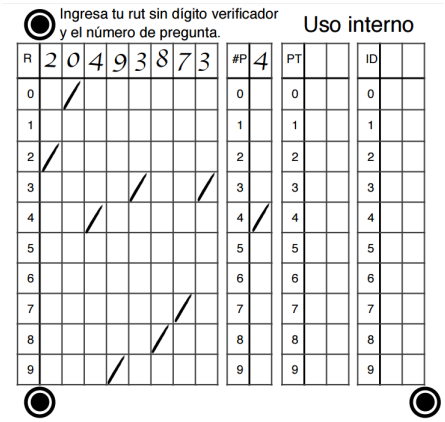
PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC1103 – Introducción a la Programación
2 - 2016

Examen

Instrucciones generales:

- La interrogación consta de cuatro preguntas con igual ponderación.
- Solo se reciben consultas sobre el enunciado en los primeros 30 minutos.
- La duración de la interrogación es de 3 horas.
- La última página es un recordatorio; no está permitido utilizar material de apoyo adicional.
- Si el enunciado de la pregunta no lo indica explícitamente, no es necesario validar los ingresos del usuario. Puedes suponer que no hay errores por ingreso de datos.
- Recuerda que esta es una instancia formal de evaluación. Tu respuesta será leída por otros por lo que debes utilizar un lenguaje apropiado. Además, no se aceptarán comentarios que no estén relacionados a la respuesta que se está respondiendo.
- Responde cada pregunta en una hoja separada, marcando tu RUT y el número de pregunta en todas las hojas prestando atención a la dirección del trazo, tal como se muestra en el siguiente ejemplo:



Pregunta 1 (1/4)

En el juego Combate Naval, cada jugador tiene un tablero de 10 x 10 casilleros, en los que se esconden barcos, que pueden ocupar 1, 2, 3, 4 o 5 casilleros consecutivos. Los barcos no pueden estar en casilleros adyacentes (siempre habrá al menos un espacio de mar entre los barcos). Los casilleros ocupados por barcos están marcados con una "X", mientras el mar se representa con un string vacío.

Escribe una función **`contar(matriz)`** que reciba una matriz de 10x10 casilleros con un tablero de Combate Naval y retorne una lista con cuántos barcos hay de tamaño 1, 2, 3, 4 y 5.

Por ejemplo, para la siguiente matriz:

	0	1	2	3	4	5	6	7	8	9
0										X
1		X	X	X	X					
2										
3	X							X	X	X
4	X				X					
5					X					
6					X					
7	X				X		X		X	
8	X									
9				X	X	X	X	X		

La función debe retornar `[3,2,1,2,1]`

Pregunta 2 (1/4)

Un amigo tuyo que es biólogo te ha pedido ayuda para crear un programa para estudiar el comportamiento de un ecosistema. Ya tienes terminada la clase Ecosistema y sus métodos, los cuales aparecen descritos más adelante. Ahora debes implementar la clase Animal de forma que pueda ser utilizada por el método **`interactuar`** de la clase **`Ecosistema`**.

1. (10 pts) Crea la clase **`Animal`** con su método **`__init__`**, el cual debe recibir como parámetros la posición x, y del animal en el terreno, el tipo de animal que es un string que puede tomar los valores "Zorro" o "Conejo" (no debes validarlo), e inicializar el atributo edad en cero.

2. (5 pts) Crea el método **`envejecer`**, que incrementa la edad del animal en 1.

3. (15 pts) Crea el método **`cazar`**, que recibe como parámetro una lista de objetos de la clase `Animal`, y que solamente cuando el animal es de tipo "Zorro", debe revisar si en la lista hay un "Conejo": si lo hay, el zorro lo caza con una probabilidad del 30 %. Debes retornar la posición de la presa en la lista (el zorro puede cazar un conejo cada vez) o -1 si el zorro no pudo cazar un conejo

4. (30 pts) Crea el método **`reproducirse`**, que recibe como parámetro una lista de objetos de la clase `Animal` y debe revisarla para determinar si hay un animal del mismo tipo en ella. Si ambos animales tienen una edad mayor que cero: si el animal es un Zorro se reproduce con un 10 % de probabilidad, mientras que si el animal es un Conejo se reproduce con un 100 % de probabilidad. Cada animal tiene un único hijo, por lo que tu método debe retornar True o False para indicar el resultado.

