Ficha 19

Registros y Objetos

1.] Introducción.

Hemos analizado en fichas anteriores el concepto de *estructura de datos,* que hemos definido como una variable capaz de contener varios valores al mismo tiempo. Sabemos que Python provee diversas formas de manejo de estructuras de datos y hemos usado algunas como las tuplas, las cadenas de caracteres y los rangos. Por razones relacionadas con el contexto y el enunciado de los problemas que hasta aquí se propusieron, en esas estructuras de datos que usamos hemos almacenado siempre elementos del mismo tipo: o bien sólo números enteros, o bien sólo cadenas de caracteres, o bien elementos de cualquier otro tipo que el programador hubiese necesitado.

Pero el hecho es que en algún momento el programador necesitará almacenar valores de *tipos diferentes* en una misma estructura de datos, para describir algún elemento o entidad del dominio del problema. Este tipo de situaciones existen y son (como siempre...) muy comunes.

A modo de ejemplo, supongamos que se requiere representar y almacenar datos de las distintas asignaturas de un plan de estudios en un programa simple. Supongamos que por cada asignatura se dispone de un código numérico de identificación y de una cadena de caracteres para su nombre.

Una primera idea simple (aunque no sea la idea final a emplear) es recordar que en Python una *tupla* es una secuencia de datos que *pueden* ser de tipos diferentes, de forma que se puede acceder a cada uno de esos datos por *vía de un índice*. En ese sentido, queda claro que una *tupla* podría usarse en forma *muy elemental* para representar una *asignatura*, ya que se podría emplear una tupla con dos elementos (uno para el código y otro para el nombre) por cada asignatura a describir, como se ve en el ejemplo que sigue [1] [2]:

```
a1 = 1, 'AED'
a2 = 2, 'PPR'
a3 = 3, 'TSB'
```

Las tres variables *a1*, *a2* y *a3* referencian a *tuplas* que contienen (cada una de ellas) los datos de una asignatura, y *cada uno de esos datos puede accederse mediante su índice*. Por ejemplo, el siguiente script muestra en consola los datos de las tres asignaturas:

```
print('Asignatura 1 - Código:', a1[0], 'Nombre:', a1[1])
print('Asignatura 2 - Código:', a2[0], 'Nombre:', a2[1])
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

Sin embargo, este enfoque tiene dos problemas: el primero es que desde el punto de vista de la *claridad del código fuente* el programador debe recordar en qué posición de cada tupla está cada valor de la asignatura y acceder a él mediante un índice numérico, que no es tan descriptivo ni simple de recordar. El programador debe saber que el código de la materia

lleva el índice 0 dentro de la tupla, y el nombre de la materia lleva el índice 1. El código fuente debería complementarse con innumerables comentarios aclaratorios que finalmente significan más trabajo y más cuidado para no cometer un error.

Y en este caso, el segundo problema es bastante más grave: en Python, una *tupla* es una *secuencia inmutable*… lo cual quiere decir que *una vez que se creó la tupla no pueden modificarse los valores almacenados en ella* [1]. Esto llevaría a situaciones claramente inaceptables: si el programador quisiese, por ejemplo, cambiar el nombre de una de las asignaturas, se produciría un error de intérprete y el programa se interrumpiría:

```
a3 = 3, 'TSB'
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])

a3[1] = 'DLC'   # Error aquí!!!
print('Asignatura 3 - Código:', a3[0], 'Nombre:', a3[1])
```

El script anterior lanzaría un error en la tercera línea, similar al que mostramos aquí:

```
Traceback (most recent call last):
  File "C:/Ejemplo/registros.py", line 7, in <module>
    a3[1] = 'DLC'
TypeError: 'tuple' object does not support item assignment
```

Por supuesto, Python provee otros tipos de secuencias y estructuras de datos *mutables* (como las *listas*) que podrían usarse en lugar de las *tuplas* para evitar el problema anterior, pero seguiríamos teniendo el primer inconveniente: el programador tendrá que recordar en que casillero exacto de cada estructura está almacenado cada valor para deducir el índice de acceso.

Claramente se requiere poder almacenar valores de tipos diferentes en una misma estructura mutable de datos, pero de forma que la sintaxis y forma de gestión sea más transparente. Como vimos, algunos de los enfoques que Python provee para lograr esto terminan siendo poco claros y por lo tanto necesitamos una nueva estructura. Esa estructura es la que estudiaremos en esta Ficha y se designa con el nombre genérico de *registro*. Y una vez que hayamos introducido el concepto de registro, haremos un giro conceptual hacia los fundamentos iniciales del paradigma de *Programación Orientada a Objetos (POO)*, replanteando el propio concepto de registro para introducir la idea de *objeto* (y la forma diseñar y aplicar objetos en nuestros programas).

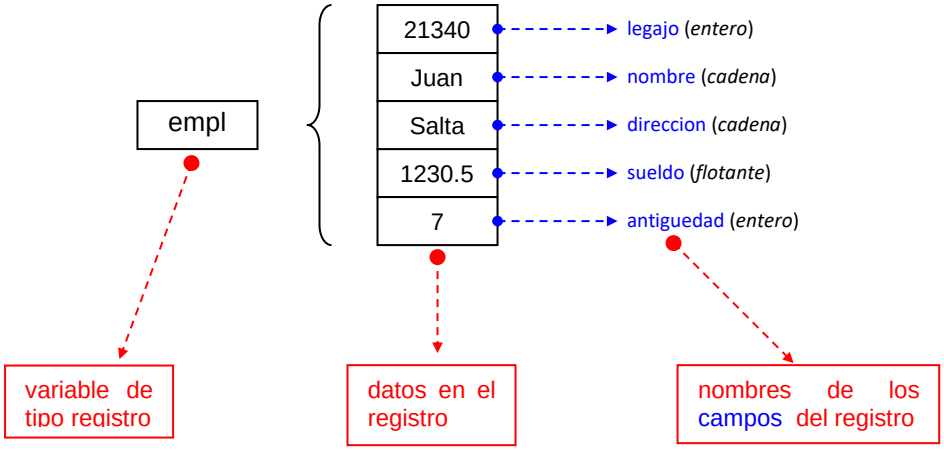
2.] Registros en Python.

Comencemos desde lo más elemental: en programación, un *registro* es un conjunto *mutable* de valores que *pueden ser de distintos tipos*. Cada componente de un registro se denomina *campo* (o *atributo*, dependiendo del contexto). Los *registros* son útiles para representar en forma clara a cualquier elemento (o *entidad*) del dominio o enunciado del problema que el programador necesite manejar y cuya descripción pudiera contener datos de tipos diferentes.

Por ejemplo, si se desea en un programa representar la información de un *empleado* de una empresa, puede usarse una variable *empl* de tipo *registro* con *campos* para el *legajo* (que sería un valor *de tipo entero*), el nombre (una *cadena de caracteres*), la dirección (otra *cadena de caracteres*), el *sueldo básico* (un valor de *coma flotante*) y la *antigüedad* de ese

empleado en la empresa (que sería otro valor *entero*). El siguiente gráfico muestra una representación de la forma que podría tener tal variable [3]:

Figura 1: Esquema de una variable de tipo registro.



En el esquema anterior se supone el uso de una *variable de tipo registro* llamada *empl*, la cual consta de cinco *campos*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*. La importancia de los registros como estructuras de datos es evidente en el ejemplo anterior: sin registros en algunos lenguajes no sería posible almacenar en una misma variable todos los datos del empleado supuesto, ya que esos datos en este caso son todos de tipos diferentes, y en otros lenguajes se produciría mucha confusión en el acceso a cada campo, ya que el programador debería usar índices y saber de antemano en qué casilla quedó cada valor.

Note que no hay ningún problema si un programador quiere definir un registro que contenga *campos del mismo tipo*: un *registro* es una colección de variables llamadas *campos*, y esos campos *pueden* ser de tipos diferentes (pero ***no deben ser obligatoriamente*** de tipos distintos).

Para definir variables de tipo registro en un programa, lo común es proceder primero a declarar un nuevo nombre o identificador de *tipo de dato* para el registro. Con este nuevo tipo, se definen luego las variables. La declaración de un *tipo registro* se puede hacer en Python con la palabra reservada *class*, básicamente en la forma que se indica en el siguiente esquema (ver modelo *test01.py* - proyecto *[F19] Registros y Objetos* que acompaña a esta ficha) [1]:

