde JSON
def _slot_from_json(slot):
    if isinstance(slot, str):
        return slot
    elif isinstance(slot, dict):
        return _componente_from_dict(slot)
    elif isinstance(slot, list):
        return [_slot_from_json(item) for item in slot]
    else:
        raise ValueError(f"Componente JSON inválido: {slot!r}")
```

---

## Capítulo 12

# Serialización de componentes

La función `_slot_to_json` recorre el problema en sentido inverso. Cuando el objeto tiene el atributo `_json` (cargado desde JSON), lo devuelve directamente. Cuando el objeto fue creado a mano en Python, usa `repr()` sobre la semántica y la precondición: `repr()` de cualquier fórmula `calcprop` produce exactamente la cadena que `_eval_expr` sabe leer (p.ej. `v('A') >> v('C')`), por lo que el round-trip es transparente. Esta propiedad permite serializar problemas escritos directamente en Python sin modificarlos.

La función auxiliar `_expr_to_str` gestiona el caso especial de los callables (lambdas de `setup` paramétrico), cuyo código fuente no es recuperable desde el objeto función:

---

```
def _expr_to_str(val, campo):
    """Convierte una semántica o precondición al string JSON equivalente.

    Usa repr() sobre objetos calcprop (cuyo repr es evaluable con _eval_expr).
    Falla con un mensaje claro si val es un callable (lambda de setup paramétrico).
    """
    if callable(val):
        raise ValueError(
            f"El campo '{campo}' es un callable (lambda). "
            "Los componentes con semánticas o precondiciones lambda solo pueden "
            "serializarse si fueron cargados con load_problema() o problema_from_dict().")
    return repr(val)

def _slot_to_json(slot):
    if isinstance(slot, str):
        return slot
    elif isinstance(slot, (Supuesto, Cuestion)):
        if hasattr(slot, '_json'):
            return slot._json
        # Fallback para objetos creados directamente en Python:
        # repr() de cualquier fórmula calcprop es evaluable por _eval_expr.
        d = {
            'tipo': 'Supuesto' if isinstance(slot, Supuesto) else 'Cuestion',
            'enunciado': slot.e,
            'semantica': _expr_to_str(slot.s, 'semantica'),
            'precond': _expr_to_str(slot.p, 'precond'),
        }
        if isinstance(slot, Cuestion):
            d['exp'] = slot.x
        return d
    elif isinstance(slot, list):
        return [_slot_to_json(item) for item in slot]
    else:
        raise ValueError(f"Tipo no serializable: {type(slot)}")
```

---

## Capítulo 13

# Serialización de problemas completos

Las funciones públicas `problema_from_dict` y `problema_to_dict` realizan la conversión bidireccional entre dicts Python y objetos `ProblemaTipo` / `ProblemaVF`.

`problema_from_dict` distingue el tipo de problema por el campo "tipo" del dict, reconstruye los componentes llamando a `_slot_from_json` para cada uno, y convierte el campo "setup" (si no es null) en un callable mediante `_make_setup_fn`. Los metadatos `_nombre` y `_setup_str` se guardan como atributos del objeto para que `problema_to_dict` pueda recuperarlos sin necesidad de solicitarlos de nuevo al usuario.

---

```
def problema_from_dict(d):
    """Crea un ProblemaTipo o ProblemaVF a partir de un dict JSON."""
    tipo = d.get("tipo")

    if tipo == "ProblemaTipo":
        componentes = [_slot_from_json(s) for s in d["componentes"]]
        setup_str = d.get("setup")
        setup = _make_setup_fn(setup_str) if setup_str else None
        p = ProblemaTipo(componentes, setup=setup)
        p._nombre = d.get("nombre", "")
        p._setup_str = setup_str
        return p

    elif tipo == "ProblemaMultiParte":
        # Schema heredado: se aplanan a un ProblemaTipo multiparte unificado.
        # Cada subpregunta aporta su intro (material de enunciado) seguido de su
        # sublista de Cuestion; la transición cuestiones->enunciado segmenta las
        # partes (regla I2), de modo que por_partes() reconstruye las subpreguntas.
        componentes = [_slot_from_json(s) for s in d["componentes"]]
        for sp in d["subpreguntas"]:
            componentes.append(sp["intro"])
            componentes.append([_componente_from_dict(c) for c in sp["cuestiones"]])
        setup_str = d.get("setup")
        setup = _make_setup_fn(setup_str) if setup_str else None
        p = ProblemaTipo(componentes, setup=setup)
        p._nombre = d.get("nombre", "")
        p._setup_str = setup_str
        return p

    elif tipo == "ProblemaVF":
        enunciado = d["enunciado"]
        cuestiones = [(c["texto"], c["respuesta"]) for c in d["cuestiones"]]
        num = d["NumPreguntas"]
        p = ProblemaVF(enunciado, cuestiones, num)
        p._nombre = d.get("nombre", "")
        return p

    else:
        raise ValueError(f"Tipo de problema desconocido: {tipo!r}")
```

---

`problema_from_dict` reconstruye los problemas según la clave `tipo` del dict: `ProblemaTipo`, `ProblemaVF` y el schema heredado `ProblemaMultiParte`. Este último ya no produce un objeto aparte: se **aplana** a un `ProblemaTipo` multiparte (los componentes comunes seguidos, por cada subpregunta, de su intro y su sublista de `Cuestion`), de modo que las partes se infieren con la regla I2 y se recorren con `por_partes()`. Así los ficheros JSON antiguos siguen cargándose sin cambios; al re-guardarlos con `save_problema` migran al schema unificado de `ProblemaTipo`.

La función inversa, `problema_to_dict`, admite `ProblemaVF` (datos directamente serializables) y `ProblemaTipo` (incluidos los multiparte, ya que son listas planas). Delega en `_slot_to_json`: los componentes pueden haber sido cargados desde JSON (con atributo `_json`) o creados directamente en Python (en cuyo caso `_slot_to_json` usa `repr()`). La única restricción es que los `setup` definidos como funciones Python no se pueden serializar (su código fuente no es introspectable); en ese caso se lanza un error claro.

---

```

Problema a dict JSON
def problema_to_dict(problema):
    """Serializa un ProblemaTipo o ProblemaVF a dict.

