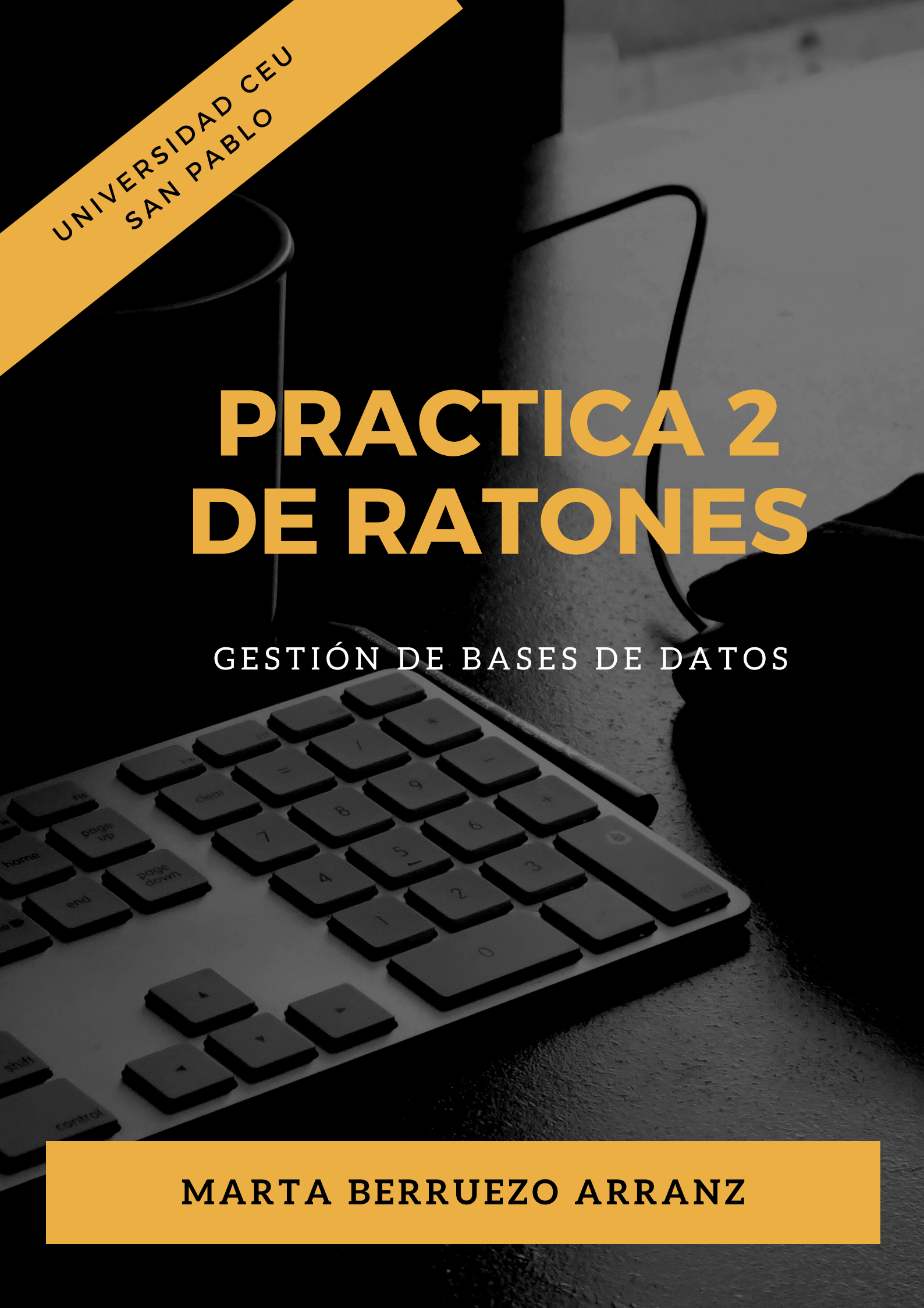

UNIVERSIDAD CEU SAN PABLO

# PRACTICA 2 DE RATONES

GESTIÓN DE BASES DE DATOS

MARTA BERRUEZO ARRANZ

# *ANALISIS Y DESCRIPCION DE LA APLICACIÓN*

Con respecto al análisis y descripción de la aplicación, lo voy a hacer siguiendo el orden en el que hice los ejercicios

