**9.2 Clases**

Esta sección tiene un [video](https://youtu.be/h7X8rrW8JLo) donde introducimos las clases, damos algunos ejemplos y también introducimos el tema de la [Sección 9.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/04_M%C3%A9todos_Especiales.md#94-m%C3%A9todos-especiales). En este video, y en el que veremos más adelante sobre herencia, usamos el código de [este archivo](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/clase9_clases_metesp.py).

La programación orientada a objetos requiere un pequeño pero importante cambio en la forma de pensar la programación tradicional. Dejá decantar los conceptos nuevos mientras leés esta sección.

En esta sección veremos el concepto de clase, cómo crear nuevos tipos de objetos, su utilidad, y las ventajas de esa forma de organizar los programas.

**Programación orientada a objetos (POO)**

La programación orientada a objetos es una forma de organizar el código. Así como un algoritmo suele estar asociado a una estructura de datos particular, la programación orientada a objetos "empaqueta" los datos junto con los métodos usados para tratarlos.

Cada uno de esos *objetos* consiste en

* Datos (atributos de los objetos).
* Comportamiento (métodos de los objetos: son funciones que actúan sobre los atributos del objeto).

Ya usaste objetos durante el curso infinidad de veces. Por ejemplo, al manipular una lista.

>>> nums = [1, 2, 3]

>>> nums.append(4) # Esto es un método de la lista

>>> nums.insert(1,10) # Otro método

>>> nums

[1, 10, 2, 3, 4] # Estos son los datos modificados por los métodos

>>>

Miremos un poco más en detalle este fragmento de código. Sabemos que nums es una variable de tipo lista. Equivalentemente, podemos decir que nums es una *instancia* de la clase *list*. Cada variable de tipo lista es una instancia de la misma clase. Al hablar de 'instancia' nos referimos a un 'objeto': un objeto es una instancia de una clase.

Un objeto de tipo lista tiene atributos (datos) y métodos. Los métodos, como append() o insert(), se definen cuando se define la clase, pero se usan para manipular los datos de un objeto concreto (nums en este caso).

**La instrucción class**

Para definir un tipo nuevo de objeto, usá la instrucción class.

class Jugador:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

self.salud = 100

def mover(self, dx, dy):

self.x += dx

self.y += dy

def lastimar(self, pts):

self.salud -= pts

Un objeto de tipo Jugador tiene como atributos x, y y salud. Sus métodos son mover() y lastimar().

Puede decirse que una clase es la definición formal de las relaciones entre los datos y los métodos que los manipulan. Un objeto es una instancia particular de la clase a la cual pertenece, con datos propios pero los mismos métodos que los demás objetos de esa clase. Este concepto te va a quedar más claro cuando lo veas funcionar y lo uses.

**Instancias**

Los programas manipulan instancias individuales de las clases. Cada instancia es un objeto, y es en cada objeto que uno puede manipular los datos y llamar a sus métodos.

Podés crear un objeto mediante un llamado a la clase como si fuera una función.

>>> a = Jugador(2, 3) # Clase Jugador definida antes

>>> b = Jugador(10, 20)

>>>

a y b son instancias de Jugador definida más arriba. Es decir, a y b son objetos de la clase Jugador.

*Importante: La instrucción class es solamente la definición de una clase, no****hace****nada por sí misma. Es similar a la definición de una función.*

**Datos de una instancia**

Cada instancia tiene sus propios datos locales. Acá pedimos ver el atributo x de cada instancia:

>>> a.x

2

>>> b.x

10

Estos datos locales se inicializan, para cada instancia, durante la ejecución del método \_\_init\_\_() de la clase.

class Jugador:

def \_\_init\_\_(self, x, y):

# Todo dato guardado en `self` es propio de esa instancia

self.x = x

self.y = y

self.salud = 100

No hay restricciones en la cantidad o el tipo de atributos que puede tener una clase.

**Métodos de una instancia.**

Los métodos de una instancia son los métodos y las funciones que actúan sobre los datos almacenados en esa instancia.

class Jugador:

...

# `mover` es un método

def mover(self, dx, dy):

self.x += dx

self.y += dy

Siempre se recibe la instancia misma como primer argumento: "self" significa "mismo" como en "mi mismo" ó "en sí misma". Es como decir "yo".

>>> a.mover(1, 2)

# `self` refiere a `a`

# `dx` refiere a `1`

# `dy` refiere a `2`

def mover(self, dx, dy):

...

Por convención siempre llamamos self a la instancia actual, y ésta es siempre pasada como primer argumento a todos los métodos. En realidad el nombre real de la variable no importa, pero es una convención en Python llamar al primer argumento self.

Podríamos usar mismo, por ejemplo, en lugar de self y todo va a funcionar igual, pero no respeta las convenciones de la comunidad:

class Jugador:

...

# `mover` es un método

def mover(mismo, dx, dy):

mismo.x += dx

mismo.y += dy

**Visibilidad en clases (Scoping)**

Las clases no definen ni limitan (como los módulos) un entorno de visibilidad.

class Jugador:

...

def mover(self, dx, dy):

self.x += dx

self.y += dy

def izquierda(self, dist):

mover(-dist, 0) # NO! Refiere a una función global `mover`.

self.mover(-dist, 0) # Sí. Llama al método `mover` definido antes.

Si necesitás referirte a un dato o un método de una clase tenés que hacer una referencia explícita (agregando el self), sino te estás refiriendo a otra cosa como en el ejemplo anterior.

**Ejercicios**

Vamos a comenzar esta serie de ejercicios modificando código que escribiste antes del parcial. En particular retomaremos el código del [Ejercicio 7.5](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/07_Plt_Especificacion_y_Documentacion/04_Flexibilidad.md#ejercicio-75-arreglemos-las-funciones-existentes). Te dejamos [acá](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/ejs.zip) una versión funcionando que podés mirar y/o usar. Tiene cosas interesantes, aunque tengas la tuya funcionando si querés pegale una mirada.

**Ejercicio 9.1: Objetos como estructura de datos.**

Durante las primeras clases trabajamos con datos en forma de tuplas y diccionarios. Un lote con cajones de frutas, por ejemplo, estaba representado por una tupla, como ésta:

s = ('Pera', 100, 490.10)

o por un diccionario, de esta otra forma:

s = { 'nombre' : 'Pera',

'cajones' : 100,

'precio' : 490.10

}

Incluso escribiste funciones para manipular datos almacenados de ese modo:

def costo(registro):

return registro['cajones'] \* registro['precio']

Otra forma de representar los datos con los que estás trabajando es definir una clase. Creá un archivo llamado lote.py y adentro definí una clase llamada Lote que represente un lote de cajones de una misma fruta. Definila de modo que cada instancia de la clase Lote (es decir, cada objeto lote) tenga los atributos nombre, cajones, y precio. Éste es un ejemplo del comportamiento buscado:

>>> import lote

>>> a = lote.Lote('Pera', 100, 490.10)

>>> a.nombre

'Pera'

>>> a.cajones

100

>>> a.precio

490.1

>>>

Vamos a crear más objetos de tipo Lote para manipularlos. Por ejemplo:

>>> b = lote.Lote('Manzana', 50, 122.34)

>>> c = lote.Lote('Naranja', 75, 91.75)

>>> b.cajones \* b.precio

6117.0

>>> c.cajones \* c.precio

6881.25

>>> lotes = [a, b, c]

>>> lotes

[<lote.Lote object at 0x37d0b0>, <lote.Lote object at 0x37d110>, <lote.Lote object at 0x37d050>]

>>> for c in lotes:

print(f'{c.nombre:>10s} {c.cajones:>10d} {c.precio:>10.2f}')

... mirá el resultado ...

>>>

Fijate que la clase Lote funciona como una "fábrica" para crear objetos que son instancias de esa clase. Vos la llamás como si fuera una función y te crea una nueva instancia de sí misma. Más aún, cada instancia es única y tiene sus propios datos que son independientes de las demás instancias de la misma clase.

Una instancia definida por una clase puede tener cierta similitud con un diccionario, pero usa una sintaxis algo diferente. Por ejemplo, en lugar de escribir c['nombre'] ó c['precio'] en objetos escribís c.nombre o c.precio.

**Ejercicio 9.2: Agregá algunos métodos**

Al definir una clase podés agregar funciones a los objetos que definís. Las funciones específicas de objetos se llaman *métodos* y operan sobre los datos guardados en cada instancia.

Agregá los métodos costo() y vender() a tu objeto Lote. Deberían dar este comportamiento:

>>> import lote

>>> s = lote.Lote('Pera', 100, 490.10)

>>> s.costo()

49010.0

>>> s.cajones

100

>>> s.vender(25)

>>> s.cajones

75

>>> s.costo()

36757.5

>>>

**Ejercicio 9.3: Lista de instancias**

Seguí estos pasos para crear una lista de las instancias de Lote (una lista de objetos Lote) a partir de una lista de diccionarios. Luego calculá el precio total de todas esas instancias.

