NOTA: Si detectas algún error en este Colab, pon un mensaje en el foro para que lo podamos solucionar o envía un correo.

1 Clases y objetos

Python es un lenguaje que también soporta el paradigma de Programación Orientada a Objetos (POO). Como ya sabemos de otros lenguajes, las clases definen la estructura y comportamiento de objetos. El término *clase* hace referencia a una plantilla que define atributos (variables propias de la clase) y métodos (funciones propias de la clase). Por contra, un *objeto* es la instanciación en tiempo de ejecución de una clase.

Podemos crear una clase en Python utilizando la palabra reservada `class` seguida de un espacio y el nombre de nuestra clase. Por convención, el nombre de la clase lo pondremos en PascalCase (i.e. la primera letra en mayúscula).

Las clases utilizan un método `__init__` (con doble subrayado anterior y posterior) como inicializador. Este método recibe un primer parámetro de entrada llamado `self` y que se usa para referirse al propio objeto (como el `this` de otros lenguajes). A parte de este primer parámetro, podemos añadir más, que serán **atributos** de la clase. En comparación con otros lenguajes como Java o C#, el método `__init__` no es realmente el constructor, sino un iniciador. En Python, este método se utiliza para configurar un objeto que ya existe en el momento en que se llama.

Por convención, los atributos seguirán las indicaciones de las variables. Esto es, en minúsculas con un subrayado (_) para separar las distintas palabras que tenga.

Observa el siguiente ejemplo donde definimos una clase persona con dos parámetros adicionales: nombre (n) y edad (e). Estos parámetros los utilizaremos para asignarlos a dos atributos de la clase, llamados nombre y edad:

```
# Definimos una clase Persona
class Persona:
  """ Creamos una nueva persona con un nombre y una edad """
  def __init__(self, n, e):
    self.nombre = n
    self.edad = e
```

Como has podido comprobar, a diferencia de otros lenguajes, en Python **no es necesario hacer explícitos** los atributos de la clase, sino que son creados en el momento que utilizamos la sintaxis `self.atributo`. Podemos ver la palabra `self` como el `this` que utilizamos en otros lenguajes.

De hecho, si definimos un atributo a nivel de clase (igual que hacemos en otros lenguajes como Java o C#), estaremos definiendo un **atributo de clase**, es decir, un atributo que tiene el mismo valor para todos los objetos que se creen:

```
# Definimos una clase Persona
class Persona:
  """ Atributo de clase (mismo valor para todas las personas) """"
  unidad_masa = "kg"

  """ Atributos de instanacia. Cada persona puede tener un nombre y una edad distintos """
  def __init__(self, n, e):
    self.nombre = n
    self.edad = e
```

Para crear un objeto o instancia de una clase, simplemente llamaremos a una función cuyo nombre es el nombre propio de la clase y pasando por parámetro aquellos que reciba su función `__init__` a excepción del primer parámetro `self`. Una vez creado el objeto, podemos acceder a sus atributos simplemente con un punto:

```
p = Persona("Juan", 25)

print("El nombre de p es ",p.nombre, " y la edad es ",p.edad)
```

De forma similar a los atributos, podemos definir **métodos** al igual que definimos cualquier función. Lo único que tenemos que tener en cuenta es que los métodos de clase tienen que ir **identados** dentro de la clase y que debemos **añadir** `self` como primer parámetro. A continuación se muestra el ejemplo de uso de un método `mostrar_info()`:

```
# Definimos una clase Persona
class Persona:
  """ Creamos una nueva persona con un nombre y una edad """
  def __init__(self, n, e):
    self.nombre = n
    self.edad = e

  def mostrar(self):
    print("El nombre es ",self.nombre, " y la edad es ",self.edad)

p = Persona("Juan", 25)
p.mostrar()
```

En otros lenguajes, se utiliza `Null` para hacer referencia a un puntero que no apunta a nada, denotando un objeto o una variable vacía. Python utiliza `None` para definir objetos y variables nulas. Lo podrás ver normalmente en alguna función que no devuelve nada.

