Introducción a la Programación

Guía Práctica 10
Guía integradora - Python

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ejercicio 1. Veterinaria - Stock

En la veterinaria "Exactas's pets", al finalizar cada día, el personal registra en papeles los nombres y la cantidad actual de los productos cuyo stock ha cambiado. Para mejorar la gestión, desde la dirección de la veterinaria han pedido desarrollar una solución en Python que les permita analizar las fluctuaciones del stock. Se pide implementar una función, que reciba una lista de tuplas donde cada tupla contiene el nombre de un producto y su stock en ese momento. La función debe procesar esta lista y devolver un diccionario que tenga como clave el nombre del producto y como valor una tupla con su mínimo y máximo stock histórico registrado.

`problema stock_productos` (in `stock_cambios` : `seq`$\langle$`str` $\times \mathbb{Z}\rangle$) : `seq`$\langle\mathbb{Z}\rangle$ {
    `requiere:` {Todos los elementos de stock_cambios están formados por un str no vacío y un entero ≥ 0.}
    `asegura:` {res tiene como claves solo los primeros elementos de las tuplas de stock_cambios (o sea, un producto).}
    `asegura:` {res tiene como claves todos los primeros elementos de las tuplas de stock_cambios.}
    `asegura:` {El valor en res de un producto es una tupla de cantidades. Su primer elemento es la menor cantidad de ese producto en stock_cambios y como segundo valor el mayor.}
}

Ejercicio 2. Veterinaria - Filtrar códigos de barra

El hijo del dueño de la veterinaria, cuya actividad principal es ver tik toks, cree que los productos cuyos códigos de barras terminan en números primos son especialmente auspiciosos y deben ser destacados en la tienda. Luego de convencer a su padre de esta idea, solicita una función en Python que facilite esta gestión. Se pide implementar una función que, dada una secuencia de enteros, cada uno representando un código de barras de un producto, cree y devuelva una nueva lista que contenga únicamente aquellos números de la lista original cuyos últimos tres dígitos formen un número primo (por ejemplo, 101, 002 y 011).

Nota: Un número primo es aquel que solo es divisible por sí mismo y por 1. Algunos ejemplos de números primos de hasta tres dígitos son: 2, 3, 5, 101, 103, 107, etc.

`problema filtrar_codigos_primos` (in codigos_barra : $seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {
    `requiere:` {Todos los enteros de codigos_barra tienen, por lo menos, 3 dígitos.}
    `requiere:` {No hay elementos repetidos en codigos_barra.}
    `asegura:` {Los últimos 3 dígitos de cada uno de los elementos de res forman un número primo.}
    `asegura:` {Todos los elementos de codigos_barra cuyos últimos 3 dígitos forman un número primo están en res.}
    `asegura:` {Todos los elementos de res están en codigos_barra.}
}

Ejercicio 3. Veterinaria - Flujo de pacientes

Con el objetivo de organizar el flujo de pacientes, en una veterinaria se anotan los tipos de mascotas que van ingresando al local. Se necesita identificar las consultas que involucran solo a perros y gatos. Por eso, se decide desarrollar una función en Python que encuentre la secuencia más larga de consultas consecutivas que solo contenga los tipos de mascota "perro" o "gato". Se pide implementar una función que, dada una secuencia de strs, que representan los tipos de animales atendidos, devuelva el índice donde comienza la subsecuencia más larga que cumpla con estas condiciones.

`problema subsecuencia_mas_larga` (in tipos_pacientes_atendidos : $seq\langle str\rangle$) : $\mathbb{Z}$ {
    `requiere:` {tipos_pacientes_atendidos tiene, por lo menos, un elemento "perro" o "gato".}
    `asegura:` {res es el índice donde empieza la subsecuencia más larga de tipos_pacientes_atendidos que contenga solo elementos "perro" o "gato".}
    `asegura:` {Si hay más de una subsecuencia de tamaño máximo, res tiene el índice de la primera.}
}

Ejercicio 4. Veterinaria - Tabla turnos

Las personas responsables de los turnos están anotadas en una matriz donde las columnas representan los días, en orden de lunes a domingo, y cada fila un rango de una hora. Hay cuatro filas para los turnos de la mañana (9, 10, 11 y 12 hs) y otras cuatro para la tarde (14, 15, 16 y 17).