>>> import fileparse

>>> with open('../Data/camion.csv') as lineas:

... camion\_dicts = fileparse.parse\_csv(lineas, select = ['nombre', 'cajones', 'precio'], types = [str, int, float])

...

>>> camion = [ lote.Lote(d['nombre'], d['cajones'], d['precio']) for d in camion\_dicts]

>>> camion

[<lote.Lote object at 0x10c9e2128>, <lote.Lote object at 0x10c9e2048>, <lote.Lote object at 0x10c9e2080>,

<lote.Lote object at 0x10c9e25f8>, <lote.Lote object at 0x10c9e2630>, <lote.Lote object at 0x10ca6f748>,

<lote.Lote object at 0x10ca6f7b8>]

>>> sum([c.costo() for c in camion])

47671.15

>>>

**Ejercicio 9.4: Usá tu clase**

Modificá la función leer\_camion() en el programa informe.py de modo que lea un archivo con el contenido de un camion y devuelva una lista de instancias de Lote como mostramos recién en el [Ejercicio 9.3](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/02_Clases.md#ejercicio-93-lista-de-instancias).

Cuando hayas hecho esto, cambiá un poco el código en informe.py y en costo\_camion.py de modo que funcionen con objetos Lote (instancias de la clase Lote) en lugar de diccionarios.

Ayuda: No deberían ser cambios importantes. Las referencias a diccionarios ahora tienen que hacer referencia a objetos (c['cajones'] cambia a c.cajones).

Hecho esto, deberías poder ejecutar tus funciones como antes:

>>> import costo\_camion

>>> costo\_camion.costo\_camion('../Data/camion.csv')

47671.15

>>> import informe

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv')

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 $32.2 8.02

Naranja 50 $91.1 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.1 15.79

Naranja 100 $70.44 35.84

**9.3 Herencia**

Esta sección tiene [un video](https://youtu.be/n7CxhdOmHb8) donde introducimos el tema de herencia.

La herencia entre clases es una herramienta muy usada para escribir programas extensibles. Exploraremos esta idea a continuación.

**Introducción**

Se usa herencia para crear objetos más especializados a partir de objetos existentes.

class Padre:

...

class Hijo(Padre):

...

Se dice que Hijo es una clase derivada o subclase. La clase Padre es conocida como la clase base, o superclase. La expresión class Hijo(Padre): significa que estamos creando una clase llamada Hijo que es derivada de la clase Padre.

**Extensiones**

Al usar herencia podés tomar una clase existente y ...

* Agregarle métodos
* Redifinir métodos existentes
* Agregar nuevos atributos

Podés verlo como una forma de **extender** de tu codigo existente. Darle nuevos comportamientos, abarcar un abanico más amplio de posibilidades ó aumentar su compatibilidad.

**Ejemplo**

Suponé que partís de la siguiente clase:

class Lote:

def \_\_init\_\_(self, nombre, cajones, precio):

self.nombre = nombre

self.cajones = cajones

self.precio = precio

def costo(self):

return self.cajones \* self.precio

def vender(self, ncajones):

self.cajones -= ncajones

Podés modificar lo que necesites mediante herencia.

**Agregar un método nuevo**

class MiLote(Lote):

def rematar(self):

self.vender(self.cajones)

Se puede usar así:

>>> c = MiLote('Pera', 100, 490.1)

>>> c.vender(25)

>>> c.cajones

75

>>> c.rematar()

>>> c.cajones

0

>>>

Esta clase heredó los atributos y métodos de Lote y la extendío con un nuevo método (rematar()).

**Redefinir un método existente**

class MiLote(Lote):

def costo(self):

return 1.25 \* self.cajones \* self.precio

Un ejemplo de uso:

>>> c = MiLote('Pera', 100, 490.1)

>>> c.costo()

61262.5

>>>

El método nuevo simplemente reemplaza al definido en la clase base. Los demás métodos y atributos no son afectados. ¿No es buenísimo?

**Utilizar un método prevalente**

Hay veces en que una clase extiende el método de la superclase a la que pertenece, pero necesita ejecutar el método original como parte de la redefinición del método nuevo. Para referirte a la superclase, usá super():

class Lote:

...

def costo(self):

return self.cajones \* self.precio

...

class MiLote(Lote):

def costo(self):

# Fijate cómo usamos `super`

costo\_orig = super().costo()

return 1.25 \* costo\_orig

Usá super() para llamar al método de la clase base (del la cual ésta es heredera).

**El método \_\_init\_\_ y herencia.**

Al crear cada instancia se ejecuta \_\_init\_\_. Ahí reside el código importante para la creación de una instancia nueva. Si redefinís \_\_init\_\_ siempre incluí un llamado al método \_\_init\_\_ de la clase base para inicializarla también.

class Lote:

def \_\_init\_\_(self, nombre, cajones, precio):

self.nombre = nombre

self.cajones = cajones

self.precio = precio

...

class MiLote(Lote):

def \_\_init\_\_(self, nombre, cajones, precio, factor):

# Fijate como es el llamado a `super().\_\_init\_\_()`

super().\_\_init\_\_(nombre, cajones, precio)

self.factor = factor

def costo(self):

return self.factor \* super().costo()

Es necesario llamar al método \_\_init\_\_() en la clase base. Es una forma de ejecutar la versión previa del método que estamos redefiniendo, como mostramos recién.

**Usos de herencia**

Uno de los usos de definir una clase como heredera de otra es organizar jerárquicamente objetos que están relacionados.

Un ejemplo: Las figuras geométricas pueden tener ciertos métodos y atributos que luego son refinados en casos concretos como círculos o rectángulos.

class FiguraGeom:

...

class Circulo(FiguraGeom):

...

class Rectangulo(FiguraGeom):

...

Imaginate por ejemplo su uso en una jerarquía lógica, o taxonómica, en la que las clases tienen una relación natural tal que hace intuitivo derivar una de otra.

Una aplicación más común, y tal vez más práctica, consiste en escribir código que es reutilizable y/o extensible. Podríamos definir una clase base para una interfaz de transferencia de datos y permitir que cada fabricante de equipo de adquisición de datos implemente los detalles de comunicación con cada interfaz en particular.

class Procesador\_de\_datos(TCPHandler):

def procesar\_pedido(self):

...

# Procesamiento de datos

La clase base contiene código de administración no específico. Cada clase hereda ese código y modifica las partes necesarias.

**Relación "isinstance"**

La herencia establece una relación de clases.

class FiguraGeom:

...

class Circulo(FiguraGeom):

Preguntamos si un objeto es una instancia de cierta clase:

>>> f = Circulo(4.0)

>>> isinstance(f, FiguraGeom)

True

>>>

*Importante: Idealmente, todo código que funcione con instancias de una clase base debería también funcionar con instancias de las clases derivadas de ella.*

**La clase base object**

Si una clase no tiene superclase, a veces se escribe object como clase base.

class Figura\_geom(object):

...

object es la superclase de todo objeto en Python.

**Herencia múltiple.**

Podés heredar de varias clases simultáneamente si los especificás en la definición de clase.

class Madre:

...

class Padre:

...

class Hijo(Madre, Padre):

...

La clase Hijo hereda características de ambos padres. Algunos detalles son un poco delicados y no vamos a usar esa forma de heredar clases en este curso, aunque vas a encontrar un poco más de información en la próxima sección.

**Ejercicios**

El concepto de herencia es especialmente útil cuando estás escribiendo código que va a ser extendido o adaptado, ya sea en bibliotecas o grandes sistemas configurables, pero también en pequeños paquetes de procesamiento de datos que pueden adquirir datos de diversas fuentes. Como ya dijimos antes, uno puede escribir las relaciones y comportamientos fundamentales y dejar los detalles de implementación de cada interfaz para cuando sean necesarios.

Para verlo mejor volvamos a la función imprimir\_informe() del [Ejercicio 6.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/02_Scripts.md#ejercicio-64-estructurar-un-programa-como-una-colecci%C3%B3n-de-funciones), que figura en el programa informe.py. Tenía más o menos este aspecto:

def imprimir\_informe(data\_informe):

'''

Imprime una tabla prolija desde una lista de tuplas con (nombre, cajones, precio, cambio)

'''

headers = ('Nombre','Cajones','Precio','Cambio')

print('%10s %10s %10s %10s' % headers)

print(('-'\*10 + ' ')\*len(headers))

for row in data\_informe:

print('%10s %10d %10.2f %10.2f' % row)

Al ejecutar tu programa informe la salida es algo parecido a esto:

>>> import informe

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv')

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 $32.2 8.02

Naranja 50 $91.1 15.18

Caqui 150 $103.44 2.02

Mandarina 200 $51.23 29.66

Durazno 95 $40.37 33.11

Mandarina 50 $65.1 15.79

Naranja 100 $70.44 35.84

A continuación vamos a trabajar con herencias relacionadas con este código.

