Clase 15 – POO II

IIC1103 Sección 9 – 2019-2

Profesor: Felipe López

21-11-2019

Resumen

- Las clase en son representaciones abstractas en Python.
- Una instancia de una clase con atributos específicos es un objeto.
- Se puede interactuar con una clase mediante sus métodos. Estos también permiten la interacción entre distintas clases.

- En Python se definen de la siguiente manera:

```
class Nombre_clase:
  def __init__(self,…):
        self.atributo1
        self.atributo2
                …
  def metodo1(self,…):

  def metodo2(self,…):
```

__str(self)__

- Es posible “sobrecargar”[1] el método (str) de un objeto para que retorne lo que nosotros queramos.
- En este caso, definimos el siguiente método de una clase:

```
def __str__(self):
    …
    return (string)
```

- Esto permitirá poder ocupar `print()` sobre un objeto de esa clase y obtener lo que retorne este método. También permitirá obtener información cada vez que hagamos `str()` a este objeto.

[1] dar una nueva funcionalidad a un método ya existente

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).

```
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def __str__(self):
    return "("+str(self.x)+","+str(self.y)+")"
```

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).

```
p1 = Punto(1,2)
print(p1)
p2 = Punto(3,4)
print(p2)
p3 = Punto(5,6)
print(p3)
```

```
(1,2)
(3,4)
(5,6)
```

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).

```
p1 = Punto(1,2)
print(p1)
p2 = Punto(3,4)
print(p2)
p3 = Punto(5,6)
print(p3)
```

```
(1,2)
(3,4)
(5,6)
```

¿Y si tuviéramos muchos puntos? Nos gustaría almacenarlos en una lista

Listas de objetos

- Al igual que cualquier tipo de datos, una lista también puede almacenar objetos.
- Por esto es importante considerar una clase como un “nuevo tipo” que uno puede crear (¡Lo que hace mucho sentido con el método __str(self)__!).
- Si quisiéramos almacenar los puntos creados en el ejemplo anterior en una lista:

```
p1 = Punto(1,2)
print(p1)
p2 = Punto(3,4)
print(p2)
p3 = Punto(5,6)
print(p3)

lista_puntos = [p1,p2,p3]
print(lista_puntos)
```

```
(1,2)
(3,4)
(5,6)
[<__main__.Punto object at
0x7fd131af1dd0>,
<__main__.Punto object at
0x7fd131ae9cd0>,
<__main__.Punto object at
0x7fd131af1d10>]
```

Listas de objetos

- Al igual que cualquier tipo de datos, una lista también puede almacenar objetos.
- Por esto es importante considerar una clase como un “nuevo tipo” que uno puede crear (¡Lo que hace mucho sentido con el método __str(self)__!).
- Si quisiéramos almacenar los puntos creados en el ejemplo anterior en una lista:

A pesar de haber definido un método str no se imprimen los puntos en consola como nos gustaría

```
sta_puntos)
(1,
(3,4)
(5,6)
[<__main__.Punto object at
0x7fd131af1dd0>,
<__main__.Punto object at
0x7fd131ae9cd0>,
<__main__.Punto object at
0x7fd131af1d10>]
```

__repr(self)__

- Similar a __str(self)__, este método nos sirve para retornar un string customizado en un objeto de una clase.
- A diferencia de __str(self)__, este método **sí** retorna un string customizado cuando se está imprimiendo una lista con objetos de esta clase.
- Por ejemplo:

```
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def __repr__(self):
    return "("+str(self.x)+","+str(self.y)+")"

p1 = Punto(1,2)
print(p1)
p2 = Punto(3,4)
print(p2)
p3 = Punto(5,6)
print(p3)

lista_puntos = [p1,p2,p3]
print(lista_puntos)
```

```
(1,2)
(3,4)
(5,6)
[(1,2), (3,4), (5,6)]
```

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).
2. Una clase Plano, que tenga una lista de listas con coordenadas. Puede asumir que el usuario siempre agregará un punto como una lista de dos elementos con las coordenadas x e y (en ese orden).

```
class Plano:
  def __init__(self):
    self.coordenadas = []

  def agregar_punto(self,punto):
    self.coordenadas.append(punto)

  def __repr__(self):
    return str(self.coordenadas)

plano = Plano()
plano.agregar_punto([1,2])
plano.agregar_punto([3,4])
plano.agregar_punto([5,6])
print(plano)
```

```
[[1,2], [3,4], [5,6]]
```