```
# declaración de un tipo Empleado como registro vacío
class Empleado:
    pass

# creación de variables de tipo Empleado
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10

e2 = Empleado()
e2.legajo = 2
e2.nombre = 'Luis'
e2.direccion = 'Calle 2'
```

```
e2.sueldo = 20000
e2.antiguedad = 15

e3 = Empleado()
e3.legajo = 3
e3.nombre = 'Pedro'
e3.direccion = 'Calle 3'
e3.sueldo = 25000
e3.antiguedad = 20
```

En la forma que se trabajó en el esquema anterior, el identificador *Empleado* constituye un nuevo *tipo de dato* para el programa y en base a él se pueden definir variables que tendrán la estructura de campos o atributos que el programador necesite.

Las variables *e1*, *e2, y e3* del ejemplo anterior se crean como referencias a *registros* de tipo *Empleado* a través de la función constructora *Empleado()*, que está automáticamente disponible para tipos definidos mediante la palabra reservada *class*. Cada vez que se invoca a esa función, Python crea un registro vacío, sin ningún campo, y *retorna la dirección de memoria* donde ese registro vacío quedó alojado:

```
e1 = Empleado()
```

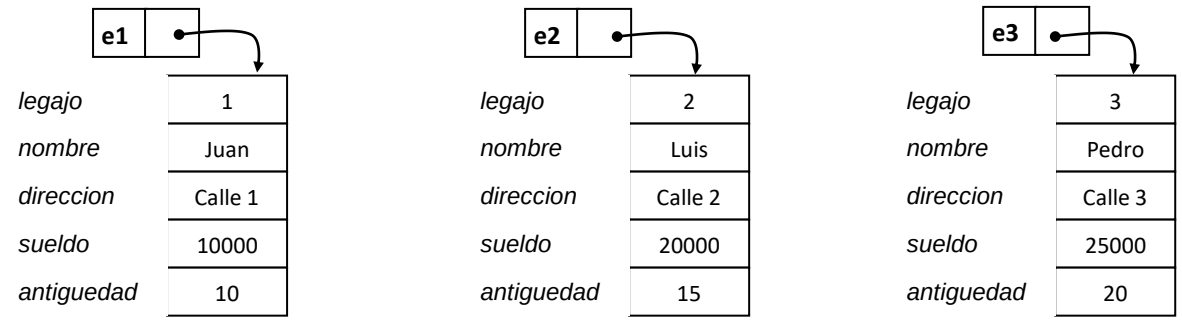
En el ejemplo anterior, la dirección retornada por la *función constructora* se asigna en la variable *e1*, que de allí en adelante se usa para acceder y gestionar el registro (se dice que *e1* está *apuntando al registro* o que está *referenciando al registro*).

Como cada variable se crea inicialmente como un *registro vacío*, entonces luego el programador debe ir agregando campos (o atributos) a esas variables según lo necesite, *usando el operador punto (.)* para hacer el acceso:

```
e1 = Empleado()
e1.legajo = 1
e1.nombre = 'Juan'
e1.direccion = 'Calle 1'
e1.sueldo = 10000
e1.antiguedad = 10
```

Como se dijo, el acceso a los campos individuales de una variable *registro* se hace con el *operador punto (.)* en forma muy sencilla: se escribe primero el *nombre de la variable* registro (que contiene la dirección de memoria de ese registro), luego un *punto*, y finalmente el *nombre del campo* que se quiere agregar o acceder. Con esto se forma lo que se conoce como el *identificador del campo*, y a partir de aquí se opera con ese campo en forma normal, como se haría con cualquier variable común. En este caso, se están agregando cinco campos a la variable *e1*, llamados *legajo*, *nombre*, *direccion*, *sueldo* y *antiguedad*, cada uno asignado con el valor inicial que se requiera. Luego de crearse las tres variables *e1*, *e2* y *e3* y luego de agregarse a cada una los campos citados, la memoria asignada a ellas podría verse como en el esquema de la *Figura 2*:

Figura 2: Esquema de memoria para tres registros de tipo *Empleado*.



Como se puede observar, cada variable apunta a un registro diferente de forma que cada uno contiene su propia copia de cada uno de los cinco campos, y cada campo en cada registro tiene su propio valor, que es independiente del valor que ese mismo campo tenga en los otros registros. Las siguientes instrucciones muestran en consola estándar el *legajo* y el *nombre* de cada uno de los tres empleados:

```
# mostrar legajo y nombre de cada empleado...
print('Empleado 1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('Empleado 2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('Empleado 3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
```

Además, como un registro es una estructura *mutable*, el valor de cualquier campo puede ser modificado cuando se necesite, recordando siempre que el acceso a un campo se realiza mediante el *operador punto*. Mostramos algunos ejemplos de operaciones válidas para los campos de los registros *e1*, *e2* y *e3* que ya hemos definido en los ejemplos anteriores [3]:

```
# algunas operaciones válidas...

# cambiar el legajo de e1...
e1.legajo = 4

# hacer que la antiguedad de e3 sea igual que la de e2...
e3.antiguedad = e2.antiguedad

# cargar por teclado el sueldo de e2...
e2.sueldo = float(input('Ingrese el nuevo sueldo del empleado 2: '))

# mostrar la direccion del empleado e1...
print('Direccion del empleado 1:', e1.direccion)

# sumar un año a la antiguedad de e3...
e3.antiguedad += 1
```

También recuerde que en Python las variables no quedan atadas a un tipo fijo y estático, y de hecho, nada impediría entonces que se cree una cuarta variable de tipo *Empleado* con una *estructura de campos diferente* a las tres que ya existen, como se ve en la variable *e4* que se muestra a continuación, en la cual *se ha agregado un campo extra para indicar el cargo del empleado* (ver modelo *test02.py* en el proyecto *[F19] Registros y Objetos* que acompaña a esta Ficha) [1] [2]:

```
# creación de una variable Empleado con un campo adicional...
e4 = Empleado()
e4.legajo = 10
e4.nombre = 'Luis'
e4.direccion = 'Calle 4'
e4.sueldo = 50000
e4.antiguedad = 30
e4.cargo = 'Gerente'
```

En el ejemplo anterior, la variable *e4* se crea también como un registro de tipo *Empleado*, y se agregan en ella los mismos cinco campos que se habían incluido en las otras tres. Pero en la última instrucción se agrega para *e4* el campo *cargo*, que las otras tres no tenían: La variable *e4* (y *sólo la variable e4*) contendrá el campo *cargo* para asignar en él el nombre del puesto o cargo ocupado por el empleado. La siguiente secuencia de instrucciones muestra los legajos y los nombres de los cuatro empleados, más el cargo del cuarto:

```
# mostrar datos básicos...
print('E1 - Legajo:', e1.legajo, '- Nombre:', e1.nombre)
print('E2 - Legajo:', e2.legajo, '- Nombre:', e2.nombre)
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre)
print('E4 - Legajo:', e4.legajo, '- Nombre:', e4.nombre, '- Cargo:', e4.cargo)
```

Como las variables *e1*, *e2* y *e3* no contienen el campo *cargo*, se produciría un error de intérprete al intentar accederlo para consultarlo:

```
print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
```

La instrucción anterior produciría un error como el siguiente:

```
File "C:/Ejemplo/test02.py", line 42, in <module>
    print('E3 - Legajo:', e3.legajo, '- Nombre:', e3.nombre, '- Cargo:', e3.cargo)
AttributeError: 'Empleado' object has no attribute 'cargo'
```

En general, la inicialización de un registro (así como cualquier otra operación) puede canalizarse por medio de funciones: no hay motivo para escribir largas secuencias de código fuente que hagan la misma tarea. En el siguiente modelo simple (*test03.py* – proyecto *[F19] Registros y Objetos*) se crean los mismos tres registros de tipo *Empleado* que ya hemos visto, pero cada uno de ellos es inicializado mediante una función que hemos llamado *init()*, y luego cada uno de ellos es visualizado mediante otra función que hemos llamado *write()*:

```
# declaración de un tipo registro vacío
class Empleado:
    pass


# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant


# una función para mostrar un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')


# una funcion de prueba
def test():
    # creación de variables vacias de tipo Empleado...
    e1 = Empleado()
    e2 = Empleado()
    e3 = Empleado()

    # inicializacion de campos de las tres variables...
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    init(e2, 2, 'Luis', 'Calle 2', 20000, 15)
    init(e3, 3, 'Pedro', 'Calle 3', 25000, 20)

    # visualizacion de los valores de los tres registros...
    write(e1)
```

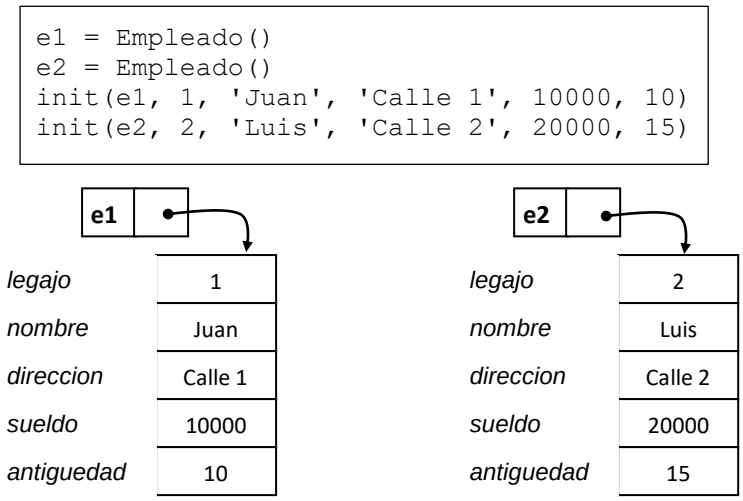
```
    write(e2)
    write(e3)


# script principal...
if __name__ == '__main__':
    test()
```

Note que en este ejemplo simple, es la propia función ***init()*** la que finalmente "define" la estructura de campos del registro que entra como primer parámetro.

Un detalle que no debe pasarse por alto es que como ya dijimos, en Python una variable de tipo *registro* contiene en realidad la *dirección de memoria* del registro asociado a ella (se dice la variable es *una referencia a ese registro*). Por lo tanto, si se asignan dos variables de tipo *registro* entre ellas (por ejemplo, dos registros de tipo *Empleado*) ***no se estará haciendo una copia*** del registro original, ***sino una copia de las direcciones de memoria***, haciendo que ambas variables queden apuntando *al mismo registro*.

A partir de aquí, entonces, el programador debe tener cuidado de entender bien lo que hace cuando opera con variables que son *referencias*: supongamos que las dos variables *e1* y *e2* apuntan a dos registros diferentes, ambos de tipo *Empleado*. El gráfico de memoria se vería como en la gráfica que sigue:

Figura 3: Dos referencias apuntando a registros diferentes.



Si luego se hace una asignación como la siguiente:

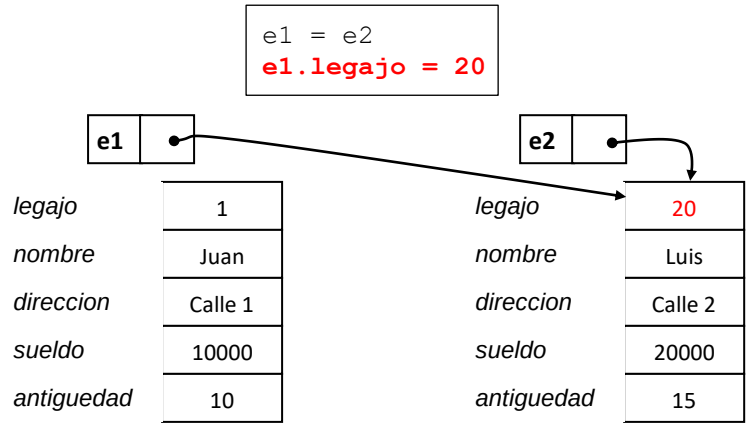
```
e1 = e2
```

entonces *ambas* variables pasarán a apuntar o referenciar al *mismo* registro (el que originalmente era apuntado por *e2*). Por lo tanto, si se modifica el valor de algún campo para e1, se estará modificando *también* el registro referido por *e2*. La asignación *e1 = e2* ***no copia*** los contenidos del registro apuntado por *e2* hacia el registro apuntado por *e1*. Lo que copia es la *dirección contenida en e2* en la variable *e1* (ver la *Figura 4* en página siguiente).

El registro apuntado originalmente por *e1* queda *des referenciado*: esto significa que el programa ha perdido la dirección de memoria de ese registro y ya no hay forma de volver a utilizarlo. En algunos lenguajes (como C++), esta situación implica un error por parte del

programador, ya que el registro des referenciado sigue ocupando memoria aunque ya no pueda ser accedido.

Figura 4: Dos referencias apuntando al mismo registro.



Pero en Python (como en Java y otros lenguajes) eso no es necesariamente un error, pues en Python existe un sistema de *recolección de residuos* de memoria que se encarga de chequear la memoria en busca de bloques de memoria "perdidos" en tiempo de ejecución. Cuando un programa se ejecuta, *en otro hilo de ejecución paralelo* corre también un proceso llamado *garbage collector*, que se encarga de liberar la memoria ocupada por los bloques des referenciados, sin que el programador deba preocuparse por ellos.

El hecho ya citado de que una variable registro mantiene una referencia al registro, también hace que al enviar como parámetro una variable registro a una función, la misma pueda usarse para modificar el ***contenido*** del registro (aunque no puede cambiarse la dirección contenida en el parámetro). La función *init()* que ya hemos visto, hace justamente eso: toma como parámetro una variable *empleado* apuntando a un registro *ya creado*, y procede a modificar *su contenido* para inicializarlo:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

Si la función *init()* intentase cambiar el valor de la variable *empleado* como se muestra en el modelo siguiente:

```
def init(empleado, leg, nom, direc, suel, ant):
    empleado = Empleado()
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
```

entonces dentro de la función la variable *empleado* estaría apuntando a un registro que efectivamente sería inicializado con los cinco campos asignados. Pero al terminar de ejecutarse la función, la variable local *empleado* se destruye, y con eso el registro que se había creado queda des referenciado. La variable registro que hubiese sido enviada como parámetro actual quedará sin cambios y seguirá apuntando al registro que apuntaba

originalmente. Con estas consideraciones, el siguiente programa (modelo *test04.py* en el proyecto *F[19] Registros y Objetos]*) produce un error al ejecutarse:

```
# declaración de un tipo registro vacío
class Empleado:
    pass


# una función para inicializar un registro de tipo Empleado
def init(empleado, leg, nom, direc, suel, ant):

    # cuidado con esta línea...
    empleado = Empleado()

    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant


# una función para mostrar por consola un registro de tipo Empleado
def write(empleado):
    print("\nLegajo:", empleado.legajo, end=' ')
    print("- Nombre:", empleado.nombre, end=' ')
    print("- Direccion:", empleado.direccion, end=' ')
    print("- Sueldo:", empleado.sueldo, end=' ')
    print("- Antiguedad:", empleado.antiguedad, end=' ')


# una funcion de prueba
def test():
    e1 = Empleado()
    init(e1, 1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)


# script principal...
if __name__ == '__main__':
    test()
```

El error se produce cuando al regresar de la función *init()* se invoca a la función *write()* para mostrar el registro. La función *write()* intentará mostrar los campos de la variable *e1*, pero debido al mecanismo antes descripto, *esa variable no tiene ningún campo*: fue creada en la función *test()* como registro vacío, luego se envió como parámetro a *init()*, la cual intentó cambiar la dirección que contenía por otra. Como eso no es posible debido al mecanismo de parametrización por valor, la variable *e1* volvió a *test()* exactamente con el mismo valor que tenía antes de invocar a *init()*: la dirección de un registro vacío. Y eso provoca el fallo en *write()*…