    ProblemaTipo con componentes creados directamente en Python se serializa
    usando repr() sobre las fórmulas calcprop. Si alguna semántica o precondición
    es un callable (lambda de setup paramétrico), la serialización falla.
    Los setup definidos como funciones Python (no cargados desde JSON) no pueden
    serializarse; en ese caso se lanza un error. ProblemaVF siempre puede serializarse.
    """
    if isinstance(problema, ProblemaTipo):
        componentes = [_slot_to_json(s) for s in problema.e]
        setup_str = getattr(problema, '_setup_str', None)
        if setup_str is None and problema.setup is not None:
            raise ValueError(
                "ProblemaTipo tiene un setup callable pero no fue creado desde JSON. "
                "No es posible serializar un setup definido directamente en Python. "
                "Escribe el código del setup como cadena y carga el problema con "
                "problema_from_dict().")
        return {
            "version": "1",
            "tipo": "ProblemaTipo",
            "nombre": getattr(problema, '_nombre', ''),
            "setup": setup_str,
            "componentes": componentes,
        }
    elif isinstance(problema, ProblemaVF):
        return {
            "version": "1",
            "tipo": "ProblemaVF",
            "nombre": getattr(problema, '_nombre', ''),
            "enunciado": problema.e,
            "NumPreguntas": problema.NumPreguntas,
            "cuestiones": [{"texto": t, "respuesta": r} for t, r in problema.c],
        }
    else:
        raise ValueError(f"Tipo no soportado: {type(problema)}")

```

---

## Capítulo 14

# Lectura y escritura de ficheros

Las funciones de alto nivel abren y cierran los ficheros JSON y delegan en las funciones de conversión.

`load_problema` detecta si el fichero contiene un banco (clave "banco") y en ese caso devuelve un error indicando al usuario que debe usar `load_banco`. Esto evita silenciosamente cargar solo el primer problema cuando el usuario ha pasado un banco por error.

---

```
def load_problema(filepath):
    """Carga un único problema desde un fichero JSON."""
    with open(filepath, 'r', encoding='utf-8') as f:
        d = _json.load(f)
    if "banco" in d:
        raise ValueError(f"{filepath!r} contiene un banco de problemas. Usa load_banco().")
    return problema_from_dict(d)

def save_problema(problema, filepath):
    """Guarda un único problema en un fichero JSON."""
    d = problema_to_dict(problema)
    with open(filepath, 'w', encoding='utf-8') as f:
        _json.dump(d, f, ensure_ascii=False, indent=2)
```

---

`load_banco` acepta tanto ficheros de banco (con "banco") como ficheros de problema individual, en cuyo caso devuelve una lista de un elemento. `save_banco` acepta tanto listas como diccionarios (cuyas claves se ignoran, usándose solo los valores).

---

```
def load_banco(filepath):
    """Carga un banco de problemas desde un fichero JSON.
    Devuelve una lista de ProblemaTipo / ProblemaVF."""
    with open(filepath, 'r', encoding='utf-8') as f:
        d = _json.load(f)
    if "banco" in d:
        return [problema_from_dict(p) for p in d["banco"]]
    else:
        return [problema_from_dict(d)]

def save_banco(problemas, filepath):
    """Guarda una lista (o dict) de problemas en un fichero JSON de banco."""
    items = list(problemas.values() if isinstance(problemas, dict) else problemas)
    banco = [problema_to_dict(p) for p in items]
    d = {"version": "1", "banco": banco}
    with open(filepath, 'w', encoding='utf-8') as f:
        _json.dump(d, f, ensure_ascii=False, indent=2)
```

---

# Capítulo 15

## Ejemplo de uso

El siguiente ejemplo crea un `ProblemaTipo` desde un dict, lo itera, lo guarda en disco y lo recarga comprobando que el ciclo completo de serialización es transparente:

---

```
from qbank import problema_from_dict, save_problema, load_problema

d = {
    "version": "1",
    "tipo": "ProblemaTipo",
    "nombre": "ejemplo_json",
    "setup": None,
    "componentes": [
        "Considere ",
        [
            {"tipo": "Supuesto", "enunciado": "A ", "semantica": "v('A')", "precond": "True"},
            {"tipo": "Supuesto", "enunciado": "B ", "semantica": "v('B')", "precond": "True"}
        ],
        [
            {"tipo": "Cuestion", "enunciado": "¿A?", "semantica": "v('A')", "precond": "True", "exp": ""},
            {"tipo": "Cuestion", "enunciado": "¿B?", "semantica": "v('B')", "precond": "True", "exp": ""}
        ]
    ]
}

p = problema_from_dict(d)
for etiqueta, enunciado, cuestiones in p:
    print(etiqueta, enunciado, cuestiones)

save_problema(p, "./tmp/ejemplo.json")
p2 = load_problema("./tmp/ejemplo.json")
for etiqueta, enunciado, cuestiones in p2:
    print(etiqueta, enunciado, cuestiones)
```

---

## Capítulo 16

# Exportación a código Python editable

Además de guardar en JSON, es posible convertir un problema a código Python puro: una lista de listas con `Supuesto` y `Cuestion` tal y como se escribiría a mano. El resultado es un fichero `.py` legible y editable con cualquier editor de texto.

### 16.1. `_slot_to_python`

Convierte un slot (cadena, `Supuesto`, `Cuestion` o lista de alternativas) al fragmento de código Python equivalente. El parámetro `nivel` controla la indentación (cada nivel añade cuatro espacios).

- Una cadena de texto se escribe con `_py_text`: una *raw string* (`r'...'`) cuando es seguro, para que el  $\text{\LaTeX}$  se lea sin barras invertidas dobladas, o `repr()` en caso contrario.
- Un `Supuesto` o `Cuestion` (representado como dict JSON interno) se escribe como llamada al constructor con *todos* sus argumentos nombrados (`enunciado`, `semantica`, `precond` y, en `Cuestion`, `exp`), cada uno en su propia línea e indentado, incluyendo los valores por defecto. Así el guión resultante es fácil de leer y editar a mano.
- Una lista de alternativas se escribe como lista Python anidada.

La función recibe el slot en su representación de dict JSON (producida por `_slot_to_json`), no el objeto Python original, porque `problema_to_python` invoca primero `problema_to_dict` para normalizar la representación.

---

Exportación a código Python