**Ejercicio 9.5: Un problema de extensibilidad**

Imaginá que necesitás que la función imprimir\_informe() pueda exportar el informe en una variedad de formatos: texto plano, HTML, CSV ó XML. Podrías escribir una función enorme que resuelva todos los casos, pero resultaría en código repetido, y difícil de mantener. Esta es una oportunidad perfecta para usar herencia de objetos.

Vamos a enfocarnos en los pasos necesarios para crear una tabla.

Al principio de la tabla tenemos los encabezados de las columnas. Después de eso, los datos de la tabla ordenados en una fila por ítem. Pongamos cada uno de esos pasos en una clase distinta. Creá un archivo llamado formato\_tabla.py y definí la siguiente clase:

# formato\_tabla.py

class FormatoTabla:

def encabezado(self, headers):

'''

Crea el encabezado de la tabla.

'''

raise NotImplementedError()

def fila(self, rowdata):

'''

Crea una única fila de datos de la tabla.

'''

raise NotImplementedError()

Por ahora la clase no hace nada, pero sirve como una especie de especificación de diseño para otras clases que vamos a definir. Una clase como ésta es a menudo llamada "clase base abstracta".

Ahora es necesario modificar la función imprimir\_informe() para que acepte como fuente de datos un objeto FormatoTabla e invoque los métodos de este objeto para producir la tabla de salida. Algo así:

# informe.py

import formato\_tabla

...

def imprimir\_informe(data\_informe, formateador):

'''

Imprime una tabla prolija desde una lista de tuplas

con (nombre, cajones, precio, diferencia)

'''

formateador.encabezado(['Nombre', 'Cantidad', 'Precio', 'Cambio'])

for nombre, cajones, precio, cambio in data\_informe:

rowdata = [ nombre, str(cajones), f'{precio:0.2f}', f'{cambio:0.2f}' ]

formateador.fila(rowdata)

Como agregaste un argumento a imprimir\_informe(), hay que modificar también informe\_camion(). Cambialo para que cree un objeto formateador de este modo:

# informe.py

import formato\_tabla

...

def informe\_camion(archivo\_camion, archivo\_precios):

'''

Crea un informe a partir de un archivo de camión

y otro de precios de venta.

'''

# Leer archivos con datos

camion = leer\_camion(archivo\_camion)

precios = leer\_precios(archivo\_precios)

# Crear los datos para el informe

data\_informe = hacer\_informe(camion, precios)

# Imprimir el informe

formateador = formato\_tabla.FormatoTabla()

imprimir\_informe(data\_informe, formateador)

Ejecutá este código:

>>> ================================ RESTART ================================

>>> import informe

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv')

... crashes ...

Debería dar inmediatamente una excepción de tipo NotImplementedError. No es nada maravilloso, pero es exactamente lo que esperábamos que sucediera, ¿no? Sigamos...

**Ejercicio 9.6: Usemos herencia para cambiar la salida**

La clase FormatoTabla que definiste en la primera parte es sólo la base de un sistema extensible. Éste es el momento de extenderla. Definí una clase FormatoTablaTXT como sigue:

# formato\_tabla.py

...

class FormatoTablaTXT(FormatoTabla):

'''

Generar una tabla en formato TXT

'''

def encabezado(self, headers):

for h in headers:

print(f'{h:>10s}', end=' ')

print()

print(('-'\*10 + ' ')\*len(headers))

def fila(self, data\_fila):

for d in data\_fila:

print(f'{d:>10s}', end=' ')

print()

Modificá la función informe\_camion() y probala:

# informe.py

...

def informe\_camion(archivo\_camion, archivo\_precios):

'''

Crea un informe por camion a partir de archivos camion y precio.

'''

# Leer archivos con datos

camion = leer\_camion(archivo\_camion)

precios = leer\_precios(archivo\_precios)

# Obtener los datos para un informe

data\_informe = hacer\_informe(camion, precios)

# Imprimir

formateador = formato\_tabla.FormatoTablaTXT()

imprimir\_informe(data\_informe, formateador)

Este código debería dar la misma salida que antes:

>>> ========================REINICIAR INTERPRETE========================

>>> import informe

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv')

Nombre Cantidad Precio Cambio

---------- ---------- ---------- ----------

Lima 100 32.20 8.02

Naranja 50 91.10 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.10 15.79

Naranja 100 70.44 35.84

>>>

Ahora probemos otras variantes. Definí, para empezar, una nueva clase llamada FormatoTablaCSV que genere la salida en formato CSV:

# formato\_tabla.py

...

class FormatoTablaCSV(FormatoTabla):

'''

Generar una tabla en formato CSV

'''

def encabezado(self, headers):

print(','.join(headers))

def fila(self, data\_fila):

print(','.join(data\_fila))

Modificá tu programa informe.py de este modo:

def informe\_camion(archivo\_camion, archivo\_precios):

'''

Crea un informe por camion a partir de archivos camion y precio.

'''

# Leer archivos con datos

camion = leer\_camion(archivo\_camion)

precios = leer\_precios(archivo\_precios)

# Obtener los datos para un informe

data\_informe = hacer\_informe(camion, precios)

# Imprimir

formateador = formato\_tabla.FormatoTablaCSV()

imprimir\_informe(data\_informe, formateador)

Ahora la salida debería tener este aspecto:

>>> ========================REINICIAR INTERPRETE========================

>>> import informe

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv')

Nombre,Cantidad,Precio,Cambio

Lima,100,32.20,8.02

Naranja,50,91.10,15.18

Caqui,150,103.44,2.02

Mandarina,200,51.23,29.66

Durazno,95,40.37,33.11

Mandarina,50,65.10,15.79

Naranja,100,70.44,35.84

Usando las mismas ideas creá la clase FormatoTablaHTML que produzca un tabla de la siguiente forma:

<tr><th>Nombre</th><th>Cajones</th><th>Precio</th><th>Cambio</th></tr>

<tr><td>Lima</td><td>100</td><td>32.20</td><td>8.02</td></tr>

<tr><td>Naranja</td><td>50</td><td>91.10</td><td>15.18</td></tr>

<tr><td>Caqui</td><td>150</td><td>103.44</td><td>2.02</td></tr>

<tr><td>Mandarina</td><td>200</td><td>51.23</td><td>29.66</td></tr>

<tr><td>Durazno</td><td>95</td><td>40.37</td><td>33.11</td></tr>

<tr><td>Mandarina</td><td>50</td><td>65.10</td><td>15.79</td></tr>

<tr><td>Naranja</td><td>100</td><td>70.44</td><td>35.84</td></tr>

Para testear tu código, modificá el programa principal de modo que use un objeto de la clase FormatoTablaHTML en lugar de uno de la clase FormatoTablaCSV para darle formato a la tabla de salida. Fijate lo fácil que es cambiar el comportamiento de un programa cuando tenés objetos que son compatibles entre sí.

**Ejercicio 9.7: Polimorfismo en acción**

Una de las grandes ventajas de la programación orientada a objetos es que podés cambiar un objeto por otro compatible y tu programa va a funcionar sin necesidad de adaptar el código que usa esos objetos.

Si escribiste un programa diseñado para usar un objeto de la clase FormatoTabla, va a funcionar sin importar *qué* objeto de esa clase uses. A este comportamiento particular se lo llama polimorfismo. Está relacionado con la capacidad de usar la misma interfaz con diferentes objetos de la misma clase, haciendo que el programa como un todo se porte distinto.

Ahora bien, un potencial problema es cómo diseñar tu programa de manera que el usuarie final pueda elegir el formato. Usar los nombres de las clases de formateadores no resultaría cómodo. Una solución posible es considerar un condicional:

def informe\_camion(archivo\_camion, archivo\_precios, fmt = 'txt'):

'''

Crea un informe con la carga de un camión

a partir de archivos camion y precio.

El formato predeterminado de la salida es txt

Alternativas: csv o html

'''

# Leer archivos con datos

camion = leer\_camion(archivo\_camion)

precios = leer\_precios(archivo\_precios)

# Obtener los datos para un informe

data\_informe = hacer\_informe(camion, precios)

# Elige formato

if fmt == 'txt':

formateador = formato\_tabla.FormatoTablaTXT()

elif fmt == 'csv':

formateador = formato\_tabla.FormatoTablaCSV()

elif fmt == 'html':

formateador = formato\_tabla.FormatoTablaHTML()

else:

raise RuntimeError(f'Unknown format {fmt}')

imprimir\_informe(data\_informe, formateador)

En este código, el usuarie especifica un nombre simplificado como txt o csv para elegir el formato. Pero bancá. ¿Es una buena idea poner un gran bloque if en la función informe\_camion()? ¿O quizás sería mejor ponerla directamente en una función de propósito general en otro lado?