```
Traceback (most recent call last):
  File "C:/Ejemplo/test04.py", line 37, in <module>
    test()
  File "C:/Ejemplo/Fuentes/[F14] Registros/test04.py", line 32, in test
    write(e1)
  File "Ejemplo/test04.py", line 21, in write
    print("\nLegajo:", empleado.legajo, end=' ')
AttributeError: 'Empleado' object has no attribute 'legajo'
```

Obviamente, los problemas anteriores se resuelven de inmediato en forma simple, haciendo que nuestra la función *init() cree el registro con la función constructora, lo inicialice y retorne el registro* que acaba de crear e inicializar, en forma similar a lo siguiente (ver modelo *test05.py* en el proyecto *F[19] Registros y Objetos]*):

```
class Empleado:
    pass

def init(leg, nom, direc, suel, ant):
    empleado = Empleado()
    empleado.legajo = leg
    empleado.nombre = nom
    empleado.direccion = direc
    empleado.sueldo = suel
    empleado.antiguedad = ant
    return empleado
```

Si la función *init()* está definida así, entonces la creación de un objeto puede hacerse de forma más simple y natural, como se ve en la función *test()* que sigue:

```
def test():
    e1 = init(1, 'Juan', 'Calle 1', 10000, 10)
    write(e1)
```

Note que esta nueva versión de nuestra función *init()* ahora no necesita tomar como parámetro el registro cuyos campos se desea crear, ya que la misma función crea ese registro y lo completa campo a campo, para finalmente retornarlo.

Para aplicar lo visto, veamos ahora un caso de estudio muy conocido, como es el de la representación de puntos en un plano. El siguiente enunciado formaliza la idea:

Problema 45.) *Desarrollar un programa que permita manejar puntos en un plano. Por cada punto se deben indicar sus coordenadas (que pueden ser números en coma flotante) y una cadena de caracteres a modo de descriptor del punto cuando se muestren sus valores en pantalla. Incluir un menú de opciones que permita:*

1. *Cargar por teclado los datos de un punto y mostrar esos datos en pantalla.*
2. *Cargar por teclado los datos de un punto, y mostrar la distancia al origen desde ese punto.*
3. *Cargar por teclado los datos de dos puntos, y mostrar la pendiente de la recta que los une.*

Discusión y solución: Un punto *p1* en un plano bidimensional gestionado a través de un par de ejes cartesianos, puede ser representado en forma simple a partir de sus dos coordenadas *(x, y)* que dan la distancia horizontal y vertical de ese punto *p1* al origen del sistema[1], como se ve en la *Figura 5* de la página siguiente.

Esto permite rápidamente pensar en que un punto puede representarse en un programa en Python mediante un registro que contenga tres campos: dos valores de tipo *float* para las

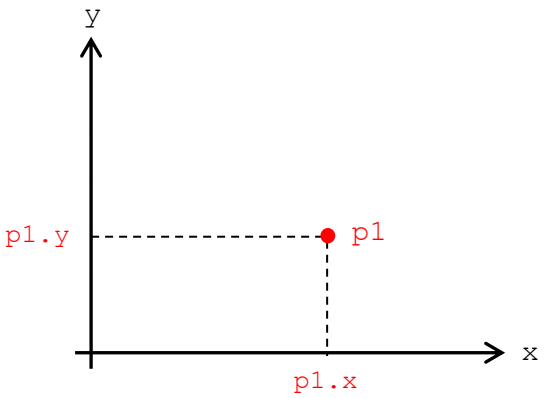
[1] El uso de un sistema de ejes cartesianos para representar objetos o propiedades en función de dos o más coordenadas o valores aparece en las más insospechadas disciplinas, y en ese sentido el cine ha mostrado algunas escenas memorables. Un ejemplo concreto y muy recordado es la película *Dead Poets Society* (*La Sociedad de los Poetas Muertos*) de 1989, dirigida por *Peter Weir*. El profesor Keating (interpretado por *Robin Williams*) enseña literatura en un colegio secundario. El libro de texto que usan sus alumnos les explica cómo "medir" el valor de una poesía usando un sistema de ejes cartesianos… y el profesor Keating les ordena arrancar esa página del libro calificándola de basura… ¡Toda una perlita!

coordenadas, y una cadena de caracteres para el descriptor (que en el gráfico anterior es "p1"). Dentro del proyecto *[F19] Registros y Objetos* (y a su vez en la carpeta *test06*), el módulo *geometry.py* contiene la declaración del tipo registro *Point* que usaremos para representar puntos, siguiendo el último modelo que hemos sugerido:

```
class Point:
    pass

def init(cx, cy, desc='p'):
    p = Point()
    p.x = cx
    p.y = cy
    p.descripcion = desc
    return p
```

La función *init(cx, cy, desc='p')* crea un registro vacío *p* de tipo Point, y luego crea y asigna los campos del punto *p* que se acaba de crear tomando los valores de esos campos desde los parámetros *cx*, *cy* y *desc*. El parámetro formal *desc* tiene por defecto asignada la cadena 'p', por lo cual si ese parámetro se omite al invocar a *init()*, el campo *descripcion* de *p* quedará valiendo 'p'.

Figura 5: Representación de un punto p1 en un sistema cartesiano (primer cuadrante).



El mismo módulo *geometry.py* contiene el resto de la funciones pedidas en el enunciado. El cálculo de la distancia de un punto hasta el origen de coordenadas del sistema se realiza con la función *distance()*, que simplemente aplica el Teorema de Pitágoras:

```
def distance(p):
    # Pitágoras...
    return math.sqrt(pow(p.x, 2) + pow(p.y, 2))
```

El módulo *geometry.py* provee también una función *to_string()* simple pero muy cómoda, que crea una cadena de caracteres a partir de un registro *p* de tipo *Point*, y retorna esa cadena lista para ser mostrada en consola si fuese necesario (o para ser usada donde disponga el programador):

```
def to_string(p):
    r = str(p.descripcion) + '(' + str(p.x) + ',' + str(p.y) + ')'
    return r
```

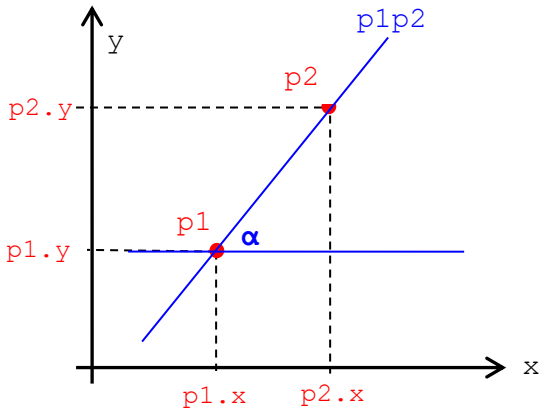
El cálculo de la pendiente de la recta que une a dos puntos *p1* y *p2*, se realiza mediante la función *gradient(p1, p2)*. Esa pendiente no es otra cosa que la *tangente trigonométrica* del ángulo *α* que forma la recta *p1p2* con la horizontal que pasa por *p1* (ver *Figura 6* en página siguiente).

Sólo hay que tomar la precaución de controlar que *Δx* no sea cero (es decir, controlar que los puntos *p1* y *p2* no formen una recta vertical) pues en ese caso el cociente no puede calcularse (y se dice que la pendiente de una recta vertical es indefinida).

En base a lo expuesto, la función *gradient(p1, p2)* es la siguiente:

```
def gradient(p1, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - p1.y
    dx = p2.x - p1.x

    # si los puntos no son colineales verticales,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    # ... este retorno debería ser controlado al regresar...
    return None
```

Figura 6: Representación geométrica de la pendiente de la recta *p1p2*.



pendiente(p1p2) = tg(α) = Δy / Δx
= (p2.y – p1.y) / (p2.x – p1.x)

El módulo *geometry.py* entonces puede quedar así (ver proyecto *[F19] Registros y Objetos*, *carpeta test06*):

```
import math


class Point:
    pass


def init(cx, cy, desc='p'):
    p = Point()
    p.x = cx
    p.y = cy
    p.descripcion = desc
    return p


def to_string(p):
    r = str(p.descripcion) + '(' + str(p.x) + ', ' + str(p.y) + ')'
    return r
```