A continuación se muestra el código de la clase `Ecosistema`:

```
class Ecosistema:
    def __init__(self):
        self.terreno=[]
        for i in range(8):
            self.terreno.append([0]*8)

    def interactuar(self):
        i=0
        while i < len(self.terreno):
            j=0
            while j < len(self.terreno):
                animal=self.terreno[i][j]
                if animal != 0:
                    vecinos=self.encontrar_animales_alrededor(i,j)
                    espacios_vacios=self.encontrar_espacios_alrededor(i,j)
                    animal.envejecer()
                    presa = animal.cazar(vecinos)
                    if presa != -1:
                        p=vecinos[presa]
                        self.terreno[p.x][p.y] = 0
                    if len(espacios_vacios) > 0 and len(vecinos) > 0:
                        eleccion = random.randint(0,len(espacios_vacios)-1)
                        espacios = espacios_vacios.pop(eleccion)
                        if animal.reproducirse(vecinos) == True:
                            hijo = Animal(espacio[0],espacio[1],animal.tipo)
                            self.terreno[hijo.x][hijo.y] = hijo
                    if len(espacios_vacios) > 0:
                        eleccion = random.randint(0,len(espacios_vacios)-1)
                        espacio = espacios_vacios.pop(eleccion)
                        self.terreno[animal.x][animal.y] = 0
                        animal.x = espacio[0]
                        animal.y = espacio[1]
                        self.terreno[animal.x][animal.y] = animal
                j = j+1
            i = i+1
```

Pregunta 3 (1/4)

Se desean obtener estadísticas de palabras más comunes utilizadas por los usuarios de Facebook. Para ello, se ha extraído un listado de palabras de más de 5.000.000 de usuarios en Chile. Se te pide implementar las siguientes funciones:

- **`eliminar_palabras_cortas(listas_palabras, n))`**:
  Recibe un listado de palabras **`lista_palabras`** y un número **`n`**, y retorna una lista con todas las palabras del listado original que contienen más de **`n`** letras.

  Por ejemplo, si **`n`** es 20, podría retornar la siguiente lista con palabras de más de 20 letras:
  `[‘desproporcionadamente’, ‘interdisciplinariedad’, ‘anticonstitucionalidad’]`

- **`palabras_mas_repetidas(lista_palabras, freq)`**:
  Recibe un listado de palabras **lista_palabras** y una frecuencia mínima **`freq`**, y retorna una lista, cuyos elementos son a su vez listas que contienen dos elementos: una palabra y la cantidad de veces que aparece en la lista.

  Por ejemplo, si **`freq`** es 1.000.000, podría retornar la siguiente lista de palabras que se repite más de 1.000.000 veces:
  `[[‘carrete’, 1234567], [‘estudio’, 2481632], [‘programar’, 8246810], [‘dormir’, 3000111]]`

- **`histograma(lista_palabras_freq)`**:
  Recibe una lista de listas de palabras y sus frecuencias respectivas **`lista_palabras_freq`** (como la ilustrada anteriormente), y muestra en pantalla una especie de histograma con un **`‘*’`** en el histograma por cada millón de veces que se repite la palabra. Por ejemplo, para la lista anterior, desplegaría lo siguiente:

  `- carrete`
  `- estudio`
  `- programar`
  `- dormir`

A modo ilustrativo, el programa que se desea construir usará dichas funciones de la siguiente manera:

```
lista_palabras_largas = eliminar_palabras_cortas(lista_palabras,10)
lista_palabras_frecuentes = palabras_mas_repetidas(lista_palabras,1000000)
histograma(lista_palabras_frecuentes)
```

Pregunta 4 (1/4)

Para preparar el enunciado de un examen, los profesores cuentan con una lista de problemas. Cada problema es una lista `[nombre, tipo, dificultad]`. Donde `nombre` identifica a la pregunta. El tipo indica qué contenido se está evaluando; solo hay 3 tipos: Objetos (`O`), Listas (`L`) y Recursión (`R`). La dificultad es un número que va de 0 a 100, donde 0 es un problema dificultad baja y 100 alta.

En esta pregunta se te pide escribir una función **recursiva `generar`** que permita generar una propuesta de examen válida a partir de una lista de problemas y límites superior inferior (`inf`) y (`sup`) e de dificultad total. El examen tendrá 3 problemas, uno de cada tipo (`O`, `L`, y `R`). Para que una propuesta de examen sea válida, la suma de la dificultad de los 3 problemas de la propuesta deben ser más grande o igual que `inf` y menor o igual que `sup`. En caso de no encontrar una solución, la función debe retornar una lista vacía.

Nota: Tú decides qué parámetros recibe esta función recursiva.
A modo de ejemplo, considera la siguiente lista de problemas y límites inferior y superior:

```
problemas = [["Don Yandran", "O",90], ["Substrings", "L",75],
             ["Castellers", "O",70], ["ADN", "L",30],
             ["UCarpool", "O",50], ["Examenes", "R",10],
             ["Viajes", "R",60], ["Diofanticas", "R",70],
             ["Baile Chino", "L",30], ["Monumentos", "O",40],
             ["Robot", "L",30],

inf = 210
sup = 230
```

La función **`generar`** podría retornar:

```
[['Don Yadran', 'O', 90], ['Substrings','L',75],['Viajes','R',60]]
```

Recordatorio contenidos examen - Segundo semestre 2016

En los ejemplos, lo marcado como `<texto>` se interpreta como lo que debes rellenar con tu código según cada caso.