En el archivo formato\_tabla.py, agregá la función crear\_formateador(nombre) que permita crear un objeto formateador dado un tipo de salida como txt, csv, o html. Modificá informe\_camion() para que se vea así:

def informe\_camion(archivo\_camion, archivo\_precios, fmt = 'txt'):

'''

Crea un informe con la carga de un camión

a partir de archivos camion y precio.

El formato predeterminado de la salida es .txt

Alternativas: .csv o .html

'''

# Lee archivos de datos

camion = leer\_camion(archivo\_camion)

precios = leer\_precios(archivo\_precios)

# Crea la data del informe

data\_informe = hacer\_informe(camion, precios)

# Imprime el informe

formateador = formato\_tabla.crear\_formateador(fmt)

imprimir\_informe(data\_informe, formateador)

Acordate de testear todas las ramas posibles del código para asegurarte de que está funcionando. Llamalo y pedile crear salidas en todos los formatos (podés ver el HTML con tu browser).

**Ejercicio 9.8: Volvamos a armar todo**

Modificá tu programa informe.py de modo que la función informe\_camion() acepte un parámetro opcional que especifique el formato de salida deseado. Por ejemplo:

>>> informe.informe\_camion('../Data/camion.csv', '../Data/precios.csv', fmt = 'txt')

Nombre Cajones Precio Cambio

---------- ---------- ---------- ----------

Lima 100 32.20 8.02

Naranja 50 91.10 15.18

Caqui 150 103.44 2.02

Mandarina 200 51.23 29.66

Durazno 95 40.37 33.11

Mandarina 50 65.10 15.79

Naranja 100 70.44 35.84

>>>

Modificá el programa principal y usá sys.argv() para poder definir un formato particular directamente desde la línea de comandos. En el siguiente ejemplo se ve un caso de uso. Idealmente, ese parámetro debería ser opcional y, si no se lo pasás, debería andar como antes.

bash $ python3 informe.py ../Data/camion.csv ../Data/precios.csv csv

Nombre,Cajones,Precio,Cambio

Lima,100,32.20,8.02

Naranja,50,91.10,15.18

Caqui,150,103.44,2.02

Mandarina,200,51.23,29.66

Durazno,95,40.37,33.11

Mandarina,50,65.10,15.79

Naranja,100,70.44,35.84

Esta versión de informe.py preparala para entregarla.

**Discusión**

El caso que vimos es un ejemplo de uno de los usos más comunes de herencia en programación orientada a objetos: escribir programas extensibles. Un sistema puede definir una interfaz mediante una superclase base, y pedirte que escribas tus propias implementaciones derivadas de esa clase. Si escribís los métodos específicos para tu caso particular podés adaptar la función de un sistema general para resolver tu problema.

Otro concepto, un poco más interesante, es el de crear tus propias abstracciones. En los ejercicios de esta parte definimos *nuestra propia clase* para crear variaciones en el formato de un informe. Tal vez estés pensando "¡Debería usar una biblioteca para crear formatos ya escrita por otre!". Bueno, no. Está bueno que puedas *tanto* crear tu propia clase *como* usar una biblioteca ya escrita. El hecho de usar tu propia clase te da flexibilidad.

Siempre que tu programa adhiera a la interfaz de objetos definida por tu clase, podés cambiar la implementación interna en los objetos que escribas para que funcionen del modo que elijas. Podés escribir todo el código vos o usar bibliotecas ya escritas, no importa. Cuando encuentres algo mejor, cambiás tu implementación para que llame al nuevo código. Si la interfaz que hiciste está bien escrita, no vas a necesitar modificar el programa que usa las diferentes implementaciones. Simplemente funcionan si cumplen los contratos de la interfaz. Es algo muy útil y es uno de los motivos por los que usar herencia puede resolverte los problemas de extensibilidad y diversidad a futuro.

Dicho esto, es cierto que diseñar un programa en el paradigma orientado a objetos puede resultar algo muy difícil. Si vas a encarar proyectos grandes con esta herramienta, consultá libros sobre patrones de diseño en POO. De todos modos, haber entendido lo que acabamos de hacer te permite llegar bastante lejos.

**9.4 Métodos especiales**

El tema de esta sección lo tratamos junto con la introducción a clases, en el video de la [Sección 9.2](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/02_Clases.md#92-clases).

Podemos modificar muchos comportamientos de Python definiendo lo que se conoce como "métodos especiales". Acá vamos a ver cómo usar estos métodos y a discutir brevemente otras herramientas relacionadas.

**Introducción**

Una clase puede tener definidos métodos especiales. Estos métodos tienen un significado particular para el intérprete de Python. Sus nombres empiezan y terminan en \_\_ (doble guión bajo). Por ejemplo \_\_init\_\_.

class Lote(object):

def \_\_init\_\_(self):

...

def \_\_repr\_\_(self):

...

Hay decenas de métodos especiales pero sólo vamos a tratar algunos ejemplos específicos acá.

**Métodos especiales para convertir a strings**

Los objetos tienen dos representaciones de tipo cadena.

>>> from datetime import date

>>> d = date(2020, 12, 21)

>>> print(d)

2020-12-21

>>> d

datetime.date(2020, 12, 21)

>>>

La función str() se usa para crear una representación agradable de ver:

>>> str(d)

'2020-12-21'

>>>

Pero para crear una representación más informativa para programadores, se usa la función repr().

>>> repr(d)

'datetime.date(2020, 12, 21)'

>>>

Las funciones str() y repr() llaman a métodos especiales de la clase para generar la cadena de caracteres que se va a mostrar.

class Date(object):

def \_\_init\_\_(self, year, month, day):

self.year = year

self.month = month

self.day = day

# Con `str()`

def \_\_str\_\_(self):

return f'{self.year}-{self.month}-{self.day}'

# Con `repr()`

def \_\_repr\_\_(self):

return f'Date({self.year},{self.month},{self.day})'

*Nota: Hay una convención para \_\_repr\_\_() que indica que debe devolver un string que, cuando sea pasado a eval() vuelva a crear el objeto subayacente. Analizá el ejemplo de datetime.date(2020, 12, 21). Si no es posible crear un string que haga eso, la convención es generar una representación que sea fácil de interpretar para una persona.*

class Punto():

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return f'({self.x}, {self.y})'

# Used with `repr()`

def \_\_repr\_\_(self):

return f'Punto({self.x}, {self.y})'

**Métodos matemáticos especiales**

Las operaciones matemáticas sobre los objetos involucran llamados a los siguientes métodos.

a + b a.\_\_add\_\_(b)

a - b a.\_\_sub\_\_(b)

a \* b a.\_\_mul\_\_(b)

a / b a.\_\_truediv\_\_(b)

a // b a.\_\_floordiv\_\_(b)

a % b a.\_\_mod\_\_(b)

a << b a.\_\_lshift\_\_(b)

a >> b a.\_\_rshift\_\_(b)

a & b a.\_\_and\_\_(b)

a | b a.\_\_or\_\_(b)

a ^ b a.\_\_xor\_\_(b)

a \*\* b a.\_\_pow\_\_(b)

-a a.\_\_neg\_\_()

~a a.\_\_invert\_\_()

abs(a) a.\_\_abs\_\_()

Así, al definir un método \_\_add\_\_(b) en la clase Punto, por ejemplo, nos permitirá sumar dos instancias de esta clase usando el operador +.

class Punto():

...

...

def \_\_add\_\_(self, b):

return Punto(self.x + b.x, self.y + b.y)

Como en el siguiente ejemplo:

>>> a = Punto(1,2)

>>> b = Punto(3,4)

>>> repr(a + b)

'Punto(4, 6)'

**Métodos especiales para acceder a elementos**

Los siguientes métodos se usan para implementar contenedores:

len(x) x.\_\_len\_\_()

x[a] x.\_\_getitem\_\_(a)

x[a] = v x.\_\_setitem\_\_(a,v)

del x[a] x.\_\_delitem\_\_(a)

Los podés implementar en tus clases.

class Secuencia:

def \_\_len\_\_(self):

...

def \_\_getitem\_\_(self,a):

...

def \_\_setitem\_\_(self,a,v):

...

def \_\_delitem\_\_(self,a):

...

**Invocar métodos**

El proceso de invocar un método puede dividirse en dos partes:

1. Búsqueda: Se usa el operator .
2. Llamado: Se usan ()

>>> m = Lote('Pera', 100, 490.10)

>>> c = m.costo # Búsqueda

>>> c

<bound method Lote.costo of <Lote object at 0x590d0>>

>>> c() # Llamado

49010.0

>>>

*Nota*: la respuesta al pedido de representación de c es algo así como <Método Lote.costo asociado al <objeto Lote en 0x590d0>>\*

**Métodos ligados**

Un método que aún no ha sido llamado por el operador de llamado a funciones () se conoce como *método ligado* y opera dentro de la instancia en la que fue originado.