En el siguiente ejemplo puedes ver una función que devuelve la cantidad de claves de un diccionario cuyo valor coincida con el valor pasado por parámetro. Si no existe ninguna, devuelve `None`. Observa también como, por convenio, utilizamos `is` y `is not` para comparar con `None` en vez de `==` y `!=`

```
def matching_list(dict, value):
  matchings = []
  for item in dict:
    if dict[item] == value:
      matchings.append(item)

  if len(matchings) != 0:
    return len(matchings)
  else:
    return None

pairs = {"item1" : "coche", "item2" : "moto", "item3" : "bicicleta"}

res = matching_list(pairs, "moto")
if res is not None:
  print("Se han encontrado "+str(res)+" elementos")
else:
  print("No hay ningún item que coincida")
```

1.1 Visibilidad

Por defecto, todos los atributos y métodos de una clase son públicos. Si queremos hacerlos privados, debemos añadir el **doble subrayado** delante del nombre (__). Observa en el siguiente ejemplo cómo desde fuera de la clase sí que podemos acceder a los atributos y métodos públicos pero no podemos acceder a los privados:

```
# Definimos una clase Persona
class Persona:


  def __init__(self, n, e):
    self.nombre = n #público
    self.__edad = e #privado

  def mostrar(self): #público
    print("El nombre es ",self.nombre, " y la edad es ",self.__edad)

  def __mostrar(self): #privado
    print("El nombre es ",self.nombre, " y la edad es ",self.__edad)

p = Persona("Juan", 25)
print("El nombre es "+p.nombre)
p.mostrar()

#Estos fallan
print("La edad es "+p.__edad)
p.__mostrar()
```

También puedes encontrar atributos o métodos que tienen un subrayado simple **(_)**. Esto, por convención, también se utiliza para hacer referencia a que son privados, aunque a nivel práctico, no lo serían.

En Python también existen métodos especiales de clases. Ya hemos visto `__init__` pero también podemos utilizar otros. Un ejemplo es `__str__` el cual devuelve una cadena con lo que queremos que se muestre por pantalla cuando imprimimos el objeto. Observa el siguiente ejemplo y después comenta la función `__str__` para comprobar la diferencia:

```
# Definimos una clase Persona
class Persona:

  def __init__(self, n, e):
    self.nombre = n
    self.edad = e

  def __str__(self):
    return "El nombre es "+self.nombre+" y la edad es "+str(self.edad)

p = Persona("Juan", 25)
print("El nombre es "+p.nombre)
print(p)
```

1.2 Herencia

La herencia nos permite crear nuevas clases derivadas (subclases) a partir de una clase principal. Para definir una subclase que herede de otra clase, debemos situarnos después de la clase principal y utilizar el nombre de la clase principal en la propia definición de la subclase. En el siguiente ejemplo puedes ver cómo definiríamos la clase `Circulo` como una subclase de `Figura`:

```
class Figura:
  def __init__(self, tipo):
    self.tipo = tipo
    self.area = 0
    self.perimetro = 0

  def __str__(self):
    return "Figura: "+self.tipo+". Area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

class Circulo(Figura): #definimos subclase de Figura
  PI = 3.14

  def __init__(self, radio):
    super().__init__("circulo") #llamamos al inicializador de la clase principal
    self.radio = radio #atributo específico de la subclase
    self.area = self.PI * self.radio**2
    self.perimetro = 2 * self.PI * self.radio

c = Circulo(3)
print(c) #__str__ se hereda de figura
```

Como puedes ver en el ejemplo anterior, en la subclase `Circulo` podemos definir atributos específicos (p.e. `radio`) y también añadir parámetros específicos en el método `__init__` (redefiniendo el método con los parámetros de la clase principal y los nuevos de la subclase). Los parámetros que pertenecen a la clase principal, se envían al `__init__` de la clase principal mediante `super()`. Los específicos se asignan directamente a algún atributo de la subclase.