```
# Exportación a código Python

def _py_text(s):
    """Literal de texto Python legible: usa una raw string cuando es seguro
    (para que el LaTeX se lea sin barras invertidas dobladas) y repr() en caso
    contrario (texto con comillas conflictivas, salto de línea o backslash final)."""
    if "\n" not in s:
        trailing_bs = len(s) - len(s.rstrip("\\"))
        if trailing_bs % 2 == 0:
            if "'" not in s:
                return f"r'{s}'"
            if '"' not in s:
                return f'r"{s}"'
    return repr(s)

def _slot_to_python(slot, nivel=1):
    pad = "    " * nivel

    if isinstance(slot, str):
        return f"{pad}{_py_text(slot)},"

    elif isinstance(slot, dict):
        # Constructor con todos los argumentos nombrados, uno por línea, para
        # que el guión Python resultante sea fácil de leer y editar a mano.
        tipo = slot["tipo"]
        ipad = "    " * (nivel + 1)
```

```

    args = [
        f"{ipad}enunciado = {_py_text(slot['enunciado'])},"
        f"{ipad}semantica = {slot['semantica']}," # cadena evaluable: "v('A')", "True", ...
        f"{ipad}precond = {slot.get('precond', 'True')}"
    ]
    if tipo == "Question":
        args[-1] += ","
        args.append(f"{ipad}exp = {_py_text(slot.get('exp', ''))}")
        cuerpo = "\n".join(args)
        return f"{pad}{tipo}(\n{cuerpo}\n{pad}),"

elif isinstance(slot, list):
    inner = "\n".join(_slot_to_python(item, nivel + 1) for item in slot)
    return f"{pad}[\n{inner}\n{pad}],"

raise ValueError(f"Tipo no convertible a Python: {type(slot)}")

def problema_to_python(problema, varname="ejercicio"):
    """Genera código Python que recrea el problema como listas editables.

    El fichero resultante puede abrirse con cualquier editor de texto,
    modificarse y ejecutarse directamente con Python. Los problemas multiparte
    se generan como un ProblemaTipo (lista plana), igual que los de una sola parte.
    Para problemas con setup, el código del setup queda como función Python
    editable (_setup). Si el setup es un callable sin _setup_str, falla.
    """
    d = problema_to_dict(problema)

    lines = ["from qbank import Supuesto, Cuestion, ProblemaTipo",
            "from calcprop import *", ""]

    setup_str = d.get("setup")
    if setup_str:
        lines += ["", "def _setup():"]
        for line in setup_str.splitlines():
            lines.append(f"    {line}")
        lines += [
            "    _locs = locals()",
            "    return {k: v for k, v in _locs.items() if not k.startswith('_)}",
            ""
        ]

    setup_arg = ", setup=_setup" if setup_str else ""

    lines.append(f"{varname} = [")
    for comp in d["componentes"]:
        lines.append(_slot_to_python(comp, nivel=1))
    lines.append("]")
    lines.append(f"p = ProblemaTipo({varname}{setup_arg})")

    lines.append("")
    return "\n".join(lines)

def save_problema_py(problema, filepath, varname="ejercicio"):
    """Guarda el problema como código Python editable en un fichero .py."""
    code = problema_to_python(problema, varname=varname)
    with open(filepath, "w", encoding="utf-8") as f:
        f.write(code)

```

## 16.2. problema\_to\_python y save\_problema\_py

problema\_to\_python es el punto de entrada público. Invoca problema\_to\_dict para obtener la representación JSON normalizada y luego genera el código Python línea a línea. El resultado es un fichero Python completo que:

1. Importa Supuesto, Cuestion, ProblemaTipo y todo de calcprop.
2. Define una función \_setup() si el problema tiene setup (el cuerpo de la función es exactamente el código Python almacenado en el campo "setup" del JSON).
3. Define la variable (varname) como lista de componentes y construye p = ProblemaTipo(varname) (o con setup=\_setup si procede). Los problemas multiparte se generan también como un ProblemaTipo (lista plana), sin tipo aparte.

El patrón \_locs = locals() en \_setup es necesario porque en Python 3 una comprensión de lista tiene su propio ámbito; capturar las variables locales antes de la comprensión evita que las variables de

iteración las sobrescriban.

`save_problema_py` simplemente escribe la cadena resultante en un fichero.

**Limitaciones:** solo funciona con `ProblemaTipo` (no con `ProblemaVF`). Si el problema tiene un `setup` definido como función Python (no cargado desde JSON), `problema_to_dict` falla antes de llegar aquí.

## 16.3. Ejemplo

---

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

---

## Parte III

# Editor visual de problemas en Jupyter

El módulo `qbank._widgets` proporciona `ProblemaTipoEditor`, un formulario interactivo basado en `ipywidgets` que permite construir, visualizar y guardar objetos `ProblemaTipo` desde un notebook de Jupyter sin escribir la estructura de listas a mano.

## Capítulo 17

# Dependencias e instalación

El módulo requiere el paquete `ipywidgets`. Si no está disponible, la importación falla silenciosamente: el bloque `try/except` de `qbank.__init__` hace que el resto del paquete siga funcionando con normalidad.

Para instalarlo en el entorno virtual:

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

En **JupyterLab 4** es necesario que la extensión `@jupyter-widgets/jupyterlab-manager` aparezca como habilitada en `jupyter labextension list`. Si los widgets muestran su representación textual (`VBox(children...=)` en lugar del formulario), el síntoma es que JupyterLab no está escaneando el directorio `share/jupyter/labextensions` del entorno virtual. La solución consiste en crear un enlace simbólico al directorio de extensiones del usuario:

```
mkdir -p ~/.local/share/jupyter/labextensions/@jupyter-widgets
ln -sfn /ruta/.venv/share/jupyter/labextensions/@jupyter-widgets/jupyterlab-manager \
    ~/.local/share/jupyter/labextensions/@jupyter-widgets/jupyterlab-manager
```

Tras reiniciar JupyterLab, `jupyter labextension list` debe mostrar la extensión como habilitada.

# Estructura del módulo qbank.\_widgets

---

```
qbank/_widgets.py
__all__ = ['ProblemaTipoEditor']

import io as _io
import contextlib as _contextlib

try:
    import ipywidgets as _w
    from IPython.display import display as _display, clear_output as _clear
    _HAS_WIDGETS = True
except ImportError:
    _HAS_WIDGETS = False

from qbank._quiz import (ProblemaTipo, Cuestion,
                        _plan_partes, _normaliza_slot)
from qbank._json import problema_from_dict, problema_to_dict, load_problema, save_problema, problema_to_python

def _need_widgets():
    if not _HAS_WIDGETS:
        raise ImportError(
            "ipywidgets no está instalado. Ejecuta:\n"
            "  pip install ipywidgets")

<<Widget de alternativa>>
<<Widget de slot>>
<<Editor principal ProblemaTipoEditor>>
```

---

## Capítulo 18

# Widget de alternativa (`_AltWidget`)

El widget `_AltWidget` representa una única alternativa dentro de un slot. Su **tipo** (`kind`) no lo elige la alternativa, sino que lo  **fija el slot padre**: `texto`, `Supuesto` o `Cuestion`. Así un slot nunca puede mezclar tipos (invariante de homogeneidad I1). Sus campos, según el tipo, son:

**e** (**enunciado**) texto de la alternativa (único campo del tipo `texto`).

**s** (**semántica**) expresión calcprop (cadena), o `True` / `False`, o una lambda para problemas paramétricos ("`lambda ns: ns['x'] >0`"); en `Supuesto` y `Cuestion`.

**p** (**precondición**) expresión calcprop; por defecto `True`; en `Supuesto` y `Cuestion`.

**x** (**explicación**) texto de feedback para Moodle; solo en `Cuestion`.

Los botones `/` reordenan la alternativa dentro de su slot (callback `_on_move`) y `la` elimina (`_on_delete`). Como el tipo es fijo, ya no hay dropdown ni lógica de visibilidad: cada `_AltWidget` construye exactamente los campos que su tipo necesita. El método `raw` devuelve los datos comunes (para conservarlos si el slot cambia de tipo) y `to_json` devuelve la cadena (tipo `texto`) o el dict (`Supuesto` / `Cuestion`) que espera `problema_from_dict`.

```
Widget de alternativa
# Widget de una alternativa (texto / Supuesto / Cuestion)

class _AltWidget:
    """Una alternativa dentro de un slot. Su tipo (=kind=) lo fija el slot
    padre, de modo que un slot nunca mezcla tipos (invariante I1)."""
    def __init__(self, kind='Cuestion', d=None, on_delete=None, on_move=None):
        self.kind = kind
        self._on_delete = on_delete
        self._on_move = on_move
        if isinstance(d, str):
            d = {'enunciado': d}
        elif d is None:
            d = {}

        self.e = _w.Text(
            value=d.get('enunciado', ''), placeholder='enunciado',
            layout=_w.Layout(width='260px'))

        up_btn = _w.Button(
            description='', tooltip='subir alternativa',
            layout=_w.Layout(width='auto', height='28px'))
        dn_btn = _w.Button(
            description='', tooltip='bajar alternativa',
            layout=_w.Layout(width='auto', height='28px'))
        up_btn.on_click(lambda _: self._on_move(self, -1) if self._on_move else None)
        dn_btn.on_click(lambda _: self._on_move(self, +1) if self._on_move else None)
        del_btn = _w.Button(
            description='', button_style='danger',
            layout=_w.Layout(width='auto', height='28px'))
        del_btn.on_click(lambda _: self._on_delete(self) if self._on_delete else None)

        if kind == 'texto':
            fila = [_w.Label('txt:', layout=_w.Layout(width='30px')), self.e]
        else:
            self.s = _w.Text(
                value=d.get('semantica', 'True'), placeholder="v('A')",
                layout=_w.Layout(width='130px'))
```

```

self.p = _w.Text(
    value=d.get('precond', 'True'), placeholder='precond',
    layout=_w.Layout(width='70px'))
fila = [
    _w.Label('e:', layout=_w.Layout(width='15px')), self.e,
    _w.Label('s:', layout=_w.Layout(width='15px')), self.s,
    _w.Label('p:', layout=_w.Layout(width='15px')), self.p,
]
if kind == 'Cuestion':
    self.x = _w.Text(
        value=d.get('exp', ''), placeholder='explicación',
        layout=_w.Layout(width='120px'))
    fila += [_w.Label('exp:', layout=_w.Layout(width='30px')), self.x]

self.box = _w.HBox(fila + [up_btn, dn_btn, del_btn])

def raw(self):
    """Datos comunes de la alternativa, para conservarlos cuando el slot
    padre cambia de tipo."""
    d = {'enunciado': self.e.value}
    if self._kind != 'texto':
        d['semantica'] = self.s.value
        d['precond'] = self.p.value
    if self._kind == 'Cuestion':
        d['exp'] = self.x.value
    return d

