1. Instanciación de un objeto

Como se mencionó en el primer apunte, todo objeto que interactúa en un sistema orientado a objetos tiene un comportamiento que está dado por las funciones ("métodos") que ejecuta cuando recibe solicitudes ("mensajes") de otros objetos.

También durante la instanciación de los objetos se ejecutan ciertas instrucciones, por ejemplo, para inicializar sus atributos. Este comportamiento en `Python` está codificado en los inicializadores `__init__`, estos no tienen valor de retorno (`None`) y deben recibir el atributo `self` que es la instancia de su misma clase (es decir el objeto) que se está instanciando. Si el objeto no tiene atributos, entonces puede obviarse este método, ya que se ejecutará uno genérico que no tiene atributos.

Por ejemplo, en el Dado presentado en el primer apunte no había atributos, solo comportamiento, por lo tanto no se llamaba al método `__init__`, pero al agregar un atributo (`ncaras`), este se hizo necesario.

```
from random import randint
class Dado:
  def tirar(self) -> int:
    return randint(1, 6)
```

```
from random import randint
class Dado:
  def __init__(self, ncaras : int) -> None:
    self.__ncaras = ncaras
  def tirar(self) -> int:
    return randint(1, self.__ncaras)
```

La llamada a `__init__` es automática al momento de construir un objeto utilizando `NombreClase()`, donde dentro del paréntesis debe ofrecer los argumentos para los atributos declarados (exceptuando `self`).

```
# ...
if __name__ == "__main__":
  d1 = Dado(5)  # llamada implicita a Dado.__init__(d1, 5)
  d2 = Dado(24) # llamada implicita a Dado.__init__(d2, 24)
  d1.tirar()    # Dado.tirar(d1)
  d2.tirar()    # Dado.tirar(d2)
```

Cada método debe limitarse a realizar una única tarea bien definida, y su nombre

debe expresar esa tarea con efectividad. Esto hace que los programas sean más fáciles de escribir, depurar, mantener y modificar.

1.1. Características de los métodos

Los métodos pueden devolver como máximo un valor, pero el valor devuelto puede ser una referencia a un objeto que contenga varios valores (por ejemplo una lista o tupla u objeto cualquiera). El tipo del valor devuelto se suele indicar luego de cerrar el paréntesis con una flecha seguida del tipo de dato (-> `<tipoDato>`). Como ya se explicó anteriormente, la palabra reservada `None` se utiliza para indicar que un método no devuelve ningún valor.

Los métodos (y también los inicializadores) pueden tener variables locales. Solamente es posible acceder a las variables locales dentro del ámbito en que están declaradas, y al finalizar la ejecución del método al que pertenecen, el valor de las mismas no se mantiene.

Los parámetros de cualquier método pueden tener un valor por omisión (*default*) esto se hace igualándolos a un valor en la declaración del mismo.

```
1  from random import randint
2  class Dado:
3    def __init__(self, ncaras : int = 6) -> None:
4      self.__ncaras = ncaras
5    def tirar(self) -> int:
6      return randint(1, self.__ncaras)
7
8  if __name__ == "__main__":
9    d1 = Dado() # Dado.__init__(d1, 6)
10   d2 = Dado(24) # Dado.__init__(d2, 24)
11   d1.tirar()
12   d2.tirar()
```

En el caso de omitir `ncaras` al instanciar un `Dado`, este tomará el valor de 6.

2. Tipos de métodos

Existe tres maneras de invocar métodos que se analizarán a continuación.

2.1. Métodos comunes

Estos son los comportamientos más habituales, de los cuales otros objetos deben ser capaces de mandar el mensaje para ejecutar.

Esto se hace pasándole un mensaje a un objeto, es decir, usando una variable que se refiera al objeto, seguida de punto (.) y el nombre del método.

```
# d1 objeto tipo Dado
d1.tirar() # Envía el mensaje "tirar"
```

2.2. Métodos internos

Desde el resto de los objetos de un sistema se debería poder solicitarle a un objeto la ejecución de sus métodos. Una excepción son aquellos métodos auxiliares que sólo se invocarán desde dentro del propio objeto, en cuyo caso se los debe hacer *invisibles* para los demás objetos. Si bien en `Python` no existe un mecanismo como tal, esto lo haremos agregando un doble guion bajo delante del nombre del método, esto se estudiará en detalle en el apunte 5 sobre **encapsulamiento**.