Como sabes, los métodos se heredan de la clase principal, pero si queremos que alguno tenga una funcionalidad distinta, podemos **sobreescribirlo** en la subclase:

```
class Figura:
  def __init__(self, tipo):
    self.tipo = tipo
    self.area = 0
    self.perimetro = 0

  def __str__(self):
    return "Figura: "+self.tipo+". Area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

class Circulo(Figura):
  PI = 3.14
```

```
  def __init__(self, radio):
    super().__init__("circulo")
    self.radio = radio
    self.area = self.PI * self.radio**2
    self.perimetro = 2 * self.PI * self.radio

  def __str__(self): #Sobreescribimos el método en la subclase
    return "Este círculo de radio "+str(self.radio)+" tiene area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

c = Circulo(3)
print(c) #__str__ se ha sobreescrito en la subclase
```

En caso de que el método `__init__` de la clase principal no reciba nada, podríamos omitir la llamada `super().__init__`. En el siguiente ejemplo, no se hace la llamada y funciona bien porque el método `__init__` de la subclase ya inicializa los atributos. No obstante, si intentas acceder al atributo `self.tipo` que está definido en la clase principal, verás como te da error porque éste solo se inicializa en el `__init__` de la clase principal:

```
class Figura:
  def __init__(self):#este método no recibe nada
    self.tipo = "default"
    self.area = 0
    self.perimetro = 0

  def __str__(self):
    return "Figura: "+self.tipo+". Area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

class Circulo(Figura):
  PI = 3.14

  def __init__(self, radio):
    self.radio = radio
    self.area = self.PI * self.radio**2
    self.perimetro = 2 * self.PI * self.radio

  def __str__(self): #Sobreescribimos el método en la subclase
    return "Este círculo de radio "+ str(self.radio)+" tiene area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

c = Circulo(3)
print(c) #__str__ se ha sobreescrito en la subclase
```

Si queremos hacer una subclase que sea exactamente igual que su clase principal, podemos poner la palabra `pass` como primer comando dentro de la clase. Es como decir "si, ya sé que la clase está vacía, pero no hagas caso y no me muestres error":

```
class Figura:
  def __init__(self, tipo):
    self.tipo = tipo
    self.area = 0
    self.perimetro = 0

  def __str__(self):
    return "Figura: "+self.tipo+". Area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

class Circulo(Figura): #definimos subclase de Figura pero sin nada adicional
  pass

c = Circulo("circulo")
print(c)
```

1.3 Polimorfismo

El concepto de **poliformismo** hace referencia a poder utilizar diferentes tipos de objetos como parámetros de un mismo nombre de función. En el ejemplo anterior, puedes ver que el método `__str__` tiene un comportamiento distinto en la subclase que en la clase principal. En este caso, estamos haciendo una sobreescritura del método.

Este concepto también es aplicable cuando tenemos una función que se puede llamar pasándole distintos tipos de atributos. En el siguiente ejemplo puedes ver como la función `mostrar_tipo` acepta varios tipos de objetos:

```
class Figura:
  def __init__(self, tipo):
    self.tipo = tipo
    self.area = 0
    self.perimetro = 0

  def __str__(self):
    return "Figura: "+self.tipo+". Area: "+str(self.area)+" y perímetro: "+str(self.perimetro)+" \n"

class Circulo(Figura):
  PI = 3.14

  def __init__(self, radio):
    super().__init__("Circulo")
    self.radio = radio
    self.area = self.PI * self.radio**2
    self.perimetro = 2 * self.PI * self.radio

class Cuadrado(Figura):

  def __init__(self, lado):
    super().__init__("Cuadrado")
    self.lado = lado
    self.area = self.lado * self.lado
    self.perimetro = self.lado *4

#-------------------- main --------------------

def mostrar_tipo(figura):#esta función se ejecuta igual para distintos tipos de objetos pasados (p.e. Circulo y Cuadrado)
  print("El tipo es "+figura.tipo)

c1 = Circulo(2)
mostrar_tipo(c1)
c2 = Cuadrado(2)
mostrar_tipo(c2)
```

El polimorfismo en Python se consigue con lo que se denomina **Duck typing**, que hace referencia a que "Si veo un pájaro que camina como un pato, nada como un pato y hace el mismo sonido que un pato, entonces tiene que ser un pato".