En primer lugar, abrimos el conector en un archivo.py llamada *persistence*, donde indicamos el usuario, la contraseña y la base de datos donde vamos a querer trabajar, en la que tiene que estar incorporada el DDL
Lo primero que hice fue **EL EJERCICIO 3,** crear una población vacía. Para ello creo una función llamada *insertPopulation* en la que recibo una población, abro el cursor, pongo el DML, inserto los valores de Python en la memoria y hago un commit para guardar los cambios en memoria.

```
def insertPopulation(population):
    """
    Esta función inserta una poblacion en memoria
    Parámetros:
        -poblacion: poblacion que se quiere insertar en memoria
    """
    cursor = cnx.cursor()

    insert_into_population = "INSERT INTO population(name,researcher,start_date,num_d
ays) VALUES (%s,%s,%s,%s)"
    start_date = str(population.get_start_date())
    values_population = (population.get_name(), population.get_researcher(), start_da
te, population.get_num_days())
    cursor.execute(insert_into_population, values_population )
    cnx.commit()
    cursor.close()
```

Mi primer problema vino a raíz de las referencias, que ha sido una de las cosas en las que mas tiempo he empleado durante toda la práctica. Para solucionarlo, me creo otra función:

```
def getIdNewPopulation():
    """
    Esta función devuelve la referencia de la ultima poblacion.
    """
    cursor = cnx.cursor()

    cursor.execute("SELECT max(reference) FROM population")
    reference = 1
    record = cursor.fetchone()
    if record[0]:
        reference = int(record[0]) + 1
    cursor.close()
    return reference
```

En esa función estoy cogiendo la referencia de la última población en memoria sumándole 1 para la nueva población e incorporarla en Python. Llamando a esta función en lab.py, lo que hago es que cuando creo la nueva población, le paso esta referencia, y así, tanto Python como la memoria tendrán las mismas referencias de poblaciones.

Lo siguiente que hice fue **EL EJERCICIO 5,** insertar un ratón en memoria. Para ello, me creo una función llamada *insertMouse,* parecida a la de la población, en la que mediante DML, incorporándolo a Python, le paso los datos que introduce Python como atributos. Este ejercicio fue un poco más laborioso debido a que no sabía que el gender, chromosome 1 y chromosome

2 los tenía que pasar a string para que SQL lo interpretara correctamente; y por último realizo un commit.

**EL EJERCICIO 6,** fue igual que el 5, insertar un ratón pero unicamente indicando el nombre, por lo que como los demás datos los da Python, se meten igual en la función *insertMouse*.

```
def insertMouse(population, mouse, commit = True):
    """
    Esta función inserta un raton en memoria, según la poblacion en la que este
    Parámetros:
        -poblacion: poblacion que se quiere insertar el raton en memoria
        -mouse: raton que se quiere insertar
        -commit: por defecto true
    """
    cursor = cnx.cursor()

    insert_into_mouse = "INSERT INTO mouse(name,birthdate,weight,gender,temperature,d
escription,chromosome1,chromosome2,id_population) VALUES (%s, %s, %s, %s, %s, %s, %s,
 %s, %s)"
    birthdate = str(mouse.get_birthdate())
    id_population = population.get_reference()
    gender_mouse = str(mouse.get_gender())
    chromosome1 = str(mouse.get_chromosome1())
    chromosome2 = str(mouse.get_chromosome2())
    values_mouse = ( mouse.get_name(), birthdate, mouse.get_weight(), gender_mouse ,
mouse.get_temperature(), mouse.get_description(), chromosome1, chromosome2, id_popula
tion)
    cursor.execute(insert_into_mouse, values_mouse )
    if (commit):
        cnx.commit()
    cursor.close()
```

Tuve el mismo problema que con respecto a las referencias, que lo solucioné de la misma forma, creándome una función que coja la última referencia de memoria, y que, a partir de ahí, lo llamo en el lab.py, meto la referencia en la función read_mouse_from_keyboard, y ahí es donde introduzco la referencia del último ratón + 1 al crear el ratón.

```
def getIdNewMouse():
    """
    Esta función devuelve la referencia del último ratón.
    """
    cursor = cnx.cursor()

    cursor.execute("SELECT max(reference) FROM mouse")
    reference = 1
    record = cursor.fetchone()
    if record[0]:
        reference = int(record[0]) + 1
    cursor.close()
    return reference
```