Para hacer más eficiente el trabajo del personal de una veterinaria, se necesita analizar si quienes quedan de responsables, están asignadas de manera continuada en los turnos de cada día.

Para ello se pide desarrollar una función en Python que, dada la matriz de turnos, devuelva una lista de tuplas de bool, una por cada día. Cada tupla debe contener dos elementos. El primer elemento debe ser True si y solo si todos los valores de los turnos de la mañana para ese día son iguales entre sí. El segundo elemento debe ser True si y solo si todos los valores de los turnos de la tarde para ese día son iguales entre sí.

Siempre hay una persona responsable en cualquier horario de la veterinaria.

```
problema un_responsable_por_turno (in grilla_horaria : seq⟨seq⟨str⟩⟩) : seq⟨Bool × Bool⟩ {
    requiere: {|grilla_horaria| = 8.}
    requiere: {Todos los elementos de grilla_horaria tienen el mismo tamaño (mayor a 0 y menor 8).}
    requiere: {No hay cadenas vacías en las listas de grilla_horaria.}
    asegura: {|res| = |grilla_horaria[0]|.}
    asegura: {El primer valor de la tupla en res [i], con i:ℤ, 0 res| es igual a True los primeros 4 valores de la columna i de grilla_horaria son iguales entre sí.}
    asegura: {El segundo valor de la tupla en res [i], con i:ℤ, 0 res| es igual a True los últimos 4 valores de la columna i de grilla_horaria son iguales entre sí.}
}
```

Ejercicio 5. Sala de Escape - Promedio de salidas

Un grupo de amigos apasionados por las salas de escape, esas aventuras inmersivas donde tienen 60 minutos para salir de una habitación resolviendo enigmas, llevan un registro meticuloso de todas las salas de escape que hay en Capital. Este registro indica si han visitado una sala y si pudieron o no salir de ella. Un 0 significa que no fueron, un 61 que no lograron salir a tiempo, y un número entre 1 y 60 representa los minutos que les tomó escapar exitosamente. Con estos datos, pueden comparar sus logros y desafíos en cada nueva aventura que emprenden juntos.

Dado un diccionario donde la clave es el nombre de cada amigo y el valor es una lista de los tiempos (en minutos) registrados para cada sala de escape en Capital, escribir una función en Python que devuelva un diccionario. En este nuevo diccionario, las claves deben ser los nombres de los amigos y los valores deben ser tuplas que indiquen la cantidad de salas de las que cada persona logró salir y el promedio de los tiempos de salida (solo considerando las salas de las que lograron salir).

```
problema promedio_de_salidas (in registro: dict⟨str, seq⟨ℤ⟩⟩) : dict⟨str, ⟨ℤ×ℝ⟩⟩ {
    requiere: {registro tiene por lo menos un integrante.}
    requiere: {Todos los integrantes de registro tiene por lo menos un tiempo.}
    requiere: {Todos los valores de registro tiene la misma longitud.}
    requiere: {Todos los tiempos de los valores de registro están entre 0 y 61 inclusive.}
    asegura: {res tiene las mismas claves que registro.}
    asegura: {El primer elemento de la tupla de res para un integrante, es la cantidad de salas con tiempo mayor estricto a 0 y menor estricto a 61 que figuran en sus valores de registro.}
    asegura: {El segundo elemento de la tupla de res para un integrante, si la cantidad de salas de las que salió es mayor a 0: es el promedio de salas con tiempo mayor estricto a 0 y menor estricto a 61 que figuran en sus valores de registro; sino es 0.0.}
}
```