Básicamente, esto significa que la adecuación de un objeto para poderlo utilizar como parámetro de una función se determinará cuando se llame a esa función. A diferencia de otros lenguajes que sí que tienen un tipado estático como Java o C#, en Python, mientras el objeto o variable que reciba la función cuadre con la lógica de la misma, le podemos pasar lo que queramos. Observa el siguiente ejemplo como la función `mostrar_longitud` no le importa el tipo de dato recibido... mientras funcione:

```
def mostrar_longitud(valor):
  print("La longitud es "+str(len(valor)))

palabra = "duck_typing"
mostrar_longitud(palabra)
lista = [5, 4, 7, 1]
mostrar_longitud(lista)
diccionario = {1: "Pepe", 2: "Paco", 3: "Marta"}
mostrar_longitud(diccionario)
```

1.4 Métodos de clase y métodos estáticos

Los métodos que hemos visto en los ejemplos anteriores se denominan **métodos de instancia**. Estos métodos son los más habituales que nos encontraremos. Como ya hemos visto, este tipo de métodos tienen el parámetro `self` como primer parámetro y pueden tener un comportamiento diferente dependiendo del objeto (instancia) que creemos de esa clase.

Además de estos métodos, nos podemos encontrar también con **métodos de clase**. Estos métodos van anotados con el decorador `@classmethod` y en vez de aceptar un parámetro `self`, reciben un parámetro `cls`. Este parámetro apunta a la clase (no al objeto como `self`), por lo que no puede modificar el estado de un objeto. Normalmente, estos métodos se usan como métodos factory, que devuelven un objeto de la clase con una determinada funcionalidad (i.e. se crea un objeto con algún tipo de características).

Finalmente, también podemos tener **métodos estáticos**. Estos métodos van anotados con el decorador `@staticmethod`. Este tipo de métodos no reciben ni `self` ni `cls` y se utilizan principalmente cuando queremos acceder a un método sin necesidad de crear un objeto (una utilidad). Por contra, no puede acceder a atributos ni métodos del propio objeto.

```
from datetime import date

class ClaseEjemplo:
  def __init__(self, nombre, edad):
    self.n = nombre
    self.e = edad

  def metodo_instancia(self):
    print("El nombre es "+self.n)

  @classmethod
  def metodo_clase(cls, nombre, anyo):
    return cls(nombre, date.today().year - anyo)

  @staticmethod
  def metodo_estatico(base, altura):
    print("Este metodo es estático y sirve para alguna funcionalidad que no dependa de ningún objeto en si")
    area = base * altura
    return area

p1 = ClaseEjemplo("Pepe", 20)
p1.metodo_instancia()

p2 = ClaseEjemplo("Ana", 25)
p2.metodo_instancia()
print(type(p2))

p3 = ClaseEjemplo.metodo_clase("Manolo", 30)
p3.metodo_instancia()
print(type(p3))

res = ClaseEjemplo.metodo_estatico(5, 3)
print(res)
```

2 Módulos y paquetes

Como hemos visto, Python es un lenguaje modular que nos permite incluir módulos con la sentencia `import`. Cuando trabajamos con clases, es habitual utilizar varios ficheros, pero la manera de poder acceder desde un fichero a una clase definida en otro fichero es la misma: mediante `import`.