**El EJERCICIO 2** no me causó mucha complicación al principio, debido a que era una mezcla entre el ejercicio 3 y el 5: insertas una población, y posteriormente insertas a los ratones. Se hace un commit a no ser que encuentre un problema, que con la excepción se hace un rollback, no guardando los cambios en memoria. Lo que fue un poco más complejo de este ejercicio, donde dedique bastantes horas, fue que a la hora de pasarlo a lab.py, cuando se crea la población se pasa la referencia, pero luego creo un parámetro nuevo llamado ref_mouse, donde le paso la referencia del ratón, que lo recibe la función generate_random_mouses como parámetro y ahí es donde modifico el código que nos dio Alberto, donde pongo que para la primera referencia,

coja la que yo le doy, pero las demás las haga automáticas después del número dado, tal y como lo hace el programa cuando no le doy referencias. Tuve algunas dudas con este cambio, pero tras una tutoría, Alberto me dio el visto bueno, ya que funcionaba correctamente.

```
def insertPopulationWithSize(population):
    """
    Esta función inserta una poblacion en memoria
    Parámetros:
        -poblacion: poblacion que se quiere insertar en memoria
    """
    try:
        insertPopulation(population)
        for mouse in (population.get_animal_list()):
            insertMouse(population, mouse)
        cnx.commit()
    except Exception:
        cnx.rollback()
```

**El EJERCICIO 4** no necesito realizar ninguna modificación en el código ni crear funciones, debido a que únicamente para seleccionar datos de la base de datos de las referencias de todos los ratones no necesitamos ninguna interacción con Python, por lo que no implementamos nada.

**EL EJERCICIO 7,** eliminar un ratón de la población, fue mas o menos sencillo, pero al principio me costó que me funcionara, debido a que si no ponía que el id_population = %s, no me ejecutaba bien. Ese fue el problema más significativo que tuve, lo demás bastante sencillo. Le paso una población y la referencia del ratón que queremos eliminar y que se encuentra en el lab.py, y lo eliminamos con el código de DML. Hacemos commit y cerramos el cursor

```
def deleteMouse(population, reference_mouse):
    """
    Esta función elimina un raton en memoria
    Parámetros:
        -
poblacion: poblacion donde se encuentra el raton que se quiere eliminar en memoria
        -reference_mouse: referencia del raton que se quiere eliminar
    """
    cursor = cnx.cursor()

    delete_mouse = "DELETE FROM mouse WHERE reference = %s and id_population = %s"
    id_population = population.get_reference()
    cursor.execute(delete_mouse, (reference_mouse, id_population))

    cnx.commit()
    cursor.close()
```

**EL EJERCICIO 8,** modificar un ratón, también me pareció bastante sencillo. Simplemente ponemos el DML del update, seleccionando los datos del ratón que si se pueden cambiar, los pasamos a la función mediante parámetros, y realizamos el commit para guardar los cambios.

```
def updateMouse(population, reference, name, weight, temperature, description):

    """
    Esta función modifica un raton en memoria
    Parámetros:
        -
poblacion: poblacion donde se encuentra el raton que se quiere modificar y que se enc
uentra en memoria
        -reference: referencia del raton que se quiere modificar
        -name: el nuevo nombre modificado
        -weight: el nuevo peso modificado
        -temperature: la nueva temperatura modificada
        -description: la nueva descripcion modificada
    """
    cursor = cnx.cursor()

    update_mouse = "UPDATE mouse SET name = %s, weight=%s,temperature=%s, description
=%s WHERE reference = %s"
    value_mouse = (name, weight, temperature, description, reference)
    cursor.execute(update_mouse, value_mouse)
    cnx.commit()
    cursor.close()
```

Con **EL EJERCICIO 9,** pasa lo mismo que con el ejercicio 4; como simplemente es seleccionar datos de los detalles de un ratón, no hace falta realizar ninguna implementación en Python, por lo que no hace falta realizar cambios en el código ni hacer una función en el persistence.

En **EL EJERCICIO 10**, crear familias, sí que es verdad que le dediqué bastante tiempo, pero al final lo conseguí. Lo que me resulto complicado fue estructurar lo que tenía que hacer, ya que la forma en la que se crean las familias en Python no es la misma que en la base de datos. Tuve que hacer 4 funciones, 3 para insertar las familias en memoria y 1 para juntar esas tres funciones. En primer lugar, voy a explicar la función que crea una familia. Recibe una familia (un objeto), y mediante el DML crea la familia con los valores que se crean de Python. El update fue lo último que hice de toda la práctica, me costó un poco entenderlo; básicamente en esta función los machos que crean familias se eliminan de la población mediante un update, porque en Python ya se incorpora. Tuve problemas porque había puesto que id_population no podía ser null, pero tras cambiarlo, me funcionó perfecto.

```
def createFamily(population, family, commit = True):
    """
    Esta función inserta una familia en memoria
    Parámetros:
        -poblacion: poblacion donde se quiere insertar la familia en memoria
        -family: familia que se quiere insertar
        -commit: por defecto true
    """
    cursor = cnx.cursor()
    create_family = "INSERT INTO family(id,id_father,id_population) VALUES(%s,%s,%s)”
    id_family = family.get_reference()
    mouse_father = family.get_parent()
    reference_father = mouse_father.get_reference()
    id_population = population.get_reference()
    values_family = (id_family, reference_father, id_population)
    cursor.execute(create_family, values_family)


    update_mouses = "UPDATE mouse SET id_population = NULL WHERE referenc
e = (%s)"
    values_father = (reference_father,)
    cursor.execute(update_mouses,values_father)
```