Ejercicio 6. Sala de Escape - Tiempo más rápido

Dada una lista con los tiempos (en minutos) registrados para cada sala de escape de Capital, escribir una función en Python que devuelva la posición (índice) en la cual se encuentra el tiempo más rápido, excluyendo las salas en las que no haya salido (0 o mayor a 60).

```
problema tiempo_mas_rapido (in tiempos_salas: seq⟨ℤ⟩) : ℤ {
    requiere: {Hay por lo menos un elemento en tiempos_salas entre 1 y 60 inclusive.}
    requiere: {Todos los tiempos en tiempos_salas están entre 0 y 61 inclusive.}
    asegura: {res es la posición de la sala en tiempos_salas de la que más rápido se salió (en caso que haya más de una, devolver la primera, osea la de menor índice).}
}
```

Ejercicio 7. Sala de Escape - Racha más larga

Dada una lista con los tiempos (en minutos) registrados para cada sala de escape a la que fue una persona, escribir una función en Python que devuelva una tupla con el índice de inicio y el índice de fin de la subsecuencia más larga de salidas exitosas de salas de escape consecutivas.

```
problema racha_mas_larga (in tiempos: seq⟨ℤ⟩) : ⟨ℤ×ℤ⟩ {
    requiere: {Hay por lo menos un elemento en tiempos entre 1 y 60 inclusive.}
    requiere: {Todos los tiempos en tiempos están entre 0 y 61 inclusive.}
    asegura: {En la primera posición de res está la posición (índice de la lista) de la sala que inicia la racha más larga.}
    asegura: {En la segunda posición de res está la posición (índice de la lista) de la sala que finaliza la racha más larga.}
    asegura: {El elemento de la primer posición de res en tiempos es mayor estricto 0 y menor estricto que 61.}
    asegura: {El elemento de la segunda posición de res en tiempos es mayor estricto 0 y menor estricto que 61.}
    asegura: {La primera posición de res es menor o igual a la segunda posición de res.}
    asegura: {No hay valores iguales a 0 o a 61 en tiempos entre la primer posición de res y la segunda posición de res.}
    asegura: {No hay otra subsecuencia de salidas exitosas, en tiempos, de mayor longitud que la que está entre la primer
    posición de res y la segunda posición de res.}
    asegura: {Si hay dos o más subsecuencias de salidas exitosas de mayor longitud en tiempos, res debe contener la primera
    de ellas.}
}
```

Ejercicio 8. Sala de Escape - Escape en solitario

Dada una matriz donde las columnas representan a cada amigo y las filas representan las salas de escape, y los valores son los tiempos (en minutos) registrados para cada sala (0 si no fueron, 61 si no salieron, y un número entre 1 y 60 si salieron), escribir una función en Python que devuelva los índices de todas las filas (que representan las salas) en las cuales el primer, segundo y cuarto amigo no fueron (0), pero el tercero sí fue independientemente de si salió o no).

```
problema escape_en_solitario (in amigos_por_salas: seq⟨seq⟨ℤ⟩⟩) : seq⟨ℤ⟩ {
    requiere: {Hay por lo menos una sala en amigos_por_salas.}
    requiere: {Hay 4 amigos en amigos_por_salas.}
    requiere: {Todos los tiempos en cada sala de amigos_por_salas están entre 0 y 61 inclusive.}
    asegura: {La longitud de res es menor igual que la longitud de amigos_por_salas.}
    asegura: {Por cada sala en amigos_por_salas cuyo primer, segundo y cuarto valor sea 0, y el tercer valor sea distinto de
    0, la posición de dicha sala en amigos_por_salas debe aparecer res.}
    asegura: {Para todo i perteneciente a res se cumple que el primer, segundo y cuarto valor de amigos_por_salas[i] es 0, y
    el tercer valor es distinto de 0.}
}
```