Por tanto, un módulo es un fichero con extensión .py. Si el módulo que queremos importar está en el mismo directorio, simplemente debemos importarlo de la siguiente manera (si son subdirectorios, bastaría con utilizar un punto como separador):

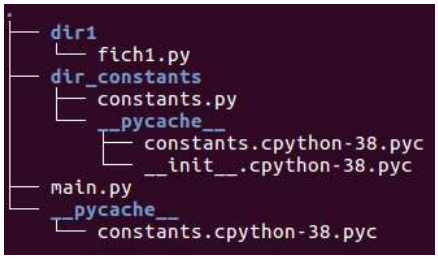
```
#Opción 1 para importar
#-----------------------------
import nombre_modulo #importa todo

#Uso
nombre_modulo.clase

#Opción 2 para importar
#-----------------------------
from nombre_modulo import clase #en este caso, únicamente importamos una clase de ese módulo
from nombre_modulo import clase, clase2 #importa 2 clases
from nombre_modulo import * #importa todo

#Uso
clase
```

El problema de este enfoque es cuando tenemos un código grande distribuido en varios directorios. En este caso, podemos utilizar **paquetes** para gestionar los distintos módulos de manera que los podemos estructurar jerárquicamente. Un paquete es un **directorio** que contiene ficheros .py. Atendiendo a la documentación actual de Python, cuando realizamos un `import paquete_x`:

- Se busca `paquete_x` en los paquetes instalados de Python.
- Si no se encuentra, se busca un fichero `paquete_x.py` en la lista de directorios que apunta la variable `sys.path`. Entre otros, esta variable contiene el directorio del script que importa el `paquete_x`.

En estos directorios, podemos tener un fichero `__init__.py` que se puede utilizar para ejecutar algún código cuando se importe el paquete. Este fichero era obligatorio en versiones anteriores a la 3.3, pero a partir de esta versión, cualquier directorio es considerado un paquete, tenga o no tenga un `__init__.py`.

Para importar un paquete simplemente tenemos que hacerlo igual que un módulo. Imagina que tenemos la siguiente estructura de directorios:



Paquetes

Si desde el fichero `main.py` queremos importar el módulo `constants`, tendríamos que escribir el siguiente código:

```
from dir_constants.constants import *
```

Este código funciona porque Python intenta buscar un fichero `constants.py` en el mismo directorio (y subdirectorios) donde se encuentra `main.py`. No obstante, si queremos importar el módulo desde el fichero `fich1.py`, será necesario configurar la variable `sys.path` para indicarle el directorio donde debe buscar este fichero:

```
import sys
sys.path.insert(1, '../dir_constants')
from constants import *
```

Habría otra opción para importar el paquete y sería generando un paquete distribuible e instalándolo, pero esta opción queda fuera del objetivo de esta sesión.

Por último, en ocasiones encontrarás al final de un fichero .py un código como el siguiente:

```
if __name__ == "__main__":
```

Esto lo entontrarás en muchas ocasiones y básicamente se utiliza cuando un fichero puede usarse **como módulo** importado **y también como ejecutable independiente**. Si ejecutas el fichero directamente, no hay diferencia entre poner el código que quieres utilizar dentro de este bloque o fuera. Sin embargo, si este fichero es importado, podrás comprobar que si no tiene este bloque, se ejecuta el código que haya en un nivel principal.

Esto ocurre porque cuando Python lee un fichero, asigna a la variable `__name__` un valor y después ejecuta el fichero. En el caso de ejecutar directamente el fichero, Python asigna el string `"__main__"` a la variable `__name__` y después ejecuta el código que haya dentro del fichero. Cuando importamos un fichero (p.e. `funciones.py`), a la variable `__name__` se le asigna el nombre del fichero en sí (i.e. `funciones`). Por tanto, el hecho de poner el código dentro de este bloque hace que solo se ejecute cuando el fichero se ejecuta directamente y no cuando es importado.