def to_json(self):
    if self._kind == 'texto':
        return self.e.value
    d = {'tipo': self._kind, 'enunciado': self.e.value,
        'semantica': self.s.value, 'precond': self.p.value}
    if self._kind == 'Cuestion':
        d['exp'] = self.x.value
    return d

```

---

## Capítulo 19

# Widget de slot (`_SlotWidget`)

Un slot es uno de los «huecos» estructurales del problema: una lista de alternativas **homogéneas** de un único tipo, que `ProblemaTipo` recorre eligiendo una por variante. El dropdown `_tipo_dd` fija ese tipo y, con él, el de todas las alternativas:

- **texto**: una o varias cadenas alternativas (texto del enunciado).
- **supuestos**: una o varias alternativas `Supuesto`.
- **cuestiones**: una o varias alternativas `Cuestion`.

Como el tipo vive en el slot (y no en cada alternativa), la homogeneidad I1 queda garantizada por construcción: ya no es posible mezclar `Supuesto` y `Cuestion` en una misma sublista. El botón + `alt` añade una fila del tipo del slot; el `_AltWidget` la reordena ( / , vía `_move_alt`) o elimina ( ). En la cabecera del slot, los botones / reordenan el slot entero dentro del problema (callback `_on_move` del editor), inserta un slot nuevo justo debajo (`_on_insert`, útil para añadir un slot a una parte anterior) y borra el slot. Al cambiar el tipo del slot, `_on_tipo` reconstruye las alternativas conservando los datos comunes (vía `raw`) y avisa al editor (`_on_change`) para repintar los separadores de parte.

El separador `Parte k` se dibuja **encima** del primer slot de cada parte, que por la regla I2 es siempre un slot de texto o supuestos posterior a un slot de cuestiones. Para que ese separador no aparezca de forma prematura mientras el docente aún encadena cuestiones, el botón + `slot` del editor crea el nuevo slot **con el mismo tipo que el último** (véase `_add_slot`): así, tras una cuestión se añade otra cuestión (misma parte, sin separador), y la parte nueva solo se marca cuando el docente cambia explícitamente el tipo del slot a texto o supuestos.

El método `to_json` devuelve el slot en el formato que espera `problema_from_dict`: una **cadena suelta** si el tipo es `texto` y hay una sola alternativa (un texto con una única alternativa se serializa como cadena directa, no dentro de una sublista), una **sublista de cadenas** si hay varias, o una **lista de dicts** para `supuestos / cuestiones`.

---

```
Widget de slot
# Widget de un slot (texto / supuestos / cuestiones)

class _SlotWidget:
    """Un «hueco» del problema: lista de alternativas homogéneas de un único
    tipo. El tipo del slot fija el de sus alternativas, garantizando I1."""

    # tipo de slot -> kind de las alternativas
    _KIND = {'texto': 'texto', 'supuestos': 'Supuesto', 'cuestiones': 'Cuestion'}

    def __init__(self, data=None, tipo=None, on_delete=None, on_change=None,
                 on_move=None, on_insert=None, index=0):
        self._on_delete = on_delete
        self._on_change = on_change
        self._on_move = on_move
        self._on_insert = on_insert
        self._alts = []

        if data is None and tipo is not None:
            tipo_init, alts_init = tipo, [] # slot nuevo del tipo indicado
        else:
            tipo_init, alts_init = self._inferir(data)
```

```

self._tipo_dd = _w.Dropdown(
    options=['texto', 'supuestos', 'cuestiones'], value=tipo_init,
    layout=_w.Layout(width='110px'))
self._lbl = _w.Label(
    f'Slot {index + 1}', layout=_w.Layout(width='52px'))

self._alts_box = _w.VBox([])
add_btn = _w.Button(
    description='+ alt', button_style='info',
    layout=_w.Layout(width='70px', height='26px'))
add_btn.on_click(lambda _: self._add_alt())

up_btn = _w.Button(
    description='', tooltip='subir slot',
    layout=_w.Layout(width='auto', height='28px'))
dn_btn = _w.Button(
    description='', tooltip='bajar slot',
    layout=_w.Layout(width='auto', height='28px'))
ins_btn = _w.Button(
    description='', tooltip='insertar slot debajo', button_style='success',
    layout=_w.Layout(width='auto', height='28px'))
up_btn.on_click(lambda _: self._on_move(self, -1) if self._on_move else None)
dn_btn.on_click(lambda _: self._on_move(self, +1) if self._on_move else None)
ins_btn.on_click(lambda _: self._on_insert(self) if self._on_insert else None)
del_btn = _w.Button(
    description='', button_style='danger',
    layout=_w.Layout(width='auto', height='28px'))
del_btn.on_click(lambda _: self._on_delete(self) if self._on_delete else None)

self._tipo_dd.observe(self._on_tipo, names='value')
self.box = _w.VBox([
    _w.HBox([self._lbl, self._tipo_dd, up_btn, dn_btn, ins_btn, del_btn]),
    self._alts_box,
    add_btn,
])

for a in alts_init:
    self._add_alt(a)
if not self._alts:
    self._add_alt()
    # un slot nuevo arranca con
    # una alternativa vacía