Ejercicio 9. Juego de la Gallina

El juego del gallina es una competición en la que dos participantes conducen un vehículo en dirección al del contrario; si alguno se desvía de la trayectoria de choque pierde y es humillado por comportarse como un "gallina". Se hizo un torneo para ver quién es el menos gallina. Juegan todos contra todos una vez y van sumando puntos, o restando. Si dos jugadores juegan y se chocan entre sí, entonces pierde cada uno 5 puntos por haberse dañado. Si ambos jugadores se desvían, pierde cada uno 10 puntos por gallinas. Si uno no se desvía y el otro sí, el gallina pierde 15 puntos por ser humillado y el ganador suma 10 puntos! En este torneo, cada persona que participa tiene una estrategia predefinida para competir: o siempre se devía, o nunca lo hace. Se debe programar la función 'torneo_de_gallinas' que recibe un diccionario (donde las claves representan los nombres de los participantes que se anotaron en el torneo, y los valores sus respectivas estrategias) y devuelve un diccionario con los puntajes obtendidos por cada jugador.

```
problema torneo_de_gallinas (in estrategias: dict⟨str, str⟩) : dict⟨str, ℤ⟩ {
    requiere: {estrategias tiene por lo menos 2 elementos (jugadores).}
    requiere: {Las claves de estrategias tienen longitud mayor a 0.}
    requiere: {Los valores de estrategias sólo pueden ser los strs "me desvío siempre" ó "me la banco y no me desvío".}
    asegura: {Las claves de res y las claves de estrategias son iguales.}
    asegura: {Para cada jugador p perteneciente a claves(estrategias), res[p] es igual a la cantidad de puntos que obtuvo al
    finalizar el torneo, dado que jugó una vez contra cada otro jugador.}
}
```

Ejercicio 10. Cola en el Banco

En el banco ExactaBank los clientes hacen cola para ser atendidos por un representante. Los clientes son representados por las tuplas (nombre, tipo afiliado) donde la primera componente es el nombre y el tipo afiliado puede ser "común" o "vip".

Se nos pide implementar una función en python que dada una cola de clientes del banco, devuelva una nueva cola con los mismos clientes pero en donde los clientes vip están primero que los clientes comunes manteniendo el orden original de los clientes vips y los comunes entre sí.

`problema reordenar_cola_priorizando_vips` (in filaClientes: Cola⟨str × str⟩) : Cola⟨str⟩ {
- `requiere:` {La longitud de los valores de la primera componente de las tuplas de la cola filaClientes es mayor a 0.}
- `requiere:` {Los valores de la segunda componente de las tuplas de la cola filaClientes son "común" o "vip".}
- `requiere:` {No hay dos tuplas en filaClientes que tengan la primera componente iguales entre sí.}
- `asegura:` {todo valor de res aparece como primera componente de alguna tupla de filaClientes.}
- `asegura:` {|res| = |filaCliente|.}
- `asegura:` {res no tiene elementos repetidos.}
- `asegura:` {No hay ningún cliente "común" antes que un "vip" en res.}
- `asegura:` {Para todo cliente c1 y cliente c2 de tipo "común" pertenecientes a filaClientes si c1 aparece antes que c2 en filaClientes entonces el nombre de c1 aparece antes que el nombre de c2 en res.}
- `asegura:` {Para todo cliente c1 y cliente c2 de tipo "vip" pertenecientes a filaClientes si c1 aparece antes que c2 en filaClientes entonces el nombre de c1 aparece antes que el nombre de c2 en res.}

}

Ejercicio 11. Sufijos que son palíndromos

Decimos que una palabra es palíndromo si se lee igual de izquierda a derecha que de derecha a izquierda. Se nos pide programar en python la siguiente función:

`problema cuantos_sufijos_son_palindromos` (in texto: str) : $\mathbb{Z}$ {
- `requiere:` {True}
- `asegura:` {res es igual a la cantidad de palíndromos que hay en el conjunto de sufijos de texto.}

}

Nota: un sufijo es una subsecuencia de texto que va desde una posición cualquiera hasta el al final de la palabra. Ej: "Diego", el conjunto de sufijos es: "Diego", "iego","ego","go", "o". Para este ejercicio no consideraremos a "" como sufijo de ningún texto.