Para invocar estos métodos debemos utilizar la variable de instancia `self` y luego el operador punto (.) para invocar al método interno con su nombre. Esto solo se puede hacer dentro de la clase, evitando así su público acceso desde fuera.

```
class Perro:
    def __init__(self, ladrido : str) -> None:
        self.__ladrido = ladrido
    def ladrar(self) -> None:
        print(self.__ladrido)
    def ladrarFuerte(self) -> None:
        print(self.__ladridoAMayusculas())
    def __ladridoAMayusculas(self) -> str:
        return self.__ladrido.upper()

if __name__ == "__main__":
    dogo_argentino = Perro("Guau! Guau!")
    pastor_ingles = Perro("Woof! Woof!")
    dogo_argentino.ladrar()
    pastor_ingles.ladrar()
    dogo_argentino.ladrarFuerte()
    pastor_ingles.ladrarFuerte()
```

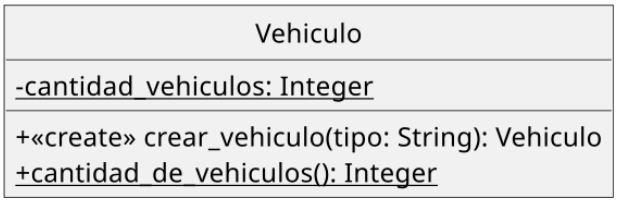
Si se ejecutara este código, se vería resultado a continuación.

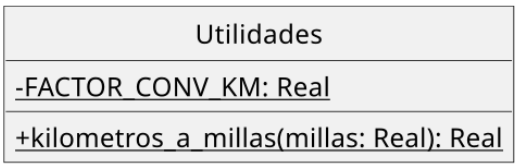
```
Guau! Guau!
Woof! Woof!
GUAU! GUAU!
WOOF! WOOF!
```

2.3. Métodos de clase y estáticos

Los métodos de clase y los estáticos son aquellos que pueden ser invocados sin la necesidad de instanciar un objeto, simplemente anteponiendo el nombre de la clase y el nombre del método a través del operador punto (.).

Al igual que los atributos de clase que se consideran estáticos (*static*) en los diagramas **UML** se indican subrayándolos.





2.3.1. Métodos de clase

Estos métodos no reciben el parámetro `self`, sino que reciben `cls` (*class*) y son marcados con el decorador `@classmethod`. Estos métodos pueden acceder al atributos de la clase a través de la variable `cls`.

Programa 1: Método de clase

```
1  class Vehiculo:
2      __cantidad_de_vehiculos = 0
3
4      def __init__(self, tipo : str) -> None:
5          self.__tipo = tipo
6          Vehiculo.__cantidad_de_vehiculos += 1
7
8      @classmethod
9      def cantidad_de_vehiculos(cls) -> int:
10         return cls.__cantidad_de_vehiculos
11
12 if __name__ == "__main__":
13     # Crear instancias de la clase
```

```
14      vehiculo1 = Vehiculo("Auto")
15      vehiculo2 = Vehiculo("Motocicleta")
16
17      # Uso del método de clase
18      numero_de_vehiculos = Vehiculo.cantidad_de_vehiculos()
19      print(f"Se han creado {numero_de_vehiculos} vehículos")
```

2.3.2. Métodos estáticos

Los métodos estáticos están presentes dentro de la clase, y pueden acceder a sus variables de clase, pero deben hacerlo a través de su nombre, a diferencia del método de clase que accede a través de la referencia `cls`.

Programa 2: Método estático

```
1  class Utilidades:
2      __FACTOR_CONV_KM = 1.60934
3
4      @staticmethod
5      def millas_a_kilometros(millas : float) -> float:
6          return millas * Utilidades.__FACTOR_CONV_KM
7
8  if __name__ == "__main__":
9      # Uso del método estático sin crear una instancia de la clase
10     resultado = Utilidades.millas_a_kilometros(10)
11     print(f"10 millas son {resultado} kilómetros")
```

Esta diferencia sutil se retomará en el próximo apunte cuando se analice la **herencia**.

3. Métodos especiales

Los métodos especiales en `Python`, también conocidos como métodos mágicos o *magic methods*, son funciones que tienen nombres especiales que comienzan y terminan con dobles guiones bajos (__). Estos métodos permiten que las instancias de las clases tengan comportamientos específicos cuando se utilizan con ciertas operaciones o funciones incorporadas.