@staticmethod
def _inferir(data):
    """(tipo_slot, [datos_de_alternativa]) a partir del componente JSON."""
    if data is None:
        return 'texto', []
    if isinstance(data, str):
        return 'texto', [data]
    if isinstance(data, dict):
        # componente suelto
        tipo = data.get('tipo', 'Cuestion')
        return 'supuestos' if tipo == 'Supuesto' else 'cuestiones', [data]
    if not data:
        # lista vacía
        return 'texto', []
    first = data[0]
    # sublista homogénea
    if isinstance(first, str):
        return 'texto', list(data)
    tipo = first.get('tipo', 'Cuestion') if isinstance(first, dict) else 'Cuestion'
    return ('supuestos' if tipo == 'Supuesto' else 'cuestiones'), list(data)

def tipo(self):
    return self._tipo_dd.value

def _on_tipo(self, change):
    # Reconstruye las alternativas con el nuevo tipo conservando los datos
    # comunes (enunciado, etc.) y avisa al editor para repintar separadores.
    datos = [a.raw() for a in self._alts]
    self._alts = []
    for d in datos:
        self._add_alt(d)
    if not self._alts:
        self._add_alt()
    if self._on_change:
        self._on_change()

def _add_alt(self, d=None):
    kind = self._KIND[self._tipo_dd.value]
    alt = _AltWidget(kind=kind, d=d, on_delete=self._del_alt,
                    on_move=self._move_alt)
    self._alts.append(alt)
    self._render_alts()

def _del_alt(self, alt):
    self._alts = [a for a in self._alts if a is not alt]
    self._render_alts()

def _move_alt(self, alt, delta):
    i = self._alts.index(alt)

```

```

j = i + delta
if 0 <= j < len(self._alts):
    self._alts[i], self._alts[j] = self._alts[j], self._alts[i]
    self._render_alts()

def _render_alts(self):
    self._alts_box.children = [a.box for a in self._alts]

def update_label(self, i):
    self._lbl.value = f'Slot {i + 1}'

def to_json(self):
    if self._tipo_dd.value == 'texto':
        textos = [a.to_json() for a in self._alts]
        if len(textos) <= 1:
            return textos[0] if textos else '' # cadena suelta (no en [...])
        return textos # sublista de textos alternativos
    return [a.to_json() for a in self._alts]

```

---

## Capítulo 20

# Editor principal (ProblemaTipoEditor)

ProblemaTipoEditor es el widget de nivel superior. Al instanciarse llama a `_display(self._ui)`, lo que hace que el formulario se renderice inmediatamente en la celda del notebook.

El parámetro `source` admite cuatro tipos:

**None** el editor comienza vacío.

**Cadena de texto** ruta a un fichero JSON; el fichero se carga con `load_problema` y el campo `filepath` se actualiza automáticamente con la ruta indicada.

**dict** se interpreta directamente como descripción del problema (en el mismo formato JSON).

**ProblemaTipo** el objeto (que debe llevar el atributo `_json` en sus componentes, es decir, debe haber sido cargado con `load_problema` o `problema_from_dict`) se serializa con `problema_to_dict` y se muestra en el editor.

Los controles inferiores se disponen en **dos filas** para no saturar una sola: arriba la **previsualización** (`Preview`, `vars`, casilla vista `profe`, `{ }` JSON y `</>.py`) y abajo el **fichero** (`ruta`, `Guardar`, `Cargar`, `.json` y `.py`).

Botón	Acción
Preview	Muestra hasta <code>vars</code> variantes <b>por partes</b> (enunciado de cada parte y, debajo, sus cuestiones). La casilla <code>{ }</code> JSON
<code>&lt;/&gt;.py</code>	Muestra el JSON equivalente al estado actual del editor en el área de salida. Previsualiza en el área de salida el guión Python ( <code>problema_to_python</code> ), análogo a <code>{ }</code> JSON pero con <code>&lt;/&gt;.py</code>
Guardar	Serializa el estado actual y lo guarda en el fichero indicado en el campo <code>filepath</code> .
Cargar	Lee el fichero indicado y actualiza el editor con su contenido.
<code>.json</code>	Muestra en el área de salida un enlace HTML ( <code>data URI</code> , atributo <code>download</code> ) para guardar el JSON en <code>.json</code>
<code>.py</code>	Igual que <code>.json</code> pero descarga el guión Python ( <code>problema_to_python</code> ), con la extensión <code>.py</code> .

El método `to_dict()` devuelve el dict JSON del estado actual del editor. El método `to_problema()` devuelve un `ProblemaTipo` listo para iterar, construyéndolo con `problema_from_dict(self.to_dict())`.

```
Editor principal ProblemaTipoEditor
# Editor principal

class ProblemaTipoEditor:
    """Editor visual de ProblemaTipo para Jupyter.

    Parameters
    -----
    source : str | dict | ProblemaTipo | None
        Fuente inicial. Puede ser la ruta a un fichero JSON, un dict,
        un objeto ProblemaTipo (creado con load_problema), o None para
        empezar un problema nuevo.
    """

    def __init__(self, source=None):
        _need_widgets()
        self._slots = []
```

```

# Cabecera
self._nombre = _w.Text(
    value='', placeholder='nombre del ejercicio',
    layout=_w.Layout(width='320px'))
self._setup = _w.Textarea(
    value='', placeholder='código Python del setup (opcional)',
    layout=_w.Layout(width='520px', height='58px'))

header = _w.VBox([
    _w.HBox([_w.Label('Nombre:', layout=_w.Layout(width='65px')),
            self._nombre]),
    _w.HBox([_w.Label('Setup:', layout=_w.Layout(width='65px')),
            self._setup]),
])

# Zona de slots
self._slots_box = _w.VBox([])
add_slot_btn = _w.Button(
    description='+ slot', button_style='success',
    layout=_w.Layout(width='85px'))
add_slot_btn.on_click(lambda _: self._add_slot())