2.1 Constantes

En Python, las constantes normalmente se definen dentro de un módulo. Por convenio, una constante se define con mayúsculas. Imaginemos que tenemos un fichero `constants.py` con el contenido que hay a continuación:

```
PI = 3.14
```

Para utilizar esta constante desde otro fichero, bastaría con realizar el import correspondiente y acceder de cualquiera de las siguientes maneras:

```
#Opción 1
from constants import *

print("El valor es "+str(PI))
```

```
print("El valor es "+str(PI))

#Opción 2
import constants

print("El valor es "+str(constants.PI))
```

En el caso de tener varias variables, es posible agruparlas dentro de una clase, de manera que las constantes se definan como atributos de clase (i.e. atributos definidos a nivel de clase sin `self`). En el ejemplo siguiente, estaríamos definiendo varios estados posibles de alguna máquina de estados:

```
class states:
    INIT = 0
    ACTIVE = 1
    IDDLE = 2
    STOP = 3
```

En este caso, la manera de utilizar estas constantes sería similar a la anterior, pero indicando también la referencia del nombre de la clase:

```
#Opción 1
from constants import *

print("El valor es "+str(states.INIT))

#Opción 2
import constants

print("El valor es "+str(constants.states.INIT))
```

3 Tarea entregable

En esta tarea vamos a practicar todo lo que hemos visto hasta ahora, programando una aplicación de reservas de vuelos. En la siguiente imagen puedes ver el diagrama de clases que tendrás que desarrollar.

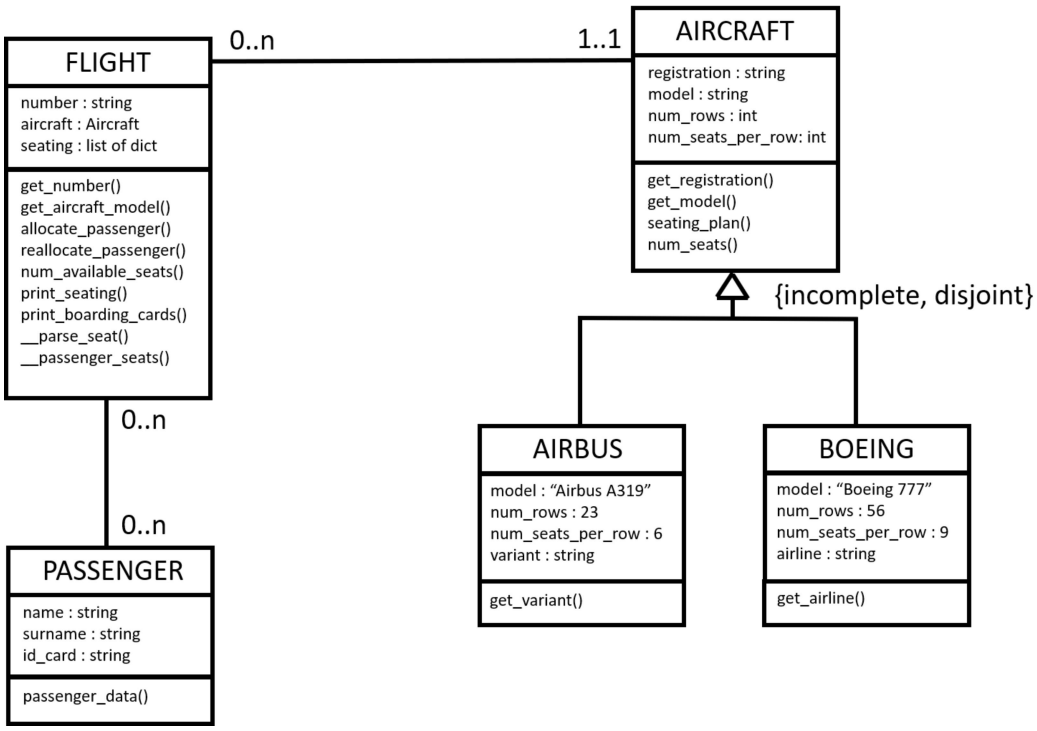


Diagrama de clases

Como ves, se trata de implementar varias clases con varios métodos. Algunas consideraciones a tener en cuenta:

- Cada clase se implementará en un fichero con el mismo nombre. Los nombres de las clases serán los mismos que se ven en el diagrama de clases, siguiendo el convenido de nomenclatura (sólo la primera letra en mayúscula), mientras que los nombres de los ficheros serán en minúscula.