1. Tipos de datos y operadores

Tipo de dato	Clase	Ejemplo
Números enteros	`int`	`2`
Números reales	`float`	`2.5`
Números complejos	`complex`	`2 + 3j`
Valores booleanos	`bool`	`True/False`
Cadenas de texto	`str`	`"hola"`

Operación	Descripción	Ejemplo
`+`	Suma	`2.3+5.4`
`-`	Resta	`45.45-10.02`
`-`	Negación	`-5.4`
`*`	Multiplicación	`(2.3+4.2j)*3`
`**`	Potenciación	`2**8`
`/`	División	`100/99`
`//`	División entera	`100//99`
`%`	Módulo	`10%3`

Prioridad (de mayor a menor): `()`; `**`; `*`, `/`, `//` o `%`; + o − .

Operación	Descripción	Ejemplo
`==` (`!=`)	Igual (distinto) a	`2==2`
`<` (`<=`)	Menor (o igual)	`1<1.1`
`>` (`>=`)	Mayor (o igual)	`3>=1`
`and`	Ambos `True`	`2>1 and 2<3`
`or`	Algún `True`	`2!=2 or 2==2`
`not`	Negación	`not True`

Prioridad (de mayor a menor): `()`; `or`; `and`; `not`; comparadores.

2. Funciones predefinidas

- `int(arg)` convierte `arg` a entero.
- `float(arg)` convierte `arg` a número real.
- `str(arg)` convierte `arg` a cadena de texto (string).
- `list(arg)` genera una lista con elementos según `arg`, que debe ser *iterable* (strings, listas, tuplas, `range`).

3. Función `print`

- Un argumento:
  `print(arg)`
- Dos o más argumentos:
  `print(arg1, arg2,arg3)`
- Uso de parámetros, por ejemplo, para eliminar salto de línea y separar con guión:
  `print(arg, sep='-', end='')`

4. Función `input`

- `ret = input(texto)` guarda en `ret` un `str` ingresado.
- `ret = int(input(texto))` guarda en `ret` un `int` ingresado.
- `ret = float(input(texto))` guarda en `ret` un `float` ingresado.

5. `if/elif/else`

```
if <cond 1> :
<codigo si se cumple cond 1>
if <cond 1.1> :
<codigo si se cumple 1.1>
else :
<codigo si no se cumple 1.1>
elif <cond 2> :
<codigo si se cumple cond 2 pero no cond 1>
else :
<codigo si no se cumple cond 1 ni cond 2>
```

6. `while`

```
while <condicion> :
<codigo que se ejecuta repetidas
veces mientras se cumpla
condicion>
```

7. Funciones propias

```
def funcion(<argumentos>):
<codigo de funcion>
return <valor de retorno>
```

Variables y parámetros definidos dentro de funciones no son visibles fuera de la función (*scope* local).

8. Programación orientada a objetos

```
class <NombreClase>:
def __init__(self, <parametros>):
self.<atributos> = <algun parametro>
def __str__(self):
<codigo sobrecarga de funcion str()>
return <string que representa
los atributos>
def <metodo propio>(self, <parametros>):
<codigo de modulo propio>
```

9. Strings, clase `str`

Acceso a caracteres particulares con operador `[]`, partiendo con índice cero. Porción de string con *slice*, por ejemplo si `string='Hola'`, `string[1:3]` es `'ol'`. Algunos métodos y funciones de strings:

- Operador +: une (concatena) dos strings.
- Operador `in`: cuando `a in b` retorna `True`, entonces el string `a` está contenido en el string `b`.
- `string.find(a)`: determina si `a` está contenido en `string`. Retorna la posición (índice) dentro de `string` donde comienza la primera aparición del sub-string `a`. Si no está, retorna -1.
- `string.upper()`, `string.lower()`: retorna `string` convertido a mayúsculas y minúsculas, respectivamente.
- `string.strip()`: retorna un nuevo string en que se eliminan los espacios en blanco iniciales y finales de `string`.
- `string.split(a)`: retorna una lista con los elementos del `string` que están separados por el string `a`. Si se omite `a`, asume que el separador es uno o más espacios en blanco o el salto de línea.
- `p.join(lista)`: suponiendo que `p` es un string, retorna un nuevo string conteniendo los elementos de la lista "unidos" por el string `p`.
- Función `len(string)`: entrega el número de caracteres de `string`.