Hacemos lo mismo con la familia normal, recibimos un parámetro, lo insertamos en memoria con la información que nos suministra de Python pero modificada para que la base de datos "la entienda" y por último hacemos el update para eliminar a la hembra de la población, porque ya se encuentra en la familia. En este momento de la práctica, solucione uno de los problemas de la práctica, que era introducir el método get_mother() en NormalFamily.py, ya que lo necesitaba para acceder a la hembra.

```
def createNormalFamily(family, commit = True):
    """
    Esta función inserta una familia normal en memoria
    Parámetros:
        -family: familia normal que se quiere insertar
        -commit: por defecto true
    """
    cursor = cnx.cursor()

    create_normal_family = "INSERT INTO normal_family(id,id_normal_mother) VALUES (%s
,%s)"
    mouse_mother = family.get_mother()
    reference_mother = mouse_mother.get_reference()
    values_normal_family = (family.get_reference(), reference_mother)
    cursor.execute(create_normal_family, values_normal_family)

    update_mouses = "UPDATE mouse SET id_population = NULL WHERE reference = (%s)"
    values_mother = (reference_mother,)
    cursor.execute(update_mouses,values_mother)
```

Ocurre similar con las familias poligámicas, recibe una familia e inserta una familia en la base de datos con la información obtenida de python, que en este caso es la referencia de la familia. Lo único que se diferencia en relación con las anteriores familias, es que en este caso no solo hay una hembra, sino que hay varias, por lo que hay que hacer un for e ir hembra por hembra, para updatear el atributo id_polygamous_mother. También realizamos el update para eliminar la el id de la población, como habíamos hecho con las otras familias

```
def createPolygamicFamily(family, commit = True):
    """
    Esta función inserta una familia poligamica en memoria
    Parámetros:
        -family: familia poligamica que se quiere insertar
        -commit: por defecto true
    """
    cursor = cnx.cursor()

    create_polygamic_family = "INSERT INTO polygamous_family (id) VALUES (%s)"
    values_polygamic_family = (family.get_reference(), )
    cursor.execute(create_polygamic_family, values_polygamic_family)

    create_mother = "UPDATE mouse SET id_polygamous_mother=%s WHERE reference = %s"
    for mother in family.get_mothers():
        reference_mother = mother.get_reference()
        values_polygamic_mother = (family.get_reference(), reference_mother)
        cursor.execute(create_mother, values_polygamic_mother)

        update_mouses = "UPDATE mouse SET id_population = NULL WHERE reference = (%s)
"
        values_mother = (reference_mother,)
        cursor.execute(update_mouses,values_mother)
```

Por último, juntamos las tres funciones para que cree una familia si o si, y luego si la familia es normal, que entre en la función para insertar una familia normal en la base de datos, y si es una familia poligámica, que lo inserte en la tabla de familia poligámica.
No hemos hecho el commit en ninguna función anterior, ya que, si algo falla, no se crea ninguna familia, va todo como en pack. Si todo funciona correctamente se realiza un commit, pero si en alguna de las funciones intercepta un error, se realiza un rollback.

```
def insertFamily(population):
    """
    Esta función inserta una familia, que puede ser una familia normal y una familia
poligamica en memoria
    Parámetros:
        -poblacion: poblacion donde se quiere insertar la familia en memoria
    """
    try:
        for family in  population.get_families_list():
            createFamily(population, family)
            #print(str(family))

            if isinstance(family, NormalFamily.NormalFamily):
                createNormalFamily(family)
            elif isinstance(family, PoligamicFamily.PoligamicFamily):
                createPolygamicFamily(family)
        cnx.commit()
    except Exception as e:
        print(e)
        cnx.rollback()
```