Ejercicio 12. Ta-Te-Ti-Facilito

Ana y Beto juegan al Ta-Te-Ti-Facilito. El juego es en un tablero cuadrado de lado entre 5 y 10. Cada jugador va poniendo su ficha en cada turno. Juegan intercaladamente y comienza Ana. Ana pone siempre una "X" en su turno y Beto pone una "O" en el suyo. Gana la persona que logra poner 3 fichas suyas consecutivas en forma vertical. Si el tablero está completo y no ganó nadie, entonces se declara un empate. El tablero comienza vacío, representado por " " en cada posición. Notar que dado que juegan por turnos y comienza Ana poniendo una "X" se cumple que la cantidad de "X" es igual a la cantidad de "O" o bien la cantidad de "X" son uno más que la cantidad de "O". Se nos pide implementar una función en python quien_gano_el_tateti_facilito que determine si ganó alguno, o si Beto hizo trampa (puso una "O" cuando Ana ya había ganado).

`problema quien_gano_el_tateti_facilito` (in tablero:$seq\langle seq\langle Char\rangle\rangle$) : $\mathbb{Z}$ {
- `requiere:` {tablero es una matriz cuadrada.}
- `requiere:` {$5 \leq |\text{tablero}[0]| \leq 10$.}
- `requiere:` {tablero sólo tiene "X", "O" y "" (espacio vacío) como elementos.}
- `requiere:` {En tablero la cantidad de "X" es igual a la cantidad de "O" o bien la cantidad de "X" es uno más que la cantidad de "O".}
- `asegura:` {res = 1 ⇔ hay tres "X" consecutivas en forma vertical(misma columna) y no hay tres "O" consecutivas en forma vertical(misma columna).}
- `asegura:` {res = 2 ⇔ hay tres "O" consecutivas en forma vertical (misma columna) y no hay tres "X" consecutivas en forma vertical(misma columna).}
- `asegura:` {res = 0 ⇔ no hay tres "O" ni hay tres "X" consecutivas en forma vertical.}
- `asegura:` {res = 3 ⇔ hay tres "X" y hay tres "O" consecutivas en forma vertical (evidenciando que beto hizo trampa).}

}

Ejercicio 13. Hospital - Atención por Guardia

Desde el Hospital Fernandez nos pidieron solucionar una serie de problemas relacionados con la información que maneja sobre los pacientes y el personal de salud. En primer lugar debemos resolver en qué orden se deben atender los pacientes que llegan a

la guardia. En enfermería, hay una primera instancia que clasifica en dos colas a los pacientes: una urgente y otra postergable (esto se llama hacer triage). A partir de dichas colas que contienen la identificación del paciente, se pide devolver una nueva cola según la siguiente especificación.

`problema orden_de_atencion` (in urgentes: cola⟨Z⟩, in postergables: cola⟨Z⟩) : cola⟨Z⟩ {
`requiere:` {No hay elementos repetidos en urgentes.}
`requiere:` {No hay elementos repetidos en postergables.}
`requiere:` {La intersección entre postergables y urgentes es vacía.}
`requiere:` {|postergables| = |urgentes|.}
`asegura:` {No hay repetidos en res.}
`asegura:` {res es permutación de la concatenación de urgentes y postergables.}
`asegura:` {Si urgentes no es vacía, en la cabeza de res hay un elemento de urgentes.}
`asegura:` {En res no hay dos seguidos de urgentes.}
`asegura:` {En res no hay dos seguidos de postergables.}
`asegura:` {Para todo c1 y c2 de tipo "urgente" pertenecientes a urgentes si c1 aparece antes que c2 en urgentes entonces c1 aparece antes que c2 en res.}
`asegura:` {Para todo c1 y c2 de tipo "postergable" pertenecientes a postergables si c1 aparece antes que c2 en postergables entonces c1 aparece antes que c2 en res.}
}