>>> m = Lote('Pera', 100, 490.10)

>>> m

<Lote object at 0x590d0>

>>> c = m.costo

>>> c

<bound method Lote.costo of <Lote object at 0x590d0>>

>>> c()

49010.0

>>>

Estos métodos ligados pueden ser el origen de errores por desprolijidad, que no son nada obvios. Por ejemplo:

>>> m = Lote('Pera', 100, 490.10)

>>> print('Costo : %0.2f' % m.costo)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: float argument required

>>>

O una fuente de comportamiento extraño que es difícil de debuggear.

f = open(filename, 'w')

...

f.close # EPA! No hicimos nada. `f` sigue abierto.

En ambos casos, el error está causado por omitir los paréntesis en el (intento de) llamado a la función. En estos casos debería haber sido: m.costo() o f.close(). Sin los paréntesis, no estamos llamando a la función sino refiriéndonos al método.

**Acceso a atributos**

Existe una forma alternativa de acceder, manipular, y administrar los atributos de un objeto.

getattr(obj, 'name') # Equivale a obj.name

setattr(obj, 'name', value) # Equivale a obj.name = value

delattr(obj, 'name') # Equivale a del obj.name

hasattr(obj, 'name') # Mira si la propiedad existe

Ejemplo:

if hasattr(obj, 'x'):

x = getattr(obj, 'x'):

else:

x = None

*Nota*: si getattr() no encuentra el atributo buscado (x en este ejemplo), devuelve el argumento opcional *arg* (None en este caso)

x = getattr(obj, 'x', None)

**Ejercicios**

**Ejercicio 9.9: Mejor salida para objetos**

Modificá el objeto Lote que definiste en lote.py (del [Ejercicio 9.1](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/02_Clases.md#ejercicio-91-objetos-como-estructura-de-datos)) de modo que el método \_\_repr\_\_() genere una salida más agradable. Por ejemplo queremos un comportamiento como éste:

>>> peras = Lote('Pera', 100, 490.1)

>>> peras

Lote('Pera', 100, 490.1)

>>>

Fijate lo que ocurre cuando leés un camión de frutas y mirás la salida resultante después de hacer estos cambios. Un ejemplo:

>>> import informe

>>> camion = informe.leer\_camion('../Data/camion.csv')

>>> camion

... fijate cuál es la salida ...

>>>

Guardá el archivo lote.py para entregar.

**Ejercicio 9.10: Ejemplo de getattr()**

getattr() es un mecanismo alternativo de leer atributos. Puede usarse para escribir código sumamente versátil. Probá este ejemplo, para empezar:

>>> import lote

>>> c = lote.Lote('Peras', 100, 490.1)

>>> columnas = ['nombre', 'cajones']

>>> for colname in columnas:

print(colname, '=', getattr(c, colname))

nombre = Peras

cajones = 100

>>>

Queremos que observes algo interesante: los datos de salida están completamente especificados por los nombres de los atributos listados en la variable columnas. Estamos usando el contenido de una variable ('nombre' y 'cajones') como nombres de otras variables, o de atributos de un objeto. No es usual.

Si te dan ganas, en el archivo formato\_tabla.py usá esta idea pero extendela, y creá una función imprimir\_tabla() que imprima una tabla mostrando, de una lista de objetos de tipo arbitrario, una lista de atributos especificados por le usuarie.

Tal como antes hicimos con la función imprimir\_informe() del [Ejercicio 6.4](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/06_Organizaci%C3%B3n_y_Complejidad/02_Scripts.md#ejercicio-64-estructurar-un-programa-como-una-colecci%C3%B3n-de-funciones) imprimir\_tabla() también debería aceptar cualquier instancia de la clase FormatoTabla para definir el formato de la salida. La idea es que funcione más o menos así:

>>> import informe

>>> camion = informe.leer\_camion('../Data/camion.csv')

>>> from formato\_tabla import crear\_formateador, imprimir\_tabla

>>> formateador = crear\_formateador('txt')

>>> imprimir\_tabla(camion, ['nombre','cajones'], formateador)

nombre cajones

---------- ----------

Lima 100

Naranja 50

Caqui 150

Mandarina 200

Durazno 95

Mandarina 50

Naranja 100

>>> imprimir\_tabla(camion, ['nombre','cajones','precio'], formateador)

nombre cajones precio

---------- ---------- ----------

Lima 100 32.2

Naranja 50 91.1

Caqui 150 103.44

Mandarina 200 51.23

Durazno 95 40.37

Mandarina 50 65.1

Naranja 100 70.44

**9.5 Objetos, pilas y colas**

Esta sección tiene un [video](https://youtu.be/-SYcwU1XOn4) donde trabajamos con colas. En este video trabajamos con los archivos que figuran en [esta carpeta comprimida](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/clase9_colas.zip).

En esta sección tendrás que resolver algunos ejercicios definiendo clases y objetos.

**Un ejercicio geométrico**

Creá una clase llamada Rectangulo que va a estar definido por dos puntos. Para esos dos puntos, usá la clase Punto de la Sección anterior. El rectángulo es paralelo a los ejes, los puntos representan dos esquinas opuestas cualesquiera. La clase debe tener un método constructor para crear el rectángulo a partir de dos puntos y los siguientes métodos:

* base() que dé la medida de la base del rectángulo.
* altura() que dé la medida de la altura del rectángulo.
* area() que dé la medida del área del rectángulo.
* Creá métodos especiales \_\_str\_\_ y \_\_repr\_\_.
* desplazar(desplazamiento) que dado un desplazamiento (de tipo Punto) desplace el rectángulo en ambas coordenadas usando el método add de la clase Punto.
* rotar() que rote el rectángulo sobre su esquina inferior derecha 90 grados a la derecha.

Probá tu código:

>>> ul = Punto(0,2)

>>> lr = Punto(1,0)

>>> ll = Punto(0,0)

>>> ur = Punto(1,2)

>>> rect1 = Rectangulo(ul,lr)

>>> rect2 = Rectangulo(ll,ur)

>>> rect1.base()

1

>>> rect1.base()

1

>>> rect2.altura()

2

>>> rect2.altura()

2

>>> rect1.rotar()

>>> rect2.rotar()

>>> rect1.base()

2

>>> rect2.base()

2

>>> rect1.altura()

1

>>> rect2.altura()

1

**Ejercicio 9.11: Canguros buenos y canguros malos**

Este ejercicio está relacionado con un error muy común en Python. Escribí una definición de una clase Canguro que tenga:

* Un método \_\_init\_\_ que recibe un nombre para el canguro y una lista (parámetro opcional) e inicializa un atributo llamado contenido\_marsupio con la lista que le pases como parámetro o como lista vacía si no le pasás nada.
* Un método llamado meter\_en\_marsupio que, dado un objeto cualquiera, lo agregue a la lista contenido\_marsupio.
* Un método \_\_str\_\_ que devuelve una representación como cadena del objeto Canguro indicando su nombre y los contenidos de su marsupio.

Probá tu código creando dos objetos, madre\_canguro y cangurito y guardá en el marsupio de la madre algunos objetos y al propio cangurito.

Luego, mirá el ejemplo canguro\_malo.py copiado a continuación. Este ejemplo intenta resolver el problema anterior, pero tiene un bug. Analizalo y corregilo. Entregá como respuesta un archivo canguros\_buenos.py conteniendo, perimero la clase definida por vos y luego una corrección de la clase definida en el ejemplo, junto con un comentario indicando dónde estaba el error y en qué constía.

# canguro\_malo.py

"""Este código continene un

bug importante y dificil de ver

"""

class Canguro:

"""Un Canguro es un marsupial."""

def \_\_init\_\_(self, nombre, contenido=[]):

"""Inicializar los contenidos del marsupio.

nombre: string

contenido: contenido inicial del marsupio, lista.

"""

self.nombre = nombre

self.contenido\_marsupio = contenido

def \_\_str\_\_(self):

"""devuelve una representación como cadena de este Canguro.