# Controles inferiores
self._filepath = _w.Text(
    value='problema.json', placeholder='ruta del fichero JSON',
    layout=_w.Layout(width='260px'))
self._n_prev = _w.BoundedIntText(
    value=5, min=1, max=100,
    layout=_w.Layout(width='55px'))
self._profe = _w.Checkbox(
    value=True, description='vista profe', indent=False,
    layout=_w.Layout(width='110px'))

prev_btn = _w.Button(description=' Preview', button_style='primary',
    layout=_w.Layout(width='105px'))
json_btn = _w.Button(description='{ } JSON', layout=_w.Layout(width='95px'))
showpy_btn = _w.Button(description='</> .py', layout=_w.Layout(width='95px'))
save_btn = _w.Button(description=' Guardar', layout=_w.Layout(width='95px'))
load_btn = _w.Button(description=' Cargar', layout=_w.Layout(width='95px'))
dl_btn = _w.Button(description='.json', layout=_w.Layout(width='85px'))
py_btn = _w.Button(description='.py', layout=_w.Layout(width='80px'))

prev_btn.on_click(self._on_preview)
json_btn.on_click(self._on_show_json)
showpy_btn.on_click(self._on_show_py)
save_btn.on_click(self._on_save)
load_btn.on_click(self._on_load)
dl_btn.on_click(self._on_download)
py_btn.on_click(self._on_download_py)

self._out = _w.Output()

# Dos filas de controles: previsualización arriba, fichero/descargas abajo.
fila_preview = _w.HBox([
    prev_btn,
    _w.Label('vars:', layout=_w.Layout(width='32px')),
    self._n_prev,
    self._profe,
    _w.Label(' ', layout=_w.Layout(width='12px')),
    json_btn, showpy_btn,
])
fila_fichero = _w.HBox([
    self._filepath,
    save_btn, load_btn,
    _w.Label(' ', layout=_w.Layout(width='12px')),
    dl_btn, py_btn,
])

self._ui = _w.VBox([
    header,
    _w.HTML('<hr style="margin:4px 0">'),
    self._slots_box,
    add_slot_btn,
    _w.HTML('<hr style="margin:4px 0">'),
    fila_preview,
    fila_fichero,
    self._out,
])

# Carga inicial
if source is not None:
    self._load_source(source)

_display(self._ui)

# Gestión de slots

```

```

def _make_slot(self, data=None, tipo=None):
    return _SlotWidget(data=data, tipo=tipo, on_delete=self._del_slot,
                       on_change=self._refresh, on_move=self._move_slot,
                       on_insert=self._insert_slot_after)

def _add_slot(self, data=None):
    # Un slot nuevo (botón «+ slot») hereda el tipo del último, de modo que
    # encadenar varias cuestiones no dispare un separador de parte prematuro:
    # la parte nueva solo se marca cuando el docente elige un tipo texto o
    # supuestos tras las cuestiones (inicio explícito de la nueva parte).
    tipo = self._slots[-1].tipo() if (data is None and self._slots) else None
    self._slots.append(self._make_slot(data=data, tipo=tipo))
    self._refresh()

def _insert_slot_after(self, slot):
    # Inserta un slot nuevo justo debajo de `slot`, heredando su tipo (p. ej.
    # para añadir otra sublista a una parte anterior antes del separador).
    i = self._slots.index(slot)
    self._slots.insert(i + 1, self._make_slot(tipo=slot.tipo()))
    self._refresh()

def _move_slot(self, slot, delta):
    i = self._slots.index(slot)
    j = i + delta
    if 0 <= j < len(self._slots):
        self._slots[i], self._slots[j] = self._slots[j], self._slots[i]
        self._refresh()

def _del_slot(self, slot):
    self._slots = [s for s in self._slots if s is not slot]
    self._refresh()

def _plan_partes(self):
    """Índice de parte de cada slot (regla I2), reutilizando la lógica
    canónica de =_quiz=: un slot 'cuestiones' aporta una Cuestion; los
    demás (texto / supuestos), una cadena."""
    muestra = [Cuestion(' ', 'True') if s.tipo() == 'cuestiones' else ' '
              for s in self._slots]
    return _plan_partes([_normaliza_slot(x) for x in muestra])

def _refresh(self):
    plan = self._plan_partes()
    npar = (plan[-1] + 1) if plan else 1
    children, parte_actual = [], -1
    for i, s in enumerate(self._slots):
        s.update_label(i)
        if npar > 1 and plan[i] != parte_actual: # frontera de parte (I2)
            parte_actual = plan[i]
            children.append(_w.HTML(
                f'<div style="color:#888;font-weight:bold;margin:6px 0 2px">
                f' Parte {parte_actual + 1} </div>'))
        children.append(s.box)
    self._slots_box.children = children

# Carga de datos

def _load_source(self, source):
    if isinstance(source, str):
        p = load_problema(source)
        self._filepath.value = source
        d = problema_to_dict(p)
    elif isinstance(source, ProblemaTipo):
        d = problema_to_dict(source)
    elif isinstance(source, dict):
        d = source
    else:
        raise ValueError(f"Fuente no reconocida: {type(source)}")

    self._nombre.value = d.get('nombre', '')
    self._setup.value = d.get('setup', '') or ''
    self._slots = []
    for comp in d.get('componentes', []):
        self._add_slot(data=comp)

# Serialización

def to_dict(self):
    """Devuelve el problema actual como dict JSON."""
    setup = self._setup.value.strip() or None
    return {
        'version': '1',
        'tipo': 'ProblemaTipo',
        'nombre': self._nombre.value,
        'setup': setup,
        'componentes': [s.to_json() for s in self._slots],
    }

```

```

def to_problema(self):
    """Devuelve un ProblemaTipo listo para iterar."""
    return problema_from_dict(self.to_dict())