```
def distance(p):
    # Pitágoras...
    return math.sqrt(pow(p.x, 2) + pow(p.y, 2))


def gradient(p1, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - p1.y
    dx = p2.x - p1.x

    # si los puntos no son colineales verticales,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    # ... este retorno debería ser controlado al regresar...
    return None
```

Y dado ese módulo, entonces el programa completo puede ser el siguiente (de nuevo, ver proyecto *[F19] Registros y Objetos*, *carpeta test06*):

```
import geometry


def opcion1():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = geometry.init(cx, cy)
    print(geometry.to_string(p))


def opcion2():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = geometry.init(cx, cy)
    d = geometry.distance(p)
    print('Distancia al origen:', d)


def opcion3():
    cx1 = float(input('Punto 1 - Coordenada x: '))
    cy1 = float(input('Punto 1 - Coordenada y: '))
    p1 = geometry.init(cx1, cy1)

    cx2 = float(input('Punto 2 - Coordenada x: '))
    cy2 = float(input('Punto 2 - Coordenada y: '))
    p2 = geometry.init(cx2, cy2)

    pd = geometry.gradient(p1, p2)
    if pd is not None:
        print('Pendiente de la recta que los une:', pd)
    else:
        print('Pendiente no definida (recta vertical)')


def menu():
    op = -1
    while op != 4:
```

```
        print('1. Cargar y mostrar un punto')
        print('2. Distancia al origen')
        print('3. Pendiente de la recta que une dos puntos')
        print('4. Salir')
        op = int(input('Ingrese opcion: '))

        if op == 1:
            opcion1()
        elif op == 2:
            opcion2()
        elif op == 3:
            opcion3()


if __name__ == '__main__':
    menu()
```

Dejamos para el lector el análisis de los detalles del programa.

3.] Registros y Objetos: Introducción a la Programación Orientada a Objetos en Python.

La *Programación Orientada a Objetos* (abreviada de ahora en más como *POO*) es un conjunto de reglas y principios de programación (o sea, un *paradigma de programación*) que busca representar en un programa las entidades u objetos del dominio (o enunciado) del problema, pero en la forma más natural posible.

En el paradigma de *Programación Estructurada (PE)* que hemos estado aplicando hasta aquí, el programador busca identificar los *procesos* (en forma de subproblemas) que aplicados sobre los datos permitan obtener los resultados buscados. Y esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de resolución de problemas que los programadores orientados a objetos siguen usando dentro del nuevo paradigma. Entonces, ¿a qué viene el paradigma de la *POO*?

Los lenguajes de *PE* se orientaron a la forma de trabajar descomponiendo procesos en subprocesos y programando a cada uno como *subrutina*, *función*, o *procedimiento* (todas formas de referirse al mismo concepto, que luego la *POO* designaría en forma más general como *método*). Pero en la práctica, la *orientación al subproblema* del paradigma de *PE* resulta ser una forma de programar que hace que sea difícil de aplicar en el desarrollo de sistemas realmente grandes (estamos hablando de 50000 líneas de código o más...), o en el desarrollo de sistemas de mucha complejidad en cuanto a las relaciones entre los procesos detectados. La sola orientación a la descomposición en subproblemas no alcanza cuando el sistema es tan complejo: se vuelve difícil de visualizar su estructura general, se hace complicado realizar hasta las más pequeñas modificaciones sin que estas reboten en la lógica de un número elevado de otras subrutinas, y finalmente se torna una tarea casi imposible replantear el sistema para agregar nuevas funcionalidades complejas que permitan que ese sistema simplemente siga siendo útil frente a continuas nuevas demandas.

La *POO* significó una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un programa. Ahora, el objetivo primario *no es identificar procesos* sino *identificar actores*: las *entidades* u *objetos* que aparecen en el escenario, enunciado o dominio del problema, *tales que esos objetos tienen no solo datos asociados sino también algún comportamiento que son capaces de ejecutar*. Piense el lector en un objeto como en un robot virtual: el programa tendrá muchos robots virtuales (objetos

de software) que serán capaces de realizar eficiente y prolijamente ciertas tareas en las que serán expertos, e interactuando con otros robots virtuales (objetos...) serán capaces de resolver el problema que el programador esté planteando.

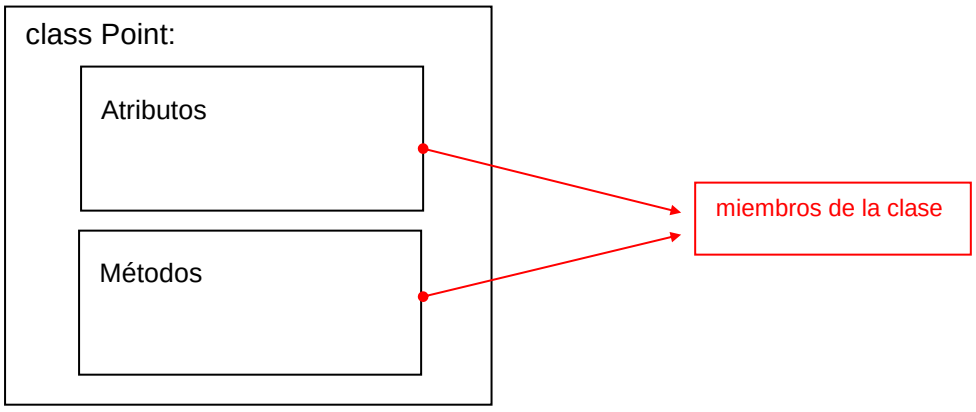
Para hacer eso, los lenguajes orientados a objetos o con soporte para la orientación a objetos (como Java, C++, Smalltalk, Eiffel, C#[2], Ada, Python y tantos otros) usan descriptores de entidades conocidos como *clases*. Básicamente, una *clase* es la descripción (a modo de plantilla) de una entidad u objeto de forma que luego esa descripción pueda usarse para crear muchos *objetos* que respondan a la descripción dada. Para establecer analogías, se puede pensar que una *clase* se corresponde con el concepto de *tipo registro* de la *PE* tradicional (y que reconocemos que hemos presentado y manejado en forma algo tosca en Python en esta ficha…), y los *objetos* creados a partir de la clase (llamados también *instancias* en el mundo de la *POO*) se corresponden aproximadamente con el concepto de *variable de tipo registro* (o simplemente *registro*) de la *PE*. Así como el *tipo registro* es uno solo y describe la forma que tienen todos los registros de ese tipo, la *clase* es única y describe *la forma y el comportamiento* de los muchos *objetos* o *instancias* que se creen de esa clase.

Como dijimos, para describir objetos que responden a las mismas características de forma y comportamiento, se declara una *clase*. La definición de una clase en Python (como ya podemos adivinar) se hace mediante la palabra reservada *class* (que es obviamente la misma que hemos usado para introducir en forma elemental el concepto de registro), y en cualquier lenguaje con soporte de POO incluye esencialmente dos elementos:

- **Atributos:** Son *variables* que se declaran dentro de la clase (equivalentes en todo al concepto de *campos* en un registro), y sirven para indicar la *forma* de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto *tiene*.
- **Métodos:** Son *funciones* declaradas *dentro de la clase* (y esta la primera gran novedad respecto del concepto llano de registro), usadas para describir el *comportamiento* de los objetos representados por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto *hace*.

Todos los *atributos* y *métodos* de una clase conforman el conjunto de *miembros* de esa clase. Esquemáticamente, si quisiéramos comenzar a diseñar una clase para representar puntos en un plano al estilo de lo hecho con registros al final de la sección anterior, el modelo (muy general) podría verse como se indica en la *Figura 7*:

Figura 7: Esquema general de una clase.



[2] Léase como *C sharp*.

En el contexto de la *PE* tradicional, un *tipo registro* describe *la forma* que tendrán las variables (o *registros*) de ese tipo, *y se definen por separado las funciones* que procesarán a esos registros tomándolos como parámetro. Pero en el ámbito de la *POO*, una clase describe *la forma* ***y*** *el comportamiento* que tendrán las variables (u *objetos* o *instancias*) de esa clase: las funciones (o sea, los *métodos*) para procesar a esos objetos y *que definen el comportamiento*, *están incluidas dentro de la propia clase*, y por lo tanto están disponibles en todos los objetos de esa clase.

Hasta aquí hemos mostrado en forma muy elemental la forma en que se podría implementar el concepto de *registro simple* en Python, desde la óptica tradicional de la *PE*. El problema 45 con el que se cerró la sección anterior es un ejemplo claro de ese enfoque: el *tipo registro Point* se definió vacío, y luego los campos en cada variable de tipo *Point* se fueron agregando con una función separada (que hemos llamado *init()* en el ejemplo). Cada variable de tipo *Point* es un conceptualmente un *registro simple*, y definimos funciones *separadas* (como *to_string()*, *distance()* y *gradient()*) que toman un *Point* como parámetro y lo procesan de alguna forma.

El concepto de *registro* como agrupamiento de variables que pueden ser de diferentes tipos es propio del contexto de la *PE*, y muchos lenguajes que soportan ese paradigma disponen de palabras reservadas específicas para definir *tipos de registros* (tales como *record* en Pascal o *struct* en C). Pero en el contexto de la *PE* un *registro* es una colección de variables (o campos) que pueden ser de tipos diferentes, pero *un registro no puede contener funciones*.

En Python la palabra reservada *class* se usa para definir *clases* en el contexto de la *POO*, y no *tipos de registros* simples al estilo de la *PE*. Por supuesto, las técnicas que hemos mostrado para crear e inicializar un *tipo de registro* simple mediante la palabra reservada *class* en Python es válida, pero el hecho es que no hay motivo real para desaprovechar la potencia de la POO y del uso de clases y métodos en esas clases si el lenguaje utilizado provee el soporte. Y Python provee clases… no registros. Por lo tanto, veamos cuál es la forma de usar correctamente el recurso.

4.] Clases en Python. Función Constructora (o Método Constructor).

Los elementos que en un *registro* se llaman *campos* (o sea, las *variables* definidas dentro del *tipo de registro*), reciben el nombre de *atributos* cuando se definen en una *clase*. Y si una *función* se define dentro de una *clase*, pasa a designarse como un *método* (en lugar de una *función*).

Finalmente (y como ya vimos), las variables que se definen en base a un *tipo registro* se llaman ellas mismas *registros* (o también *instancias*), pero las variables que se definen en base a una *clase* se designan como *objetos* (aunque aquí también cabe la designación de *instancias*).

En Python, la palabra reservada *class* está claramente pensada para definir *clases* (y no *tipos de registros* simples). Obviamente, un programador que aún no tiene conocimientos de *POO* puede usar la palabra *class* para definir *tipos de registros* simples incluyendo sólo campos o atributos en la clase pero sin incluir funciones o métodos. Y ese ha sido el criterio que hemos seguido a lo largo de esta Ficha: usar versiones simplificadas de clases (con atributos pero sin métodos) para emular el concepto de registro.

Recuerde: una clase puede pensarse como un tipo de registro que además de contener campos (o atributos), puede contener funciones (o métodos). Y uno de los métodos más importantes que una clase puede contener se designa como *función constructora* o (más correcto) *método constructor* (o simplemente, un *constructor*).

El objetivo de un *constructor* es la creación y la inicialización de un objeto o instancia. En Python, un *método constructor* debe llamarse específicamente *__init__()* (con dos guiones de subrayado delante y detrás de la palabra *init*) y como dijimos, debe definirse dentro del ámbito de la clase. El programa siguiente (*test07.py – Proyecto [F19] Registros y Objetos*) es una adaptación de los que ya vimos para crear registros de tipo *Empleado*, pero ahora modificados para que la clase *Empleado* contenga el constructor *__init__()* ya citado:

```
# declaración de una clase con un constructor...
class Empleado:
    # un constructor dentro de la clase Empleado...
    def __init__(self, leg=0, nom='No sabe', direc='No sabe', suel=0, ant=1):
        self.legajo = leg
        self.nombre = nom
        self.direccion = direc
        self.sueldo = suel
        self.antiguedad = ant
```

En Python todo método de una clase ***debe*** definir un primer parámetro formal llamado *self* que representa al objeto o instancia sobre el cual se aplica el método. Si ese método es el constructor (*__init__()*) entonces *self* es el objeto o registro que se está creando e inicializando.

Si la clase *Empleado* tiene definido el constructor que hemos mostrado, entonces la siguiente instrucción crea un objeto *e1* de la clase *Empleado*, y lo inicializa con los valores enviados como parámetros actuales:

```
e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
```

La parte derecha de la asignación anterior es una invocación al constructor. Aquí hay dos hechos muy importantes a destacar:

a. Aun cuando el nombre *__init__()* no está presente en esa invocación, Python interpreta que si el nombre de una clase se usa a la derecha de una expresión de asignación y se le pasan parámetros, entonces se está invocando *implícitamente* al constructor de esa clase.

b. Note además que no es necesario enviar como parámetro explícito al propio objeto *e1* que se quiere crear para ser asignado en *self*: Python también interpreta que si se invoca a un constructor, el objeto que se construye es enviado *automáticamente* como parámetro a ese constructor y asignado en *self*, y por si fuese poco, el constructor automáticamente también retorna la dirección del objeto creado (sin necesidad de incluir un *return*) razón por la cual el retorno del constructor es asignado en la variable *e1*.

En otras palabras: la expresión anterior invoca al constructor *__init__()* de la clase *Empleado*. El parámetro *self* de ese constructor queda automáticamente asignado con la dirección del objeto que se está creando (sin tener que enviar un parámetro actual desde el exterior para que sea asignado en *self*). El resto de los parámetros actuales enviados (los valores 1, 'Juan', 'Calle 1', 10000, y 10) son asignados en forma normal en los parámetros formales *leg*, *nom*,

direc, *suel* y *ant* y con esos valores se crean los campos o atributos del objeto. El bloque de acciones del constructor tiene las instrucciones:

```
self.legajo = leg
self.nombre = nom
self.direccion = direc
self.sueldo = suel
self.antiguedad = ant
```

lo cual significa que para el objeto *self* se está creando el campo self.legajo con el valor del parámetro formal *leg*, y algo similar para los campos o atributos self.nombre, self.direccion, self.sueldo y self.antiguedad. La dirección del objeto *self* es la que automáticamente retorna el constructor *__init__()*, y esa dirección se asigna en la variable *e1* que se mostró en el ejemplo.

Una vez que el objeto ha sido creado e inicializado de esta forma, puede ser utilizado de allí en adelante en la misma forma en que hemos visto en esta Ficha: se usa el nombre de la variable seguido de un punto y luego el nombre del atributo que se quiere acceder, como se ve en el ejemplo que sigue:

```
# creación e inicialización de objetos de tipo Empleado...
# ... usando el constructor...
e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
e2 = Empleado(2, 'Luis', 'Calle 2', 20000, 15)
e3 = Empleado(3, 'Pedro', 'Calle 3', 25000, 20)
e4 = Empleado(4)

# acceso a atributos...
e4.nombre = "Ana"
e4.sueldo = e3.sueldo + 10000
e4.direccion = 'Calle 4'
e4.antiguedad += 1
```

Y de aquí en más, se esperaría que las que antes eran funciones separadas para procesar un objeto, ahora sean métodos incluidos dentro la clase que describe a los objetos. En particular, en los modelos que hemos visto para crear registros que representen Empleados contábamos con una función muy simple llamada *write()* que permitía mostrar por pantalla el contenido de un registro tomado como parámetro. Mantendremos esa función en nuestra clase Empleado, por ahora la convertiremos en un método:

```
class Empleado:
    # un constructor...
    def __init__(self, leg=0, nom='No sabe', direc='No sabe', suel=0, ant=1):
        self.legajo = leg
        self.nombre = nom
        self.direccion = direc
        self.sueldo = suel
        self.antiguedad = ant