En este ejercicio, como en los ejercicios anteriores, tuve el gran problema de las referencias, que sin duda ha sido lo más pesado de la práctica. Para solucionarlo, hago lo mismo que en mouse y población: me creo una función que coga la última referencia de la base de datos y le sumo dos. Por lo tanto, al lab.py le paso la referencia, y la introduzco en el método que tienen las familias de family_creation y ahí, lo pongo como un atributo que te dan de tanto la familia normal como la poligámica. De esta forma cada vez que cree una familia, mirará la última referencia y le sumará 1.
En esta parte, que estuve bastante tiempo, encontré otro de los problemas que tenía la práctica, que era que no se podía acceder a la referencia de las familias y ponerlas tanto en las familias normales como en las poligámicas, ya que eran atributos privados. Lo que tuve que hacer es cambiar el código de Alberto, de poner self. __reference a poner self._reference.

```
def getIdNewFamily():
    """
    Esta función devuelve la referencia de la ultima familia.
    """
    cursor = cnx.cursor()

    cursor.execute("SELECT max(id) FROM family")
    reference = 1
    record = cursor.fetchone()
    if record[0]:
        reference = int(record[0]) + 1
    cursor.close()
    return reference
```

En **EL EJERCICIO 11,** insertar los hijos de una familia a una población, fue más sencillo que las familias. Al principio pensé en hacerlo directamente en una función, inserto un ratón como habíamos hecho en el ejercicio 5 pero le pongo is_son en el insert, y claro, ahí se me ocurrió que porque no llamar a la función insertMouse y luego modificaba el ratón añadiéndole el id_mouse, para poder reutilizar el código y que no se repitiese. Es por eso por lo que mi función *reproduceFamilies* recibe una población, recorre todas las familias que se han creado anteriormente y va de hijo en hijo; llama a la función *insertMouse* y luego a una función nueva *créateChildren*. Si se ha realizado todo sin ningún fallo, se realiza un commit, si ha habido algún error, se realiza un rollback.

```
def reproduceFamilies(population):
    """
    Esta función crea hijos en memoria.
        -poblacion: poblacion donde se quiere crear hijos en memoria.
    """
    try:
        for family in population.get_families_list():
            for children in family.get_children():
                insertMouse(population, children)
                createChildren(population, children, family)
        cnx.commit()
    except Exception:
        cnx.rollback()
```

Esta función recibe el ratón hijo, y la familia a la que pertenece el ratón. Se realiza un update para modificar el ratón que se ha creado en la otra función e incorporarle el id_son con la familia escogida.

```
def createChildren(population, mouse, family, commit = True):
    """
    Esta función modifica ratones en memoria para que sean hijos.
    Parámetros:
        -
poblacion: poblacion donde se quiere modificar los ratones en memoria para que sean h
ijos
        -mouse: raton que se quiere modificar para que sea hijo
        -family: familia donde se encuentra el raton
        -commit: por defecto true
    """
    cursor = cnx.cursor()

    create_mother = "UPDATE mouse SET id_son = (%s) WHERE reference = (%s)"
    reference_family = family.get_reference()
    reference_mouse = mouse.get_reference()
    values_children = (reference_family, reference_mouse)
    cursor.execute(create_mother, values_children)
```

Llegamos al **EJERCICIO 1,** el más complicado y largo de toda la práctica, en el que debías tener todo hecho para que tuviese sentido. Fue sin duda al que más tiempo le dediqué, y donde me di cuenta de fallos anteriores como por ejemplo el introducir el update cunado creo familias para eliminar a los machos y las hembras de la población. Lo primero que hice, fue quitar todo el código de los csv`s, porque ahora Python va a leer de una base de datos. Leí el código de Alberto, y vi lo que hacía, le pedía al usuario un nombre de un csv. En mi caso tenía que pedir la

referencia de una población para poder abrirla, por lo que me creo una función llamada selectPopulations en la que selecciono todas las referencias de las poblaciones, las paso a int y las returneo, para que en lab.py se reciban y se puedan introducir en la pregunta. Una vez el usuario ha introducido el número, lo meto en la función selectOnePopulation, que es donde, después de seleccionar todo, se creara la población. Pero esa función es la última que tengo que crear, porque primero tengo que crear todos los ratones, las familias y los hijos.