# Callbacks de botones

def _on_preview(self, _):
    with self._out:
        _clear()
        try:
            # Vista por partes: cada parte muestra su enunciado en una línea
            # y, debajo, sus cuestiones; el enunciado de la parte k+1 aparece
            # tras las cuestiones de la parte k (no concatenado al principio).
            # Con la casilla «vista profe» desmarcada se usa por_partes() -la
            # misma salida que ven el alumno y los exportadores (WYSIWYG)-;
            # marcada, por_partes_profe() muestra TODAS las cuestiones de cada
            # sublista y marca como rechazadas ( ) las que no superan su
            # precondición, en vez de descartar la variante.
            p = self.to_problema()
            n = self._n_prev.value
            profe = self._profe.value
            if profe:
                print("> vista profe: todas las cuestiones; = rechazada "
                      "por precondición (no la ve el alumno).\n")
            variantes = p.por_partes_profe() if profe else p.por_partes()
            cnt = 0
            for etiqueta, partes in variantes:
                print(f" Variante {etiqueta} ")
                for enunciado, cuestiones in partes:
                    if enunciado:
                        print(enunciado)
                    for c in cuestiones:
                        if c[1] is True:
                            print(f" {c[0]}")
                        elif c[1] is False:
                            print(f" {c[0]}")
                        else:
                            # rechazada: c[1] es el motivo
                            print(f" {c[0]} [{c[1]}")

                    print()
                    cnt += 1
                    if cnt >= n:
                        break
            if cnt == 0:
                print("(sin variantes)")
            if not profe:
                with _contextlib.redirect_stdout(_io.StringIO()):
                    remaining = sum(1 for _ in variantes)
                    total = cnt + remaining
                    print(f"> {total} variante{'s' if total != 1 else ''} válida{'s' if total != 1 else ''} en total.")
        except Exception as exc:
            print(f"Error: {exc}")

def _on_save(self, _):
    with self._out:
        _clear()
        try:
            path = self._filepath.value.strip()
            p = problema_from_dict(self.to_dict())
            save_problema(p, path)
            print(f" Guardado en {path!r}")
        except Exception as exc:
            print(f"Error al guardar: {exc}")

def _on_load(self, _):
    with self._out:
        _clear()
        try:
            path = self._filepath.value.strip()
            self._slots = []
            self._load_source(path)
            print(f" Cargado desde {path!r}")
        except Exception as exc:
            print(f"Error al cargar: {exc}")

def _on_show_json(self, _):
    with self._out:
        _clear()
        try:
            import json
            print(json.dumps(self.to_dict(), ensure_ascii=False, indent=2))
        except Exception as exc:
            print(f"Error: {exc}")

def _on_show_py(self, _):
    # Previsualiza el guión Python (problema_to_python) en el área de salida,
    # de forma análoga a {JSON}, sin generar el enlace de descarga.

```

```

with self._out:
    _clear()
    try:
        p = problema_from_dict(self.to_dict())
        print(problema_to_python(p))
    except Exception as exc:
        print(f"Error: {exc}")

def _on_download(self, _):
    # JupyterLab no ejecuta el JS de IPython.display.Javascript desde un
    # callback de botón (solo muestra «<IPython.core.display.Javascript object>»).
    # En su lugar mostramos un enlace HTML con el JSON embebido como data URI
    # (atributo `download`): el navegador lo descarga al pulsarlo, sin ejecutar JS.
    import json as _j
    import os, base64
    from IPython.display import HTML
    with self._out:
        _clear()
        try:
            data = _j.dumps(self.to_dict(), ensure_ascii=False, indent=2)
            filename = os.path.basename(self._filepath.value.strip() or 'problema.json')
            b64 = base64.b64encode(data.encode('utf-8')).decode('ascii')
            href = f"data:application/json;base64,{b64}"
            _display(HTML(
                f'<a download="{filename}" href="{href}" '
                f'style="font-size:14px;font-weight:bold">'
                f' Descargar {filename}</a>'
                f'<br><span style="color:gray">(pulsar el enlace para guardar el fichero)</span>'))
        except Exception as exc:
            print(f"Error al descargar: {exc}")

def _on_download_py(self, _):
    # Igual que _on_download pero exporta el problema como guión Python
    # editable (problema_to_python). El nombre del fichero se deriva de la
    # ruta JSON cambiando la extensión a .py.
    import os, base64
    from IPython.display import HTML
    with self._out:
        _clear()
        try:
            p = problema_from_dict(self.to_dict())
            code = problema_to_python(p)
            base = os.path.basename(self._filepath.value.strip() or 'problema.json')
            filename = os.path.splitext(base)[0] + '.py'
            b64 = base64.b64encode(code.encode('utf-8')).decode('ascii')
            href = f"data:text/x-python;base64,{b64}"
            _display(HTML(
                f'<a download="{filename}" href="{href}" '
                f'style="font-size:14px;font-weight:bold">'
                f' Descargar {filename}</a>'
                f'<br><span style="color:gray">(pulsar el enlace para guardar el fichero)</span>'))
        except Exception as exc:
            print(f"Error al descargar: {exc}")

```

---

## Parte IV

¿Y ahora qué?

## 20.1. ¿Qué tenemos?

- Contamos con un módulo de cálculo proposicional que determina la veracidad o falsedad de cada enunciado, basada en los supuestos de cada caso particular.
- Poseemos una metodología para programar ejercicios utilizando listas de sublistas que incluyen textos, supuestos y cuestiones.
- Tenemos la capacidad de desempaquetar dichas listas mediante `ProblemaTipo`, lo que nos permite generar múltiples ejercicios, con la tranquilidad de que el módulo de cálculo proposicional se encarga de identificar qué opciones son verdaderas y cuáles son falsas, de acuerdo con la composición de los supuestos de cada enunciado.

Con esto, hemos desarrollado un procedimiento para generar un amplio conjunto de variantes de ejercicios de opción múltiple.

## 20.2. ¿Qué nos falta?

Falta dar formato a lo que hemos desarrollado para ser usado con los estudiantes. El módulo `CalcProp-export` nos permitirá crear bancos de preguntas en formato  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$ , que podrán ser utilizados con [Auto Multiple Choice \(AMC\)](#), así como también generar bancos en formato `xml` para su empleo en el entorno Moodle mediante el paquete de  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  `moodle`, específicamente para cuestiones de [opción múltiple](#).