3.1. `__init__`

El método `__init__` es el inicializador de la clase y se llama automáticamente cuando se crea una nueva instancia de la clase. Esto ya ha sido explicado con anterioridad.

```
1  class Persona:
2      def __init__(self, nombre : str, edad : int) -> None:
3          self.__nombre = nombre
4          self.__edad = edad
5
6  if __name__ == "__main__":
7      p = Persona("Juan", 30)
```

3.2. `__del__`

El método `__del__` se llama cuando una instancia es liberada de la memoria. Podemos forzar este comportamiento utilizando la palabra reservada `del`

```
1  class Persona:
2      def __init__(self, nombre : str, edad : int) -> None:
3          self.__nombre = nombre
4          self.__edad = edad
5
6      def __del__(self) -> None:
7          print(f'Adiós, {self.__nombre}')
8
9  if __name__ == "__main__":
10     p = Persona("Juan", 30)
11     del p  # Adiós, Juan
```

3.3. `__hash__`

El método `__hash__` otorga un valor único para cada objeto, si no se define por *default* devuelve el resultado de la función integrada `hash(self)`. Se puede personalizar generando un código único dependiendo de la implementación del sistema.

```
1  import random
2  class Persona:
3      def __init__(self, nombre : str, edad : int) -> None:
4          self.__nombre = nombre
5          self.__edad = edad
```

```
6           # CUIDADO: no asegura que sea único y debería serlo!!
7           # Es solo un ejemplo.
8           self.__id = int(random.random()*1_000_000_000)
9
10      def __hash__(self) -> int:
11          return self.__id
12
13  if __name__ == "__main__":
14      p1 = Persona("Juan", 30)
15      p2 = Persona("Juan", 30)
16
17      print(hash(p1))
18      print(hash(p2))
```

3.4. `__repr__` y `__str__`

El método `__repr__` proporciona una representación formal y detallada de una instancia de clase. Debe ser información específica, ya que se utiliza habitualmente para depurar. Si no está definida `__str__` se se llamará a este método cuando se intente convertir el objeto en *string*.

```
1   class Persona:
2       def __init__(self, nombre : str, edad : int) -> None:
3           self.__nombre = nombre
4           self.__edad = edad
5
6       def __repr__(self) -> str:
7           return f'Persona(nombre={self.__nombre}, ' \
8                  f'edad={self.__edad})' \
9                  f'id={self.__hash__()}'
10
11  if __name__ == "__main__":
12      p1 = Persona("Juan", 30)
13      p2 = Persona("Juan", 30)
14      print(p1)  # Persona(nombre=Juan, edad=30, id=####)
15      print(p2)  # Persona(nombre=Juan, edad=30, id=####)
```

El método `__str__` define la representación en forma de cadena de una instancia. Siempre que se intente convertir a `str()` será llamada esta función.

```
1   class Persona:
2       def __init__(self, nombre : str, edad : int) -> None:
3           self.__nombre = nombre
4           self.__edad = edad
5
```



```
6       def __str__(self) -> str:
7           return f'{self.__nombre}, {self.__edad} años'
8
9   if __name__ == "__main__":
10      p1 = Persona("Juan", 30)
11      p2 = Persona("Juan", 30)
12      print(p1)  # Juan, 30 años
13      print(p2)  # Juan, 30 años
```

3.5. `__bool__`

El método `__bool__` se utiliza para definir el comportamiento de una instancia al evaluarse en un contexto booleano.

```
1   class Persona:
2       def __init__(self, nombre : str, edad : int) -> None:
3           self.__nombre = nombre
4           self.__edad = edad
5
6       def __bool__(self) -> bool:
7           return self.__edad > 0
8
9   if __name__ == "__main__":
10      p = Persona("Juan", 30)
11      if p:
12          print("+1 año de vida")
13      else:
14          print("-1 año de vida")
```

3.6. Operadores de comparación

En esto métodos además de recibir la instancia `self`, se recibe otra instancia del mismo objeto, para ello se utilizará el *typehint* `Self` que está dentro del módulo `typing`.

El método `__lt__` se utiliza para definir el comportamiento del operador menor que (<).

```
1   from typing import Self
2   class Persona:
3       def __init__(self, nombre : str, edad : int) -> None:
4           self.__nombre = nombre
5           self.__edad = edad
6
```

```
    def __lt__(self, other : Self) -> bool:
        return self.__edad < other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 25)
    print(p1 < p2)  # False
```

El método `__le__` se utiliza para definir el comportamiento del operador menor o igual que (<=).

```
from typing import Self
class Persona:
    def __init__(self, nombre : str, edad : int) -> None:
        self.__nombre = nombre
        self.__edad = edad

    def __le__(self, other : Self) -> bool:
        return self.__edad <= other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 30)
    print(p1 <= p2)  # True
```