Ejercicio 14. Hospital - Alarma epidemiológica

Necesitamos detectar la aparición de posibles epidemias. Para esto contamos con un lista de enfermedades infecciosas y los registros de atención por guardia dados por una lista expedientes. Cada expediente es una tupla con ID paciente y enfermedad que motivó la atención. Debemos devolver un diccionario cuya clave son las enfermedades infecciosas y su valor es la proporción de pacientes que se atendieron por esa enfermedad. En este diccionario deben aparecer solo aquellas enfermedades infecciosas cuya proporción supere cierto umbral.

`problema alarma_epidemiologica` (in `registros` : `seq`⟨Z × `str`⟩, in `infecciosas` : `seq`⟨`str`⟩, in `umbral` : ℝ) : dict⟨str, ℝ⟩ {
`requiere:` {0 < umbral < 1.}
`asegura:` {claves de res pertenecen a infecciosas.}
`asegura:` {Para cada enfermedad perteneciente a infecciosas, si el porcentaje de pacientes que se atendieron por esa enfermedad sobre el total de registros es mayor o igual al umbral, entonces res[enfermedad] = porcentaje.}
`asegura:` {Para cada enfermedad perteneciente a infecciosas, si el porcentaje de pacientes que se atendieron por esa enfermedad sobre el total de registros es menor que el umbral, entonces enfermedad no aparece en res.}
}

Ejercicio 15. Hospital - Empleado del mes

Dado un diccionario con la cantidad de horas trabajadas por empleado, en donde la clave es el ID del empleado y el valor es una lista de las horas trabajadas por día, queremos saber quienes trabajaron más para darles un premio. Se deberá buscar la o las claves para la cual se tiene el máximo valor de cantidad total de horas, y devolverlas en una lista.

`problema empleados_del_mes` (horas:dicc⟨ℤ, *seq*⟨ℤ⟩) : *seq*⟨ℤ⟩ {
`requiere:` {No hay valores en horas que sean listas vacías.}
`asegura:` {Si ID pertence a res entonces ID pertence a las claves de horas.}
`asegura:` {Si ID pertenece a res, la suma de sus valores de horas es el máximo de la suma de elementos de horas de todos los otros IDs.}
`asegura:` {Para todo ID de claves de horas, si la suma de sus valores es el máximo de la suma de elementos de horas de los otros IDs, entonces ID pertenece a res.}
}

Ejercicio 16. Hospital - Nivel de ocupación

Queremos saber qué porcentaje de ocupación de camas hay en el hospital. El hospital se representa por una matriz en donde las filas son los pisos, y las columnas son las camas. Los valores de la matriz son Booleanos que indican si la cama está ocupada o no. Si el valor es verdadero (True) indica que la cama está ocupada.

Se nos pide programar en Python una función que devuelve una secuencia de reales, indicando la proporción de camas ocupadas en cada piso.

`problema nivel_de_ocupacion` (in camas_por_piso:*seq*⟨*seq*⟨Bool⟩⟩) : *seq*⟨ℝ⟩ {

`requiere:` {Todos los pisos tienen la misma cantidad de camas.}
`requiere:` {Hay por lo menos 1 piso en el hospital.}
`requiere:` {Hay por lo menos una cama por piso.}
`asegura:` {|res| = |camas_por_piso|.}
`asegura:` {Para todo $0 \leq i < |res|$ se cumple que res[i] es igual a la cantidad de camas ocupadas del piso i dividido el total de camas del piso i).}
}