- Los nombres de los parámetros de los `__init__` serán los mismos que los atributos de la clase que definen.
- Las clases **Airbus** y **Boing** se implementarán en el fichero de la superclase **Aircraft**.
- Los atributos de cada clase serán privados. Su nomenclatura empezará por `__` y se utilizarán los *getters* para acceder a su valor.
- Por el momento, no hace falta ningún tipo de comprobración de errores.

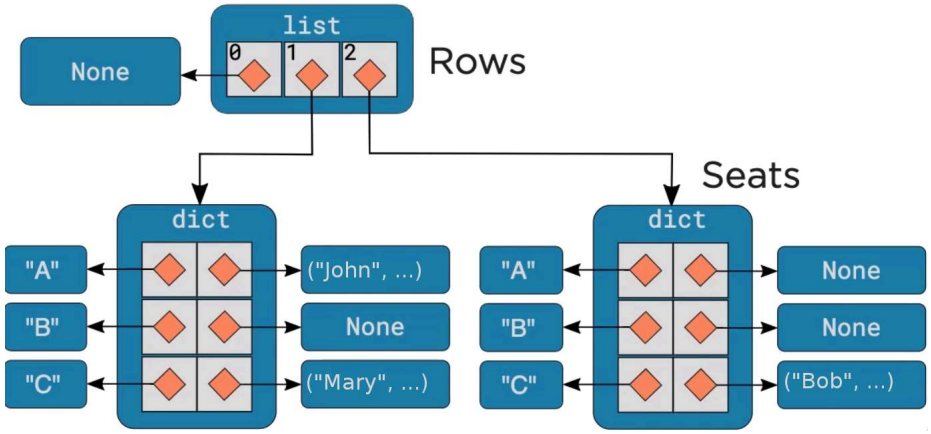
La clase **Aircraft** contiene información de la aeronave. De esta clase, cabe mencionar el método `seating_plan()` que genera una tupla con dos valores. El primero es una lista con el tamaño de filas + 1, siendo `None` el valor de cada elemento de la lista (ten en cuenta que estamos añadiendo un valor de más, pero que no será una fila real que esté disponible). El segundo es un string de letras (p.e. "ABCDEF"), que representa los asientos de cada fila. El método `num_seats` simplemente devuelve el número total de asientos reales que tiene la aeronave. A continuación, puedes ver la descripción de ambos métodos:

```
seating_plan()
    """Generates a seating plan for the number of rows and seats per row
    Returns:
      rows: A list of Nones (size num_rows + 1).
      seats: A string of letters such as "ABCDEF"
    """
num_seats()
    """Calculates the number of seats
    Returns:
      seats: The number of seats
    """
```

La clase **Passenger** se utiliza para almacenar la información de cada pasajero: nombre, apellido e id. Esta clase tiene un método `passenger_data()` que devuelve una tupla con los tres valores:

```
passenger_data()
    """Obtains the data of a passenger
    Returns:
      name: The passenger's name such as 'Jack'
      surname: The passenger's family name such as 'Shephard'
      id_card: The passenger's id card such as '85994003S'
    """
```

La clase **Flight** representa la información de un vuelo. Posiblemente, la parte más complicada es la representación de los asientos, que puedes encontrar en el atributo `seating`. Esto se representa mediante una lista de diccionarios, donde cada posición de la lista hace referencia a una fila de asientos (1, 2, etc.). Para cada fila, habrá un diccionario donde las claves serán las letras de los asientos ('A', 'B', etc.) y los valores serán los datos del pasajero (nombre, apellido e id). Para ello, debes utilizar la información que te devuelve `seating_plan`. Como la numeración de las filas empieza en 1, mantenemos la fila 0 con un valor `None` pero que no será utilizable. En la siguiente imagen puedes ver una representación gráfica (Nota: en la imagen solo aparece el nombre del pasajero):