El método `__eq__` se utiliza para definir el comportamiento del operador de igualdad (==).

```
from typing import Self
class Persona:
    def __init__(self, nombre : str, edad : int) -> None:
        self.__nombre = nombre
        self.__edad = edad

    def __eq__(self, other : Self) -> bool:
        return self.__edad == other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 30)
    print(p1 == p2)  # True
```

El método `__ne__` se utiliza para definir el comportamiento del operador de desigualdad (!=).

```
from typing import Self

class Persona:
```

```
    def __init__(self, nombre : str, edad : int) -> None:
        self.__nombre = nombre
        self.__edad = edad

    def __ne__(self, other : Self) -> bool:
        return self.__edad != other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 25)
    print(p1 != p2)  # True
```

El método `__gt__` se utiliza para definir el comportamiento del operador mayor que (>).

```
from typing import Self
class Persona:
    def __init__(self, nombre : str, edad : int) -> None:
        self.__nombre = nombre
        self.__edad = edad

    def __gt__(self, other : Self) -> bool:
        return self.__edad > other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 25)
    print(p1 > p2)  # True
```

El método `__ge__` se utiliza para definir el comportamiento del operador mayor o igual que (>=).

```
from typing import Self
class Persona:
    def __init__(self, nombre : str, edad : int) -> None:
        self.__nombre = nombre
        self.__edad = edad

    def __ge__(self, other : Self) -> bool:
        return self.__edad >= other.__edad

if __name__ == "__main__":
    p1 = Persona("Juan", 30)
    p2 = Persona("Ana", 25)
    print(p1 >= p2)  # True
```

3.7. Operadores matemáticos

Estos métodos devuelven habitualmente una nueva instancia del mismo objeto (`Self`) resultado de la operación.

El método `__add__` se utiliza para definir el comportamiento del operador de adición (+).

```
from typing import Self
class Vector:
    def __init__(self, x : float, y : float) -> None:
        self.__x = x
        self.__y = y

    def __add__(self, other : Self) -> Self:
        return Vector(self.__x + other.__x, self.__y + other.__y)

    def __str__(self) -> str:
        return f'({self.__x},{self.__y})'

if __name__ == "__main__":
    v1 = Vector(1, 2)
    v2 = Vector(3, 4)
    v3 = v1 + v2
    print(v3) # (4, 6)
```

El método `__sub__` se utiliza para definir el comportamiento del operador de sustracción (-).

```
from typing import Self
class Vector:
    def __init__(self, x : float, y : float) -> None:
        self.__x = x
        self.__y = y

    def __sub__(self, other : Self) -> Self:
        return Vector(self.__x - other.__x, self.__y - other.__y)

    def __str__(self) -> str:
        return f'({self.__x},{self.__y})'

if __name__ == "__main__":
    v1 = Vector(3, 4)
    v2 = Vector(1, 2)
    v3 = v1 - v2
    print(v3) # (2,2)
```

El método `__mul__` se utiliza para definir el comportamiento del operador de multiplicación (`*`).

```python
from typing import Self
class Vector:
    def __init__(self, x : float, y : float) -> None:
        self.__x = x
        self.__y = y

    def __mul__(self, escalar : float) -> Self:
        return Vector(self.__x * escalar, self.__y * escalar)

    def __str__(self) -> str:
        return f'({self.__x},{self.__y})'

if __name__ == "__main__":
    v1 = Vector(1, 2)
    v2 = v1 * 3
    print(v2) # (3,6)
```

El método `__truediv__` se utiliza para definir el comportamiento del operador de división (`/`).

```python
from typing import Self

class Fraccion:
  def __init__(self,numerador:float,denominador:float) -> None:
    self.__num = numerador
    self.__den = denominador

  def __truediv__(self, other : Self) -> Self:
    nuevo_num = self.__num * other.__den
    nuevo_den = self.__den * other.__num
    return Fraccion(nuevo_num, nuevo_den)

  def __str__(self) -> str:
    return f'{self.__num}/{self.__den}'

if __name__ == "__main__":
    f1 = Fraccion(1, 2)
    f2 = Fraccion(3, 4)
    f3 = f1 / f2
    print(f3)  # 4/6
```

El método `__floordiv__` se utiliza para definir el comportamiento del operador de división entera (`//`) y `__mod__` se utiliza para definir el comportamiento del operador de módulo (`%`).

4. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: https://docs.python.org/3/reference/datamodel.html
- UBA – Algoritmos I: https://algoritmos1rw.ddns.net/
- Interprete Visual de Python: http://www.pythontutor.com/visualize.html