"""

t = [ self.nombre + ' tiene en su marsupio:' ]

for obj in self.contenido\_marsupio:

s = ' ' + object.\_\_str\_\_(obj)

t.append(s)

return '\n'.join(t)

def meter\_en\_marsupio(self, item):

"""Agrega un nuevo item al marsupio.

item: objecto a ser agregado

"""

self.contenido\_marsupio.append(item)

#%%

madre\_canguro = Canguro('Madre')

cangurito = Canguro('gurito')

madre\_canguro.meter\_en\_marsupio('billetera')

madre\_canguro.meter\_en\_marsupio('llaves del auto')

madre\_canguro.meter\_en\_marsupio(cangurito)

print(madre\_canguro)

# Al ejecutar este código todo parece funcionar correctamente.

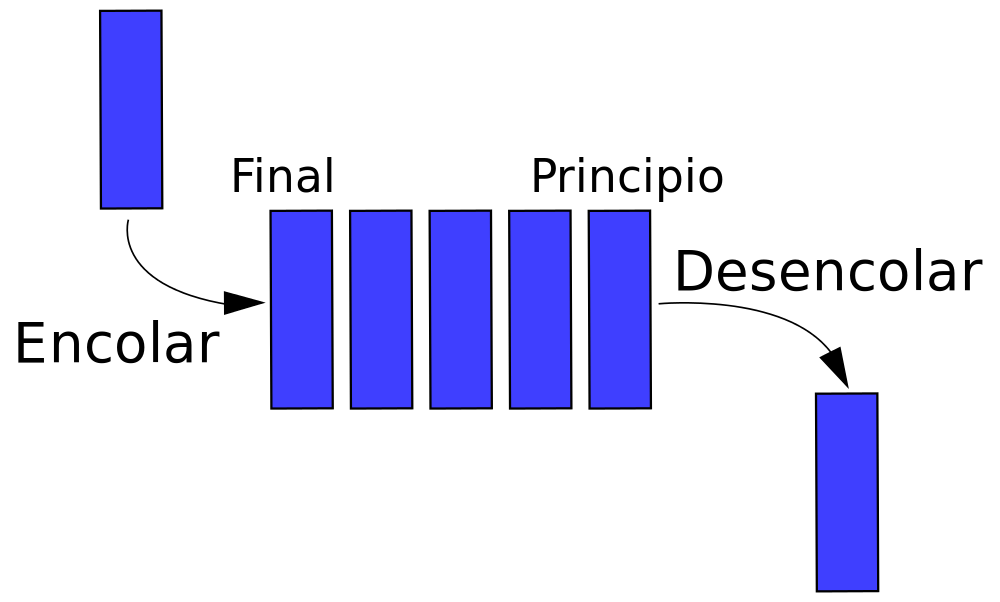
# Para ver el problema, imprimí el contenido de cangurito.

Ojo: al corregir el bug no modifiques la firma del constructor \_\_init\_\_ del objeto: asegurate que siga aceptando los mismos parámetros (obligatorios y opcionales) que antes.

**Colas**

Una **cola** es una estructura de datos. Se caracteriza por contener una secuencia de elementos y dos operaciones: encolar y desencolar. La primera, encolar, agrega un elemento al final de la secuencia que contiene la cola. Desencolar, por su parte, devuelve el primer elemento de la secuencia y lo elimina de la misma.

Las colas también se llaman estructuras FIFO (del inglés First In First Out), debido a que el primer elemento en entrar a la cola será también el primero en salir. El nombre cola se le da por su analogía con las colas que hacemos (o hacíamos cuando podíamos salir de casa) para entrar al cine, por ejemplo.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/colas.png)

Esta es una posible implementación de la clase Cola:

class Cola:

'''Representa a una cola, con operaciones de encolar y desencolar.

El primero en ser encolado es tambien el primero en ser desencolado.

'''

def \_\_init\_\_(self):

'''Crea una cola vacia.'''

self.items = []

def encolar(self, x):

'''Encola el elemento x.'''

self.items.append(x)

def desencolar(self):

'''Elimina el primer elemento de la cola

y devuelve su valor.

Si la cola esta vacia, levanta ValueError.'''

if self.esta\_vacia():

raise ValueError('La cola esta vacia')

return self.items.pop(0)

def esta\_vacia(self):

'''Devuelve

True si la cola esta vacia,

False si no.'''

return len(self.items) == 0

**Ejercicio 9.12: Torre de Control**

Usando un par de objetos de la clase Cola, escribí una nueva clase llamada TorreDeControl que modele el trabajo de una torre de control de un aeropuerto con una pista de aterrizaje. Los aviones que están esperando para aterrizar tienen prioridad sobre los que están esperando para despegar. La clase debe funcionar conforme al siguiente ejemplo:

>>> torre = TorreDeControl()

>>> torre.nuevo\_arribo('AR156')

>>> torre.nueva\_partida('KLM1267')

>>> torre.nuevo\_arribo('AR32')

>>> torre.ver\_estado()

Vuelos esperando para aterrizar: AR156, AR32

Vuelos esperando para despegar: KLM1267

>>> torre.asignar\_pista()

El vuelo AR156 aterrizó con éxito.

>>> torre.asignar\_pista()

El vuelo AR32 aterrizó con éxito.

>>> torre.asignar\_pista()

El vuelo KLM1267 despegó con éxito.

>>> torre.asignar\_pista()

No hay vuelos en espera.

Guardá tu solución (conteniendo también la definición de la clase Cola) en torre\_control.py para entregar al final de la clase.

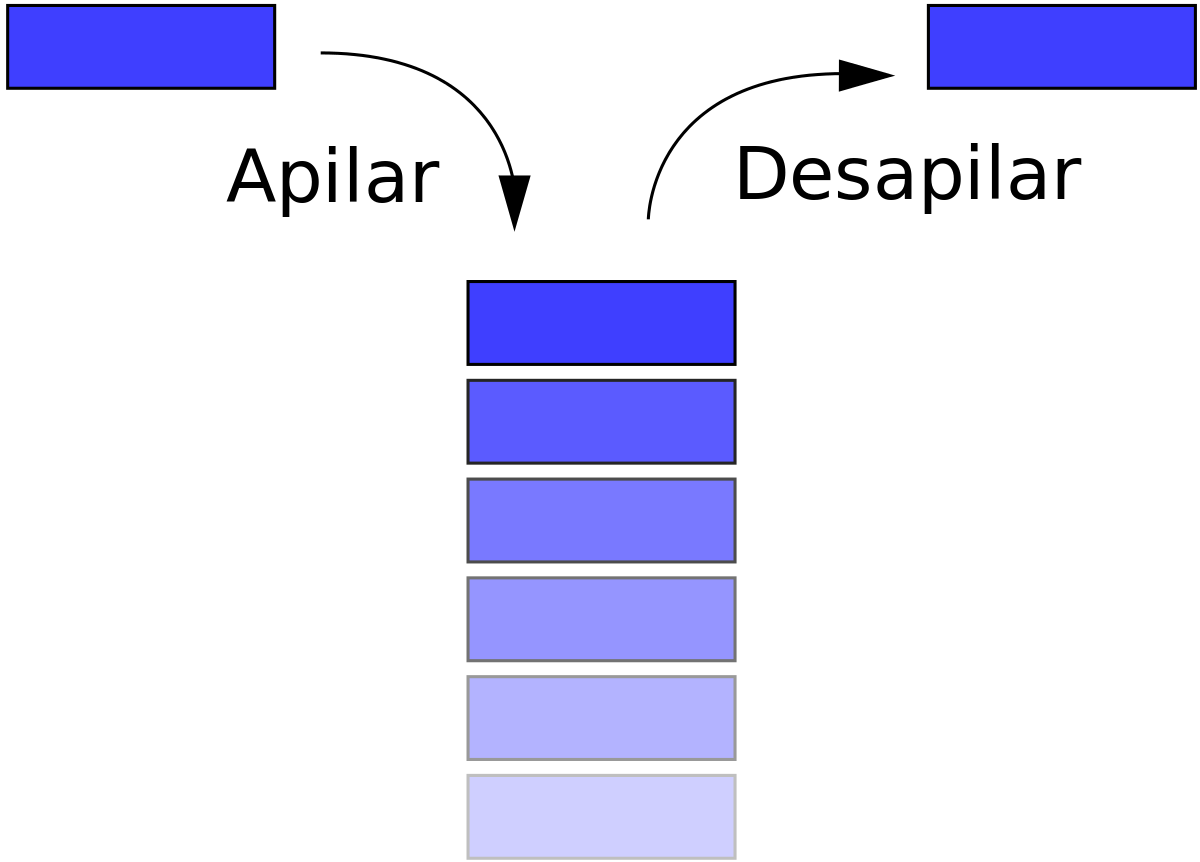
**Pilas**

Una **pila** (*stack* en inglés) es una estructura de datos. Se trata de una lista ordenada que permite almacenar y recuperar datos, con un modo de acceso de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Funcionan de manera opuesta que las colas que mencionamos antes.

Las pilas y colas son estructuras de datos que se aplican en multitud de contextos debido a su simplicidad y capacidad de modelar diferentes procesos.

La operaciones (métodos) elementales de las pilas son *apilar* (coloca un objeto en la pila) y *desapilar* (retira el último elemento apilado). En inglés se llaman *push* y *pop* y son análogos al *encolar* y el *desencolar* de la colas.

En cada momento solamente se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado. La operación *desapilar* justamente permite la obtención de este elemento, que es retirado de la pila.

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/pilas.png)

La **pila de llamadas** (en inglés *call stack*) de un lenguaje (por ejemplo Python), es una pila manejada por el intérprete que almacena la información sobre las subrutinas activas en cada instante. También se la conoce como pila de ejecución o pila de control y se usa para llevar registro de las funciones que se fueron llamando y el de las variables definidas en cada contexto.