Representación del plan de reservas

Por defecto, todos los asientos estarán a `None` hasta que se asigne un pasajero. A continuación puedes ver los métodos que nos permiten gestionar los asientos:

```
allocate_passenger()
    """Allocate a seat to a passenger
    Args:
```

```
      seat: A seat designator such as '12C' or '21F'
      passenger: The passenger data such as ('Jack', 'Shephard', '85994003S')
    """
reallocate_passenger()
    """Reallocate a passenger to a different seat
    Args:
      from_seat: The existing seat designator for the passenger such as '12C'
      to_seat: The new seat designator
    """
num_available_seats()
    """Obtains the amount of unoccupied seats
    Returns:
      The number of unoccupied seats
    """
```

Además, también tenemos dos métodos que nos permiten imprimir información por consola. El método `print_seating()` mostrará el listado de asientos por consola. Para mostrar las filas una detrás de otra, puedes importar la función `pprint()` del módulo `pprint`:

```
from pprint import pprint

pprint(seating)
```

También tenemos el método `print_boarding_cards()` que mostrará por pantalla las tarjetas de embarque de todos los pasajeros. A continuación tienes la definición de estos métodos. Es importante que `print_boarding_cards()` devuelva algo similar a lo que muestra la función (una especie de tarjeta y no solo el nombre):

```
print_seating()
    """Prints in console the seating plan
    Example of one row:
      {'A': None, 'B': None, 'C': None, 'D': None, 'E': None, 'F': None}
    """
print_boarding_cards()
    """Prints in console the boarding card for each passenger
    Example of one boarding card:
    ---------------------------------------------------------
    |     Jack Sheppard 85994003S 15E BA758 Airbus A319      |
    ---------------------------------------------------------
    """
```

También se pide implementar dos métodos privados que serán usados por otros métodos de la clase:

```
__parse_seat()
    """Divide a seat designator in row and letter
    Args:
      seat: The seat designator to be divided such as '12C'
    Returns:
      row: The row of the seat such as 12
      letter: The letter of the seat such as 'C'
    """
__passenger_seats()
    """A generator function to iterate the occupied seating locations
    Returns:
      generator: Tuple of the passenger data and the seat
    """
```

Finalmente, aunque a ti no te haga falta, debes implementar explícitamente el método `get_seating()`.

Para realizar pruebas, deberás definir un fichero `main.py`, que tendrá una estructura similar a la siguiente (puedes copiarla y pegarla con los imports necesarios). Amplíala con lo que necesites, pero el código este que te proporciono, te debería funcionar:

```
    def make_flights():
        f1 = Flight(number = "BA117", aircraft = Aircraft(registration = "G-EUAH", model = "Airbus A319", num_rows = 22, num_seats_
        f2 = Flight(number = "AF92", aircraft = Boeing(registration = "F-GSPS", airline = "Emirates"))
        f3 = Flight(number = "BA148", aircraft = Airbus(registration = "G-EUPT", variant = "A319-100"))
```

```
    p1 = Passenger("Jack", "Shephard", "85994003S")
    p2 = Passenger("Kate", "Austen", "12589756P")
    p3 = Passenger("James", "Ford", "56278665F")
    p4 = Passenger("John", "Locke", "10265448H")
    p5 = Passenger("Sayid", "Jarrah", "15758664M")

    f1.allocate_passenger("12A", p1.passenger_data())
    f1.allocate_passenger("18F", p2.passenger_data())
    f1.allocate_passenger("18E", p3.passenger_data())
    f1.allocate_passenger("1C", p4.passenger_data())
    f1.allocate_passenger("4D", p5.passenger_data())

    return f1, f2, f3

f1, f2, f3 = make_flights()
for fl in f1, f2, f3:
  fl.print_seating()
  fl.print_boarding_cards()
```