En primer lugar, creo la función selectOneMouse, que al principio todo ese código lo utilizaba en otra función para crear muchos ratones, pero si quiero solo crear un ratón (como es el caso de las familias) necesito una función que me lo cree, y asi reutilizo código. Por lo tanto, en esta función recibo la referencia del ratón, selecciono todos los atributos del ratón y los voy modificando para que Python los entienda; pasándolos a int, a datetime, a un Enum… finalmente, creo el ratón y lo returneo.

```
def selectOneMouse(reference_mouse):
    """
    Esta función selecciona de memoria un raton y lo crea, a partir de una referencia
    Parametros:
        -reference_mouse: referencia de un raton
    Devuelve:
        -un raton
    """
    cursor = cnx.cursor()
    mouse = "SELECT * FROM mouse WHERE reference = %s"
    value_value = (reference_mouse,)
    cursor.execute(mouse, value_value)
    for information in cursor.fetchall():
        reference = int(information[0])
        if (reference <= 0):
            raise ValueError("The references of the mouses should be > 0")
        name = str(information[1])
        birthdate = datetime.strptime(str(information[2]),'%Y-%m-%d').date()
        weight = float(information[3])
        if (weight <= 0):
            raise ValueError("The weight of the mouses should be > 0")
        try:
            gender = EnumsMouse.Gender.from_str(str(information[4]))
        except NotImplementedError as nie:
            print(str(nie))
            raise ValueError("The gender should be Male or female")
        temperature = float(information[5])
        if (temperature <= 0):
            raise ValueError("The temperature of the mouses should be > 0")
        description = str(information[6])
        try:
            chromosome1 = EnumsMouse.Chromosome.from_str(information[7])
        except NotImplementedError:
            raise ValueError("The chromosome should be X, Y, X_MUTATED or Y_MUTATED")
        try:
            chromosome2 = EnumsMouse.Chromosome.from_str(information[8])
        except NotImplementedError:
            raise ValueError("The chromosome should be X, Y, X_MUTATED or Y_MUTATED")

    mouse = Mouse.Mouse(reference = reference, name = name, birthdate = birthdate, we
ight = weight, gender = gender, temperature = temperature, description = description,
 chromosome1 = chromosome1, chromosome2 = chromosome2 )
    cursor.close()
    return mouse
```

Para crear todos los ratones, recibo la referencia de la población que es la que habíamos adquirido gracias a la primera función de este ejercicio, y simplemente selecciono todas las referencias de ratones que se encuentren en la población x, voy creando ratones con la función anterior, los meto en una lista, y returneo esa lista.

```
def selectAllMousesOfPopulation(reference_population):
    """
    Esta función selecciona todos los ratones de una poblacion a partir de la referen
cia de la poblacion
    Parametros:
        -reference_population: la referencia de una poblacion
    Devuelve:
        -una lista de ratones de una poblacion
    """
    cursor = cnx.cursor()

    selectMouse = "SELECT reference FROM mouse WHERE id_population = %s"
    value = (reference_population,)
    cursor.execute(selectMouse, value)
    mouses_list = []
    for reference in cursor.fetchall():
        mouse = selectOneMouse(reference[0])
        mouses_list.append(mouse)
    return mouses_list
```

Ahora pasamos a crear familias, que fue de las cosas más complicadas de este ejercicio y que más tiempo estuve pensando. Como hay dos tipos de familias, pero necesito una lista de familias para poder introducirlo en la población, tendré 2 funciones que crearan familias de cada tipo y una que juntara esas dos funciones.
En esta función, selecciono el id de la familia haciendo un inner join con la familia normal, y una vez lo incorporo a Python, ya puedo seleccionar el id del padre, CREAR UN RATON con la función que habíamos creado (por lo tanto, estaríamos reutilizando el código), seleccionar el id de la madre y hacer lo mismo, y seleccionar los hijos e ir creándolos en Python uno a uno. Pensé en juntar madre, padre e hijos y luego ir creando, pero como con hijos necesito hacer un for, me parecía más lioso asiqué lo hice uno por uno. Finalmente creo la familia, ya que tengo el ratón padre, el ratón madre, los ratoncillos hijos, y la referencia de la familia.

```
def selectNormalFamily(reference_population):
    """
    Esta función selecciona de memoria una familia normal y la crea, a partir de la r
eferencia de la poblacion
    Parametros:
        -reference_population: es la referencia de la poblacion
    Devuelve:
        -una lista con familias normales
    """
    cursor = cnx.cursor()
    select_normal_family = "SELECT nf.id FROM family as f\
                            INNER JOIN normal_family as nf\
                            on f.id = nf.id\
                            WHERE f.id_population = %s"
    values_normal_family = (reference_population,)
    cursor.execute(select_normal_family, values_normal_family)
    normal_families_list = []
```

```
    for data in cursor.fetchall():
        id_family = (data[0],)
        select_father = "SELECT id_father FROM family WHERE id = %s"
        cursor.execute(select_father, id_family)
        reference_father = cursor.fetchall()
        final_reference_father = reference_father[0][0]
        father = selectOneMouse(final_reference_father)

        select_mother = "SELECT id_normal_mother FROM normal_family WHERE id = %s"
        cursor.execute(select_mother, id_family)
        reference_mother = cursor.fetchall()
        final_reference_mother = reference_mother[0][0]
        mother = selectOneMouse(final_reference_mother)
        select_son = "SELECT reference FROM mouse WHERE id_son = %s"
        cursor.execute(select_son, id_family)
        son_list = []
        for reference in cursor.fetchall():
            son = selectOneMouse(reference[0])
            son_list.append(son)
        family = NormalFamily.NormalFamily(parent = father , mother = mother, referen
ce= id_family, children = son_list)
        normal_families_list.append(family)

    return normal_families_list
```