Por ejemplo, si definimos las siguientes funciones:

def f():

x = 50

a = 20

print("En f, x vale", x)

def g():

x = 10

b = 45

print("En g, antes de llamar a f, x vale", x)

f()

print("En g, después de llamar a f, x vale", x)

la ejecución de g() resulta en:

>>> g()

En g, antes de llamar a f, x vale 10

En f, x vale 50

En g, después de llamar a f, x vale 10.

Para poder volver a recuperar el valor 10 para x en g() luego de llamar a f() se manejó adecuadamente la pila de llamadas. Podemos pensar que en la ejecución de g(), justo antes de llamar a f() había un *estado* que podría ser resumido en estado = {función: 'g', próxima\_línea\_a\_ejecutar: 4, variables: {x: 10, b: 45}}. Luego se ejecuta la cuarta línea de código. El intérprete incrementa próxima\_línea\_a\_ejecutar y, antes de llamar a f(), **apila** el estado en la pila de llamadas. Al llamar a f(), el nuevo estado pasa a ser estado = {función: 'f', próxima\_línea\_a\_ejecutar: 1, variables = {}}. El intéreprete ejecuta las tres líneas de código de f, incrementando la variable próxima\_línea\_a\_ejecutar en cada paso, y agregando x:50 y luego a:20 el estado de las variables. Por lo tanto, termina la ejecución de f en el estado = {función: 'f', próxima\_línea\_a\_ejecutar: 4, variables = {x: 50, a: 20}}. Como ya no hay más código que ejecutar de f() el intérprete **desapila** un estado y continúa con la ejecución usando estado = {función: 'g', próxima\_línea\_a\_ejecutar: 5, variables: {x: 10, b: 45}}, y por lo tanto imprime:

En g, después de llamar a f, x vale 10.

Estos conceptos son importantes para la clase próxima donde estudiaremos funciones que se llaman a sí mismas *recursivamente*. Si no fuera por la pila de llamadas, los valores de las variables de las diferentes instancias de una función recursiva correrían el riesgo de mezclarse y confundirse.

**Ejercicio 9.13: implementar el TAD pila**

Implementá en una clase Pila el TAD descripto anteriormente con los métodos apilar(), desapilar() y esta\_vacia().

Usala para reproducir el siguiente código:

def mostrar\_x\_del\_estado(estado):

print(f"Ejecutando {estado['función']}(), x vale {estado['variables']['x']}")

pila\_de\_llamadas = Pila()

#la ejecución está en la línea 3 de g(). El estado tiene x=10.

estado = {'función': 'g', 'próxima\_línea\_a\_ejecutar': 3, 'variables': {'x': 10, 'b': 45}}

mostrar\_x\_del\_estado(estado)

#sigo ejecutando, toca llamar a f(): incremento y apilo el estado.

estado['próxima\_línea\_a\_ejecutar'] = 5

pila\_de\_llamadas.apilar(estado)

#llamo a f y ejecuto primeras líneas

estado = {'función': 'f', 'próxima\_línea\_a\_ejecutar': 3, 'variables': {'x': 50, 'a': 20}}

mostrar\_x\_del\_estado(estado)

#termina ejecución de f: se desapila el estado:

estado = pila\_de\_llamadas.desapilar()

mostrar\_x\_del\_estado(estado)

Su ejecución debería dar:

Ejecutando g(), x vale 10

Ejecutando f(), x vale 50

Ejecutando g(), x vale 10

**9.6 Teledetección**

En este [video](https://youtu.be/oQ0ce5qhJp0) introducimos el tema de esta sección.

En este ejercicio vamos a trabajar con una imagen satelital obtenida por sensores a bordo del satélite Landsat8. Es un ejercicio optativo para entregar. Si querés, hacelo y guardalo en el archivo NDVI.py.

**Ejercicio 9.14: Optativo de teledetección**

**Autora:**[**Mariela Rajngewerc**](https://github.com/marielaraj/)

La imagen original fue bajada de la página del [earthexplorer](https://earthexplorer.usgs.gov/). En esa página se pueden bajar imágenes con distinto nivel de pre-procesamiento. Para este ejercicio bajamos una imagen de nivel de procesamiento 2, esto quiere decir que los valores de los pixeles representan la reflectancia en superficie en distintas longitudes de onda. [Acá](https://www.usgs.gov/media/files/landsat-8-collection-1-land-surface-reflectance-code-product-guide) pueden encontrar el manual de estas imagenes donde les detallan la descripción tanto de los nombres de lor archivos como de los preprocesamiento que tienen realizados.

Para este ejercicio hemos realizado un clip de cada una de las bandas originales de la imagen y ya multiplicamos a cada una de las bandas por el factor de escala indicado en el manual (0,0001).

Las longitudes de onda y la resoluciones de cada banda de la imágen se describen a continuación:

| **Banda** | **Longitud de onda (nanómetros)** | **Resolución espacial (metros)** |
| --- | --- | --- |
| Banda 1 - Aerosoles | 430 - 450 | 30 |
| Banda 2 - Azul | 450 - 510 | 30 |
| Banda 3 - Verde | 530 - 590 | 30 |
| Banda 4 - Rojo | 640 - 670 | 30 |
| Banda 5 - Infrarrojo cercano | 850 - 880 | 30 |
| Banda 6 - Infrarrojo medio 1 | 1570 - 1650 | 30 |
| Banda 7 - Infrarrojo medio 2 | 2110 - 2290 | 30 |

Si desean abrir los datos de la imagen original en Python deberán bajar algunas librerías específicas para la manipulación de datos satelitales, por ejemplo: **gdal**. [Acá](https://www.github.com/marielaraj/pycon_tallerimgssat) hay un tutorial de los primeros pasos.

En la carpeta [clip](https://drive.google.com/file/d/1uoigo5s2xgWfbBQdUcJfOdhfYMNjZ8Ku/view?usp=sharing) encontrarán los datos que vamos a usar en los ejercicios. Cada banda del clip se encuentra en formato .npy

**Ejercicios:**

**Ejercicio 9.15: Ver una banda**

a) Usá [numpy](https://numpy.org/doc/stable/reference/generated/numpy.load.html) para levantar cada una de las bandas y plt.imshow(banda) para verla. ¿Se ve correctamente? Podés ajustar el rango de visualización de colores usando los parámetros vmin y vmax.

*Sugerencia*: Con plt.hist(banda.flatten(),bins = 100) vas a ver un histograma de los valores en la matriz banda. Podés usarlo para guiarte en la búsqueda del rango que tiene sentido usar como vmin y vmax.

b) Probá usando percentiles para fijar el rango. Algo como

vmin = np.percentile(data.flatten(), q)

vmax = np.percentile(data.flatten(), 100-q)

c) Escribí una función crear\_img\_png(carpeta, banda) que, dada una carpeta y un número de banda, muestre la imagen de dicha banda y la guarde en formato .png. Asegurate de incorporar un colorbar al lado de la imágen.

Tené en cuenta lo que hiciste en los puntos anteriores para que se vea adecuadamente.

**Ejercicio 9.16: Histogramas**

Escribí ahora otra función, llamada crear\_hist\_png(carpeta, banda, bins) que, dada una carpeta, un número de banda y una cantidad de bins, muestre el histograma (con la cantidad de bins seleccionados) de los valores de dicha banda y la guarde en formato .png.

**Ejercicio 9.17: Máscaras binarias**

a) Usá las funciones crear\_img\_png y crear\_hist\_png que hiciste en los puntos anteriores para generar las imágenes e histogramas de cada banda.

b) ¿Qué banda o bandas parecieran tener histogramas bimodales, mostrando diferentes tipos de pixels? Elegí una de esas bandas y, observando el histograma, seleccioná un umbral que te permita distinguir los dos tipos de píxels. Por ejemplo, podés crear una matriz del mismo tamaño de la banda donde a cada píxel le corresponda un 1 o un 0, 1 si está por arriba del umbral y 0 si no.

Graficá la imágen binaria así obtenida. ¿A qué corresponden los dos tipos de píxeles que pudiste distinguir tan fácilmente?

**Ejercicio 9.18: Clasificación manual**

En este ejercicio vamos a trabajar con un índice: el Índice de Vegetación de Diferencia Normalizada, también conocido como [NDVI](https://es.wikipedia.org/wiki/%C3%8Dndice_de_vegetaci%C3%B3n_de_diferencia_normalizada) por sus siglas en inglés. Este índice, basado en la intensidad de la radiación de dos bandas del espectro electromagnético que interactúan particularmente con la vegetación, aporta información sobre la cantidad, estado y desarrollo de la misma.