    # un método para mostrar un registro de tipo Empleado...
    def write(self):
        print("\nLegajo:", self.legajo, end=' ')
        print("- Nombre:", self.nombre, end=' ')
        print("- Direccion:", self.direccion, end=' ')
        print("- Sueldo:", self.sueldo, end=' ')
        print("- Antiguedad:", self.antiguedad, end=' ')
```

La vieja función write() está ahora indentada dentro de la clase Empleado. Eso la convierte en un método de la clase, y como dijimos, todo método en Python debe tomar un parámetro

self que referencie al objeto que invocó al método y sobre el cual ese método se aplicará. Y es a través de *self* que se accede a los campos del objeto que se está procesando.

El detalle final es que como ahora write() es un método, todo objeto de la clase Empleado dispone en su interior de ese método, del mismo modo que dispone de los atributos. Y para invocar a ese método, se sigue la misma técnica que para acceder a un atributo: se escribe el nombre de la variable u objeto para el cual se quiere aplicar ese método, seguido de un punto y del nombre del método a activar, enviando los parámetros que el método pida:

```
def test():
    # creación e inicialización de variables de tipo Empleado...
    # ... usando el constructor...
    e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
    e2 = Empleado(2, 'Luis', 'Calle 2', 20000, 15)
    e3 = Empleado(3, 'Pedro', 'Calle 3', 25000, 20)
    e4 = Empleado(4)

    # acceso a atributos...
    e4.nombre = "Ana"
    e4.sueldo = e3.sueldo + 10000
    e4.direccion = 'Calle 4'
    e4.antiguedad += 1

    # visualización de los valores de los tres registros...
    e1.write()
    e2.write()
    e3.write()
    e4.write()
```

Recordemos que el parámetro *self* se define en la cabecera de todo método de cualquier clase, y referencia (o apunta) al objeto que invocó al método. Pero al invocar a ese método *no debe ser enviado el objeto como parámetro actual*: Python asume que la dirección del objeto que llama al método se debe enviar *implícitamente* como parámetro para ser asignado en *self*, y eso es realizado en forma automática. Así, en la instrucción *e1.write()* es el objeto *e1* el que Python envía implícitamente para que se asigne en *self*, y al activarse el método, *self* queda apuntando al mismo objeto que apunta *e1*. Algo similar ocurrirá en el resto de las instrucciones que se ven resaltadas en verde en el ejemplo anterior.

El programa básico sobre el que estuvimos trabajando en esta sección puede verse completo en el modelo *Proyecto [F19] Registros y Objetos,* programa *test07.py* que acompaña a esta ficha.

En este momento en que ya hemos realizado una presentación elemental de los primeros pasos en la POO en Python, conviene hacer un algunas observaciones importantes para terminar de pulir conceptos:

- ✓ Como dijimos, la POO es un paradigma de programación que busca principalmente identificar *actores*, a los que designa como *objetos*, en un problema, aunque luego también deberán identificarse los *procesos* que esos objetos pueden aplicar. Las técnicas e ideas propias de la PE para identificar subproblemas y convertirlos en funciones en realidad siguen siendo válidos en la POO, pero la POO requiere un diseño más amplio de las soluciones y de alguna manera contiene a la PE.
- ✓ El paradigma de la POO se basa en ciertas propiedades muy potentes a la hora de modelar un sistema complejo. Esencialmente esas propiedades son tres: el *encapsulamiento*, la *herencia* y el *polimorfismo*. El *encapsulamiento* es la propiedad

de la POO que permite que los atributos y los métodos sean definidos *juntos* dentro del ámbito de una clase (en definitiva, es la propiedad que permite que una función se defina dentro de lo que originalmente sería un tipo registro, marcando con eso la diferencia entre *tipo registro* y *clase*). Y por otra parte, la *herencia* y el *polimorfismo* son herramientas realmente muy potentes… pero quedan fuera del alcance de este curso (que NO ES un curso de POO)… ☹.

- ✓ El paradigma de la POO es actualmente el más utilizado en programación de aplicaciones, y muchos lenguajes y plataformas (en mayor o en menor medida) dan soporte a ese paradigma. Python es uno de esos lenguajes, y es especialmente notable que casi todos los tipos de datos que Python provee listos para usar son en realidad *clases*, y que las variables que se definen en base a esos tipos son en consecuencia *objetos*… Ejemplos muy conocidos a esta altura son las clases *str*, *tuple*, *list* y *range* (por citar solo algunas de las que más hemos utilizado en las fichas de este curso).
- ✓ Cualquier variable de tipo *str* (o sea, cualquier variable de tipo cadena de carateres) en Python, *es en realidad un objeto de la clase str*. Y como tal, contiene muchos métodos listos para usar que se acceden con el operador punto. Algunos ejemplos muy conocidos son los métodos *upper()*, *lower()*, *isalpha()* o *isdigit()* que se han aplicado en problemas de tratamiento de caracteres.
- ✓ La clase *list*, por su parte, permite definir objetos que representan arreglos adaptativos (o sea, arreglos que pueden ajustar su tamaño a medida que el programador lo requiera) y también incluye numerosos métodos tales como *append()* (por citar solo al más conocido por nosotros).
- ✓ La función *open()* que usamos para abrir un archivo, abre el archivo pedido y retorna un *objeto* de una clase nativa de Python que representa al archivo que se acaba de abrir. Esa clase provee métodos como *read()*, *readline()* y *close()* (entre muchos otros) para gestionar a ese archivo (como hemos visto aunque en forma básica hasta aquí para abrir y leer archivos de texto).
- ✓ Tanto si se aplica el modelo ingenuo de definir *tipos de registros simples* como si se aplica a pleno el modelo de *clases* notemos que en ambos casos se está definiendo para ese programa una nueva palabra que tiene rango de *tipo de dato*: si se define el *tipo de registro Empleado* en forma de registro llano (sin métodos), o si se define la *clase Empleado*, esa palabra *Empleado* se usa de allí en más para definir variables (u objetos) de tipo *Empleado*. El tipo *Empleado* no existía previamente en Python como un tipo válido para definir variables, pero el programador definió el nuevo tipo y lo agregó a su programa.
- ✓ Lo dicho en el párrafo anterior refleja una situación muy común en programación: si el programador descubre que necesita ciertos tipos de datos y el lenguaje que está usando dispone de esos tipos ya listos para usar, estamos en presencia de *tipos de datos nativos* del lenguaje. Está claro que entonces en Python los tipos *int, float, bool, str, tuple*, *range, list* y muchos otros, representan *tipos nativos*.
- ✓ Pero como vimos, el programador puede necesitar algún tipo de datos que el lenguaje ***no*** provea en forma nativa. El uso de *tipos de registros simples* o de *clases* es una forma en que puede definirlos y agregarlos a sus programas. Esos nuevos tipos

se designan en general como *tipos de datos abstractos*. En el ejemplo anterior de la clase *Empleado*, desarrollada por el programador para un requerimiento específico, esa clase *Empleado* representa un *tipo abstracto*.

- ✓ El proceso llevado a cabo por el programador para crear un nuevo tipo abstracto (*Empleado* en el ejemplo del párrafo anterior), se conoce con el nombre general de *implementación* del tipo abstracto. En ese proceso el programador lleva a cabo un mecanismo conceptual de identificación de datos y procesos que se suele designar como *mecanismo de abstracción*, que a su vez consta de dos elementos: el *mecanismo de abstracción de datos* (por el cual el programador identifica cuáles son *los campos o atributos* que debe contener una clase), y el *mecanismo de abstracción funcional* (mediante el cual el programador identifica cuáles son *las funciones o métodos* que debe contener la clase). Volveremos sobre este tema en una ficha posterior de este mismo curso.

5.] Ejemplo de aplicación.

Para cerrar esta introducción con un ejemplo más profundo, veremos ahora nuevamente el problema de la representación de puntos en un plano, pero ahora aplicando POO en lugar de registros simples.

Problema 46.) *Rediseñe el programa planteado para el problema 45 (que pedía representar puntos en un plano), pero ahora desde la perspectiva de las ideas básicas de la POO que se expusieron en esta sección.*

Discusión y solución: (La solución puede verse completa en el *Proyecto [F19] Registros y Objetos,* subcarpeta *test08*).

Estrictamente hablando, lo único que tenemos que hacer es convertir lo hecho en el programa original (*Proyecto [F19] Registros y Objetos,* subcarpeta *test06*) para que ahora la clase *Point* represente una *clase* acorde a los elementos presentados aquí, y no un *tipo de registro simple*.

Dentro del proyecto *[F19] Registros y Objetos*, subcarpeta *test08*, el módulo *geometry.py* contiene ahora la declaración de la clase *Point* con el siguiente constructor:

```
class Point:
    def __init__(self, cx, cy, desc='p'):
        self.x = cx
        self.y = cy
        self.descripcion = desc
```

El constructor *__init__(self, cx, cy, desc='p')* crea los campos del objeto *self* que toma automáticamente como parámetro. El parámetro formal *desc* tiene por defecto asignada la cadena 'p', por lo cual si ese parámetro se omite al invocar al constructor, el campo *descripcion* de *point* quedará valiendo 'p'.

El mismo módulo *geometry.py* en esa misma carpeta contiene también el resto de la funciones originales, pero ahora definidas como métodos, con una notable diferencia (que a continuación se explicará) referida al método *__str__()*:

```
import math


class Point:
    def __init__(self, cx, cy, desc='p'):
        self.x = cx
        self.y = cy
        self.descripcion = desc

    def __str__(self):
        r = str(self.descripcion) + '(' + str(self.x) + ', ' + str(self.y) + ')'
        return r

    def distance(self):
        # Pitágoras...
        return math.sqrt(pow(self.x, 2) + pow(self.y, 2))

    def gradient(self, p2):
        # calcular "delta y" y "delta x"
        dy = p2.y - self.y
        dx = p2.x - self.x

        # si los puntos no son colineales verticales,
        # retornar la pendiente...
        if dx != 0:
            return dy / dx

        # de otro modo, la pendiente es indefinida...
        # ... este retorno debería ser controlado al regresar...
        return None
```

En el modelo original (basado en registros simples) contábamos con una función *to_string()* que creaba una cadena de caracteres a partir de un registro tomado como parámetro, y retornaba esa cadena lista para ser mostrada en consola si fuese necesario. Esta era la función original:

```
def to_string(point):
    r = str(point.descripcion) + '(' + str(point.x) + ',' + str(point.y) + ')'
    return r
```

Pero ahora, en nuestra nueva versión de la clase *Point*, esa función es un *método* (está dentro de la clase y recibe el parámetro implícito *self*), y ha cambiado de nombre: en lugar de llamarse *to_string()*, ahora se llama *__str__()*, y tiene el siguiente aspecto:

```
def __str__(self):
    r = str(self.descripcion) + '(' + str(self.x) + ', ' + str(self.y) + ')'
    return r
```

En Python existen ciertos métodos especiales dentro de una clase cualquiera (uno de ellos es el *constructor*) que se caracterizan por la presencia de los dos guiones bajos adelante y detrás del identificador.

Esos métodos normalmente llevan a cabo alguna tarea estándar y Python los trata de forma especial en un programa. Por ejemplo, un constructor crea e inicializa un objeto, y Python (como vimos) asume una forma especial de invocarlo y de obtener el objeto creado. Y en particular el método *__str__()* tiene la tarea de obtener y retornar una cadena con el contenido del objeto *self*. Pero si ya hacía eso nuestra función *to_string()* original, entonces ¿porqué el cambio de nombre? El tema es que Python asume *automáticamente* que se está invocando específicamente a *ese* método *__str__()* en cualquier contexto en el cual se requiera convertir un objeto a una cadena. Eso significa (por ejemplo) que en el siguiente script:

```
p = Point(2, 7)
print(p.__str__())
```

la instrucción *print()* en la que muestra el punto convertido a cadena también puede escribirse así, y el resultado será exactamente el mismo:

```
print(p)
```

Vale decir: si la clase contiene el método *__str__()* convenientemente programado para obtener una cadena a partir del objeto que invoca al método, entonces no necesita invocar en forma explícita a ese método cuando pida la conversión a cadena de ese objeto (por ejemplo en un *print()* o al invocar a la función *str()*: solo nombre a la variable con la que se referencia al objeto, y asegúrese que el contexto efectivamente necesita convertir el objeto a una cadena. El ejemplo adicional que sigue:

```
p = Point(2, 7)
r = "Punto: " + str(p.__str__())
print(r)
```

es equivalente a:

```
p = Point(2, 7)
r = "Punto: " + str(p)
print(r)
```

Siguiendo con la clase Point, el cálculo de la pendiente de la recta que une a dos puntos *p1* y *p2*, se realizaba con la función *gradient(p1, p2)* que originalmente era la siguiente:

```
def gradient(p1, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - p1.y
    dx = p2.x - p1.x

    # si los puntos no forman una recta vertical,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    return None
```

Pero en nuestra nueva versión, la función *gradient()* es un método de la clase *Point*, y el objeto (o punto) con el que se llama al método entrará implícitamente como parámetro en *self*. El segundo punto (que sería *p2* en este caso), sí necesita ser tomado como parámetro explícito:

```
def gradient(self, p2):
    # calcular "delta y" y "delta x"
    dy = p2.y - self.y
    dx = p2.x - self.x

    # si los puntos no son colineales verticales,
    # retornar la pendiente...
    if dx != 0:
        return dy / dx

    # de otro modo, la pendiente es indefinida...
    # ... este retorno debería ser controlado al regresar...
    return None
```

Y de este modo, en el siguiente script:

```
p1 = Point(5, 8)
p2 = Point(2, 4)
pd = p1.gradient(p2)
print(pd)
```

se está usando el objeto *p1* para invocar al método, y por lo tanto *p1* entrará al método asignándose en *self* (dicho de otro modo, dentro del método el objeto *p1* se llama *self*). El objeto *p2* es (como se ve) el único que debe ser enviado explícitamente entre los paréntesis del método al invocarlo: como p1 es el que invoca al método, Python lo toma como self y no debemos hacer más.

La función *distance()* original, ahora es también un método, pero no presenta mayor dificultad y dejamos su análisis para el lector.

Asumiendo que el módulo *geometry.py* existe y que es el mismo que presentamos un poco más arriba con la clase *Point* entonces el programa completo es el siguiente (ver tanto el módulo como el programa en el proyecto *[F19] Registros y Objetos*, subcarpeta *test08*):

```
import geometry


def opcion1():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = geometry.Point(cx, cy)
    print(p)


def opcion2():
    cx = float(input('Coordenada x: '))
    cy = float(input('Coordenada y: '))
    p = geometry.Point(cx, cy)
    d = p.distance()
    print('Distancia al origen:', d)


def opcion3():
    cx1 = float(input('Punto 1 - Coordenada x: '))
    cy1 = float(input('Punto 1 - Coordenada y: '))
    p1 = geometry.Point(cx1, cy1)

    cx2 = float(input('Punto 2 - Coordenada x: '))
    cy2 = float(input('Punto 2 - Coordenada y: '))
    p2 = geometry.Point(cx2, cy2)

    pd = p1.gradient(p2)
    if pd is not None:
        print('Pendiente de la recta que los une:', pd)
    else:
        print('Pendiente no definida (recta vertical)')

def menu():
    op = -1
    while op != 4:
        print('1. Cargar y mostrar un punto')
        print('2. Distancia al origen')
        print('3. Pendiente de la recta que une dos puntos')
        print('4. Salir')

        op = int(input('Ingrese opcion: '))
```

```
            if op == 1:
                opcion1()
            elif op == 2:
                opcion2()
            elif op == 3:
                opcion3()


if __name__ == '__main__':
    menu()
```

Para finalizar, note el lector que la clase *Point* (y también el tipo de registro original *Point* que fue nuestro primer diseño) constituye un *tipo abstracto* para nuestro programa (es un tipo que en principio el lenguaje no provee ya listo para usar. Y si lo provee, el programador decidió implementrar SU propia versión). El *mecanismo de abstracción de datos* permitió identificar los *atributos* que un *Point* necesita (*x*, *y*, *descripcion*), y el *mecanismo de abstracción funcional* permitió identificar los *métodos* que finalmente se implementaron (*__init__()*, *__str__()*, *gradient()* y *distance()*).

Bibliografía

[1] Python Software Foundation, "Python Documentation," 2021. [Online]. Available: https://docs.python.org/3/.

[2] M. Pilgrim, "Dive Into Python - Python from novice to pro", Nueva York: Apress, 2004.

[3] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.