Para las familias poligámicas, hago lo mismo que las familias normales, pero lo único que cambia es a la hora de crear a las hembras, como pueden ser una o varias, tengo que hacer un for, igual que en los hijos, pero en las mujeres. Y luego hago lo mismo, creo una familia poligámica a partir del ratón macho, la hembra/hembras, los hijos y la referencia de la familia.

```
def selectPoligamicFamily(reference_population):
    """
    Esta función selecciona de memoria una familia poligamica y la crea, a partir de
la referencia de la poblacion
    Parametros:
        -reference_population: es la referencia de la poblacion
    Devuelve:
        -una lista con familias poligamicas
    """
    cursor = cnx.cursor()
    select_poligamic_family = "SELECT pf.id FROM family as f\
                              INNER JOIN polygamous_family as pf\
                              on f.id = pf.id\
                              WHERE f.id_population = %s"
    values_poligamic_family = (reference_population,)
    cursor.execute(select_poligamic_family, values_poligamic_family)
    poligamic_families_list = []
    for data in cursor.fetchall():
        id_family = (data[0],)

        select_father = "SELECT id_father FROM family WHERE id = %s"
        cursor.execute(select_father, id_family)
        reference_father = cursor.fetchall()
        final_reference_father = reference_father[0][0]
        father = selectOneMouse(final_reference_father)

        select_mother = "SELECT reference FROM mouse WHERE id_polygamous_mother = %s"
        cursor.execute(select_mother, id_family)
        mother_list = []
        for reference in cursor.fetchall():
            mother = selectOneMouse(reference[0])
            mother_list.append(mother)
```

```
        select_son = "SELECT reference FROM mouse WHERE id_son = %s"
        cursor.execute(select_son, id_family)
        son_list = []
        for reference in cursor.fetchall():
            son = selectOneMouse(reference[0])
            son_list.append(son)

        family = PoligamicFamily.PoligamicFamily(parent = father , mothers = mother_l
ist, reference= id_family, children = son_list)
        poligamic_families_list.append(family)
    return poligamic_families_list
```

Para juntar estas dos funciones, hago otra función que reciba la referencia de una población, y meto todas las familias en una lista, que returneo.

```
def selectAllFamilies(reference_population):
    """
    Esta función junta todas las familias que hay, a partir de la referencia de una p
oblación
    Parametros:
        -reference_population: es la referencia de la poblacion
    Devuelve:
        -una lista con familias tanto normales como poligamicas
    """
    families_list = []
    normal_families = selectNormalFamily(reference_population)
    poligamic_families = selectPoligamicFamily(reference_population)
    for normal_family in normal_families:
        families_list.append(normal_family)
    for poligamic_family in poligamic_families:
        families_list.append(poligamic_family)
    return families_list
```