Una forma de iterar sobre los caracteres de `string`:

```
for char in string:
<operaciones con el caracter char>
```

10. Listas

Secuencias de elementos-objetos. Se definen como `lista = [<elem1>,<elem2>,...,<elemN>]`. Los elementos pueden o no ser del mismo tipo. Para acceder al elemento `i`, se usa `lista[i]`. La sublista `lista[i:j]` incluye los elementos desde la posición `i` hasta `j-1`. Algunos métodos y funciones de listas:

- Operador +. concatena dos listas.
- Operador `in`: `a in b` retorna `True` cuando el elemento `a` está contenido en la lista `b`. Si no está contenido, retorna `False`.
- `lista.append(a)`: agrega `a` al final de la lista.
- `lista.insert(i,a)`: inserta el elemento `a` en la posición `i`, desplazando los elementos después de `i`.
- `lista.pop(i)`: retorna el elemento de la lista en la posición `i`, y lo elimina de `lista`.
- `lista.remove(elem)`: elimina la primera aparición de `elem` en la lista.
- Función `len(lista)`: entrega el número de elementos de `lista`.

Para iterar sobre los elementos de `lista`:

```
for elem in lista:
<lo que quieran hacer con elem>
```

11. Archivos

Abrir un archivo:
`archivo = open(<nombre archivo>,<modo>)`, p. ej. `archivo = open('archivo.txt','r')`. `<modo>` puede ser `'w'` para escribir un archivo nuevo, `'r'` para leer un archivo (predeterminado), y `'a'` para escribir en un archivo ya existente, agregando datos al final del archivo.

Algunos métodos del objeto que retorna la función `open`:

- `archivo.readline()`: retorna un string con la línea siguiente del archivo, comenzando al inicio del archivo.
- `archivo.write(string)`: escribe en el archivo el string `string`.
- `archivo.close()`: cierra el archivo.

Para leer un archivo entero puedes usar `for`, que iterará línea por línea del archivo:

```
archivo = open('archivo.txt','r')
for linea in archivo:
<lo que quieran hacer con linea>
```

12. Búsqueda y ordenamiento

Búsqueda secuencial o lineal Busca secuencialmente un elemento dentro de una lista de tamaño n hasta encontrarlo. Peor caso: elemento no está en lista, n comparaciones. Uso: lista desordenada.

Búsqueda binaria Supone lista ordenada. Divide la lista en sublistas dependiendo del valor del elemento buscado: si el elemento es mayor que el elemento medio de la lista, se sigue por la sublista derecha; si no, por la lista izquierda. Peor caso: para una lista de n elementos, se realizan $log_2(n)$ comparaciones.

Ordenamiento por selección Busca el índice del elemento más pequeño de la lista, e intercambia los valores entre el índice encontrado y el primer elemento; luego, encuentra el segundo elemento más pequeño, y lo intercambia con el elemento de la segunda posición, realizando la misma operación para el tercero, cuarto, etc..

Ordenamiento por inserción Itera por los elementos de la lista, partiendo por el primer elemento, y generando una lista ordenada con los elementos mientras itera. Es similar a cómo una persona ordena una lista de cartas.

Métodos de Python para ordenamiento

- `lista.sort()`: ordena `lista` en forma ascendente.

13. Recursión y backtracking

Elementos de una función recursiva:

- Caso base: para terminar la recursión.
- Llamada recursiva: llamada a la misma función dentrl de la función, con parámetros distintos que hacen disminuir el problema original.

Para programar una función recursiva, descomponer la función en elementos que puedan ser llamados con subconjuntos de datos. Ejemplo: suma de los primeros N números

```
def suma(N):
if N == 1: # caso base
return 1:
else: #descomponer en N y restantes N-1
return N + suma(N-1)
```

Backtracking: cuando hay muchas formas o caminos de buscar una solución. Se puede formar un arbol con las soluciones, y el algoritmo comienza en el nodo raíz. Cada camino dentro del árbol es un código recursivo, y dentro del código se exploran las ramas del árbol. Ejemplo: encontrar la suma de todas las formas de combinar una lista de números:

```
def suma_conm_lista(lista, sublista=[], suma=0):

# Caso base: sublista tiene N elementos,
# o la lista tiene 0 elementos
if len(lista) == 0:
print sublista, suma
return
else:
# Quito un elemento de la lista y lo
# agrego a la sublista para llamar
# a la función en forma recursiva,
# y aumento la suma con el
# elemento que saqué
for i in range(len(lista)):
suma_conm_lista( lista[:i]+lista[i+1:],\
sublista+[lista[i]], suma+lista[i])
```