Para calcular el NDVI se utilizan las bandas espectrales Roja e Infrarroja y el cálculo se hace mediante la siguiente fórmula:

(INFRARROJO\_CERCANO - ROJO) / (INFRARROJO\_CERCANO + ROJO)

a) Calcular el NDVI en una nueva matriz.

b) Categorizá los valores obtenidos en cada píxel de acuerdo a clases que nos sean más útiles y fáciles de interpretar. La tabla a continuación muestra una propuesta de categorías que podés considerar:

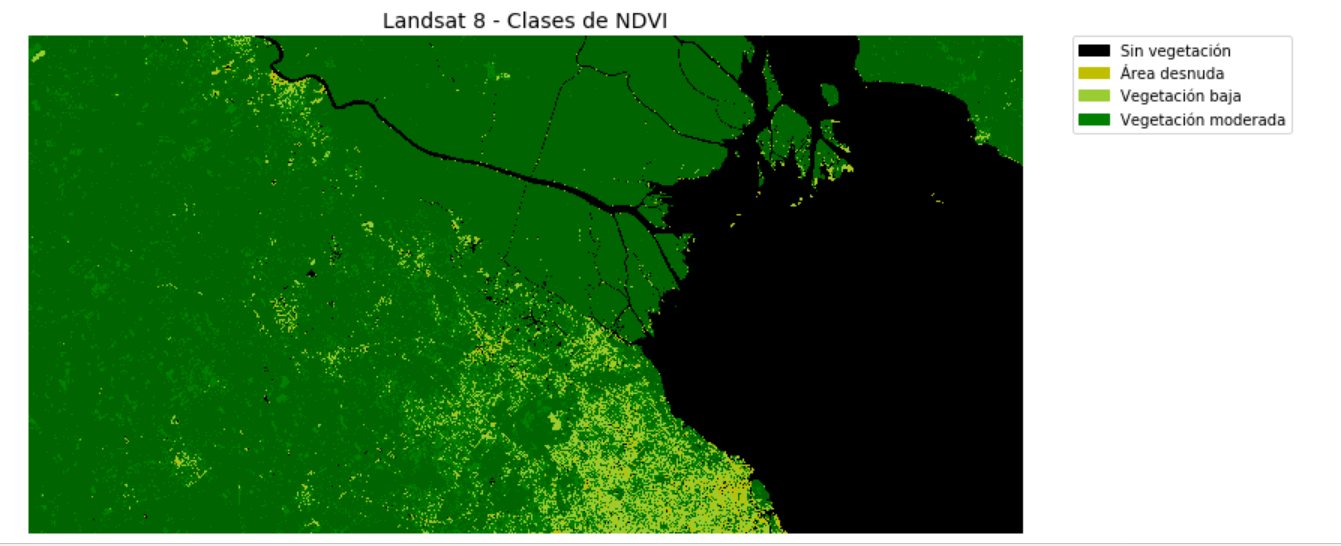
| **Valor de NDVI** | **Nombre de la clase** | **Identificador de Clase** | **color** |
| --- | --- | --- | --- |
| < 0 | No vegetada | 0 | black |
| entre 0 y 0.1 | Área desnuda | 1 | y |
| entre 0.1 y 0.25 | Vegetación baja | 2 | yellowgreen |
| entre 0.25 y 0.4 | Vegetación moderada | 3 | g |
| >0.4 | Vegetación densa | 4 | darkgreen |

Creá un np.array que le asigne a cada píxel el número dado por el *identificador de categoría* correspondiente según la tabla. Llamá *clases\_ndvi* a la matriz así obtenida.

c) Generá un gráfico con matplotlib mostrando las clases obtenidas.

d) Crear un colorMap para lograr asignarle a cada clase el color sugerido en la tabla. Para esto podés usar la función ListedColormap incluída en matplotlib.colors y crear un colorMap (cmap).

e) Ponele una leyenda que indique el nombre de cada clase con el color asignado, para eso te sugerimos usar la función Patch que se encuentra en matplotlib.patches. Para que puedas orientarte, te mostramos a continuación un ejemplo de resultado esperado:

[](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/img.png)

Si llegaste hasta acá, no te olvides de guardar tu trabajo en el archivo NDVI.py y entregarlo. A continuación, un ejercicio que usa herramientas un poco más avanzadas de aprendizaje automático.

**Ejercicio 9.19: Clasificación automática**

En el ejercicio anterior definimos a mano los umbrales que distinguen las clases. Es posible hacer esto de forma automática. Para eso se usan técnicas de clustering. El siguiente código muestra un ejemplo con un clasificador muy sencillo: kmeans. Este clasificador está ya implementado en la biblioteca [sklearn](https://scikit-learn.org/stable/) que es una biblioteca dedicada al aprendizaje automático en python (probablemente la más usada para esto).

# filtro datos ruidosos o que puedan traer problemas.

# el NDVI debe estar entre -1 y 1.

ndvi[ndvi > 1] = 1

ndvi[ndvi < -1] = -1

#importo el clasificador y defino una instancia para clasificar con dos etiquetas

from sklearn.cluster import KMeans

kmeans = KMeans(n\_clusters = 2)

#le saco la estructura bidimensional a la matriz NDVI y la llamo datos

datos = ndvi.reshape(-1,1) #datos es un vector con un dato de NDVI por pixel.

#entreno o ajusto el el clasificador con los datos (demora!)

kmeans.fit(datos) #ajusta el modelo

#usa el modelo ajustado para poner etiquetas

etiquetas = kmeans.predict(ndvi.reshape(-1,1))

#visualizo los resultados recuperando la estructura bidimensional de la matriz

plt.imshow(etiquetas.reshape(ndvi.shape))

Probá ajustando el número de clusters (n\_clusters = 5, por ejemplo) y corriendo nuevamente el modelo. Ponele colores diferentes a las diferentes clases obtenidas.

Si tarda mucho podés trabajar con un pedazo de la imágen. Por ejemplo si hacés ndvi\_clip = ndvi[1000:2000,2000:3000] te quedás con un cuadradito que es un octavo de la imagen original y podés usarlo para probar cosas rápido. Si te convencen los resultados podés correr tu algoritmo sobre la imágen completa.

**Sugerencia: Código para colorbars**

Revisar este código para el cmap:

from matplotlib import colors

# Creo colores

cmap = colors.ListedColormap(['black', 'y',

'yellowgreen', 'green', 'darkgreen'])

# Defino los limites de cada color

limites = [0, 1, 2, 3, 4]

norm = colors.BoundaryNorm(limites, cmap.N)

# Genero el grafico con colores

plt.imshow(clases\_ndvi, cmap=cmap, norm=norm)

Y éste para las leyendas:

import matplotlib.patches as mpatches

# Genero leyenda y grafico con leyenda

texts = ['Sin vegetacion', 'Area desnuda', 'Vegetacion baja',

'Vegetacion moderada', 'Vegetacion densa']

patches = [mpatches.Patch(color=cmap(i), label="{:s}".format(texts[i]) ) for i in range(len(texts))]

plt.legend(handles=patches, bbox\_to\_anchor=(0.2,1.3), loc='center', ncol=1 )

plt.imshow(clases\_ndvi, cmap=cmap, norm=norm)

plt.show()

**9.7 Cierre de la clase**

En esta clase vimos las ventajas de estructurar un programa con las ideas Programación Orientada a Objetos. Vimos cómo podés definir tus propias clases y te mostramos cómo este paradigma puede aportar tanto a la organización de un programa "estático" como también para preparar programas para que sean fácilmente extensibles.

Una de las formas de extender el comportamiento de un programa es definir una interfase de interacción entre objetos de modo que un comportamiento nuevo pueda programarse sin tocar (casi) el código preexistente.

Otra forma muy interesante es definir clases base, abstractas, que van a ser implementadas luego, por herencia, en sus versiones definitivas.

La experiencia te va a permitir decidir (quizás tardíamente) cuándo es mas conveniente una arquitectura o la otra.

En fin, para cerrar esta clase, entregá:

* El archivo informe.py del [Ejercicio 9.8](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/03_Herencia.md#ejercicio-98-volvamos-a-armar-todo).
* El archivo lote.py del [Ejercicio 9.9](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/04_M%C3%A9todos_Especiales.md#ejercicio-99-mejor-salida-para-objetos).
* El archivo torre\_control.py del [Ejercicio 9.12](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/05_Pilas_Colas.md#ejercicio-912-torre-de-control).
* El archivo canguros\_buenos.py del [Ejercicio 9.11](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/05_Pilas_Colas.md#ejercicio-911-canguros-buenos-y-canguros-malos).
* El archivo NDVI.py del [Ejercicio 9.14](https://github.com/python-unsam/Programacion_en_Python_UNSAM/blob/master/Notas/09_Clases_y_Objetos/06_Teledeteccion.md#ejercicio-914-optativo-de-teledetecci%C3%B3n) (optativo).

Además te pedimos que completes [este formulario](https://docs.google.com/forms/d/13FKe29wNTl7yfVQk4wW6k90vrrEwDlH79b3WUB95DiU) usando tu dirección de mail como identificación. Al terminar vas a obtener un link para enviarnos tus ejercicios.

¡Nos vemos!