Y una vez tengo todos los datos necesarios para poder crear una población, creo una función que reciba la referencia de la población, que seleccione todos los atributos de la misma, los pase a string, datetime y int, y por último, llamo a la función que crea todos los ratones de la población y los mete en una lista, es decir, selectAllMousesOfPopulation, y creo todas las familias y las meto en una lista con *selectAllFamilie.* De esta forma, ya tengo todos los datos necesarios para poder crear la población y returnearla. Por lo tanto, ya tendría el ejercicio completo. A pesar de ser el más complicado, lo saque victoriosamente.

```
def selectOnePopulation(reference_population):
    """
    Esta función selecciona de memoria una poblacion y la crea, a partir de la refere
ncia de la poblacion
    Parametros:
        -reference_population: es la referencia de la poblacion
    Devuelve:
        -una poblacion
    """
    cursor = cnx.cursor()

    select_population = "SELECT * from population WHERE reference = %s"
    value_reference = (reference_population,)
    cursor.execute(select_population,value_reference)
    population  = 0
    for data in cursor.fetchall():
        name = str(data[1])
        researcher = str(data[2])
        start_date = datetime.strptime(str(data[3]),'%Y-%m-%d').date()
```

```
        num_days = int(data[4])
        if (num_days <= 0):
            raise ValueError("Num days of the experiment should be > 0")

    mouses_list = selectAllMousesOfPopulation(reference_population)
    families_list = selectAllFamilies(reference_population)

    population = Population.Population(reference = reference_population, name = name,
 researcher = researcher, start_date = start_date,  num_days = num_days, animal_list
= mouses_list, families_list = families_list)
    cursor.close()
    return population
```

Por último, cierro la conexión cuando le damos a la opción 14 creando una función en el persistence y pasándosela a lab.py.

```
def closeConexion():
    """
    Esta funcion cierra la conexion con la terminal.
    """
    try:
        cnx.close()
    except Exception:
        print("no se puede cerrar la conexión")
```

## *LISTADO DE TODO EL CODIGO FUENTE DE LA APLICACIÓN*

Con respecto a lo que he modificado, he creado un modulo nuevo llamado persistence, he modificado el lab.py incorporando las funciones que hay en persistence y, por último, he cambiado algunos de los módulos de Alberto, como son Population, Family, PoligamicFamily, y NormalFamily, incorporando los cambios necesarios para que me pueda funcionar correctamente.

En lab.py, en cada opción hay un persistence.*nombredelafunción*, indicando la función del persistence.

En Population, hay un nuevo parámetro llamado ref_mouse, en el método __generate_Random_mouses hay modificaciones, incluyendo ref_mouse como parámetro y poniendo esa referencia en el primer ratón que se cree (líneas 319 a la 333) a parte veo si esa referencia es un numero int o no (línea 304 a la 305). En el método family_creation introduzco un parámetro nuevo reference_first_family y se lo paso en la creación de las familias. Por último, en family_reproduction le paso un parámetro nuevo llamado ref_last_mouse, y que se lo paso a la función reproduce de las familias.

En NormalFamily he introducido el get_mother, en el método reproduce le paso la ref_last_mouse y se lo indico a la hora de crear los ratones.

En PoligamicFamily hago lo mismo que en NormalFamily, le paso la ref_last_mouse, y se lo pongo al ratón que se crea.

Encontré 2 problemas en la práctica:

1. NormalFamily no incluía el método get_mother()
2. Family no tenía que tener la referencia privada, es decir, no tenía que poner __reference, solo una barra baja: _reference.

## *CONCLUSIONES*

| DIA | CONCEPTO | TIEMPO EMPLEADO ( EN HORAS) |
|---|---|---|
| 24-abr | **Entender todo el código, ejercicio 3 comienzo del 2,4,5** | 3 |
| 24-abr | **clase: corregir 5, hacer 2,4,6** | 3 |
| 25-abr | **problemática de referencias** | 3 |
| 26-abr | **solucionar problema de referencias con tutoría** | 1 |
| 28-abr | **ejercicio 7 y 8** | 2 |
| 29-abr | **empezar familias** | 1 |
| 30-abr | **problemática con referencias de familias** | 2 |
| 02-may | **intentar solucionar 10 (sin éxito) y empezar 11** | 1 |
| 05-may | **solucionar 10 y 11 empezar 1** | 3 |
| 07-may | **clase: continuar con el 1: población, ratones, problema con familia** | 4 |
| 08-may | **terminar el ejercicio 1, solucionar problema de familias y documentar** | 4 |
| | **escribir el Word con la explicación** | 2 |
| | **TOTAL :** | **29** |

Me costó bastante empezar con la práctica, pero una vez tenia los conceptos, fue más o menos todo rodado. Les dedique mucho, muchísimo tiempo a las referencias, ya que suponía cambiar el código de Alberto para que tuviese sentido mi código que estaba yo introduciendo. También los ejercicios 10 y 11 fueron laboriosos, pero sin duda, el que más tiempo he empleado ha sido el ejercicio 1. Si que es verdad que ya tenía el hábito. Ha sido una practica interesante, donde he aprendido a incorporar lo que habíamos hecho el año pasado con lo de este, por lo que muy completo. En total he dedicado a la práctica 29 horas, como se puede observar en la tabla; y en las horas están incluidas todas las tutorías que tuve.