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).
2. Una clase Plano, que tenga una lista de listas con coordenadas. Puede asumir que el usuario siempre agregará un punto como una lista de dos elementos con las coordenadas x e y (en ese orden).
   a) Agregue a su código alguna condición de tal manera que no se puedan agregar más de 3 puntos en el plano.

```python
class Plano:
  def __init__(self):
    self.coordenadas = []

  def agregar_punto(self,punto):
    if len(self.coordenadas)<3:
      self.coordenadas.append(punto)

  def __repr__(self):
    return str(self.coordenadas)

plano = Plano()
plano.agregar_punto([1,2])
plano.agregar_punto([3,4])
plano.agregar_punto([5,6])
print(plano)
plano.agregar_punto([7,8])
print(plano)
```

```
[[1,2], [3,4], [5,6]]
[[1,2], [3,4], [5,6]]
```

Atributos de una clase: Listas

- En la clase pasada vimos que los atributos de una clase pueden ser int, float, string, y bool.
- Como se puede observar en el ejemplo anterior, los atributos de una clase también pueden ser listas (de listas, de listas…).
- Se trabaja con ellos igual que como hemos aprendido a trabajar con las listas durante el curso.

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).
   c) Agregue a la clase Punto un método que reciba las coordenadas de otro Punto, y calcule su distancia con este.
2. Una clase Plano, que tenga una lista de listas con coordenadas. Puede asumir que el usuario siempre agregará un punto como una lista de dos elementos con las coordenadas x e y (en ese orden).
   a) Agregue a su código alguna condición de tal manera que no se puedan agregar más de 3 puntos en el plano.

```
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def distancia_otro(self,x2,y2):
    return math.sqrt((y2-self.y)**2+(x2-
self.x)**2)

punto1 = Punto(0,0)
print(punto1.distancia_otro(3,4))
```

```
5.0
```

Ejercicio 1

1. Crear:
   a) Una clase Punto, que tenga dos atributos (coordenada x y coordenada y).
   b) Tres objetos de la clase punto, con coordenadas (1,2),(3,4),(5,6).
   c) Agregue a la clase Punto un método que reciba las coordenadas de otro Punto, y calcule su distancia con este.
   d) Agregue a la clase Punto un método que reciba otro Punto, y calcule su distancia con este.
2. Una clase Plano, que tenga una lista de listas con coordenadas. Puede asumir que el usuario siempre agregará un punto como una lista de dos elementos con las coordenadas x e y (en ese orden).
   a) Agregue a su código alguna condición de tal manera que no se puedan agregar más de 3 puntos en el plano.

```
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def distancia_otro(self,otro_punto):
    return math.sqrt((otro_punto.y-self.y)**2+(otro_punto.x-
self.x)**2)

punto1 = Punto(0,0)
punto2 = Punto(3,4)
print(punto1.distancia_otro(punto2))
```

```
5.0
```

Interacciones entre clases

- La esencia de la programación orientada a objetos es poder modelar la realidad.
- Esto implica el poder hacer que las clases interactúen entre ellas.
- Un tipo de interacción es que un método de una clase reciba como parámetro un objeto. **Este objeto puede ser de la misma clase u otra clase.** Tal como vimos en el ejemplo anterior.

Ejercicio 2

1. Crear una clase Punto:
   a) Que tenga dos atributos (coordenada x y coordenada y).
   b) Que tenga un método calcular_distancia que calcule la distancia respecto a otro Punto (objetos de la clase Punto).
   c) Que tenga un método __repr__, que represente al punto en este formato: (x,y)
2. Crear una clase Plano:
   a) Que tenga como atributo:
      a) Un punto (que será el centro del plano).
      b) Una lista de Puntos (objetos de la clase Punto anterior).
   b) Que tenga un método __str__, que retorne la lista de puntos del plano.

```python
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def distancia_otro(self,otro_punto):
    return math.sqrt((otro_punto.y-
self.y)**2+(otro_punto.x-self.x)**2)

  def __repr__(self):
    return "("+str(self.x)+","+str(self.y)+")"

punto0 = Punto(0,0)
plano = Plano(punto0)
punto1 = Punto(1,1)
punto2 = Punto(3,4)
plano.lista_puntos.append(punto1)
plano.lista_puntos.append(punto2)
print(plano)
```

```
[(1,1), (3,4)] Centro: (0,0)
```

```python
class Plano:
  def __init__(self,punto):
    self.lista_puntos = []
    self.centro = punto

  def __str__(self):
    return str(self.lista_puntos)+" Centro:
"+punto
```

Interacciones entre clases

- Un atributo de una clase puede ser no solo de tipo int, float, string, bool o una lista, sino que también otro objeto.
- Además, un atributo de tipo lista también puede contener objetos.
- Esto nos permite ampliar las interacciones entre distintas clases.
- Nuevamente, no existe restricción alguna. Es decir, **los atributos de una clase pueden ser un objeto de la misma clase o de otras.** Y un **atributo que contenga una lista de objetos, puede contener objetos de la misma clase y/o otra.**

Ejercicio 3

1. Crear una clase Punto:
   a) Que tenga dos atributos (coordenada x y coordenada y).
   b) Que tenga un método calcular_distancia que calcule la distancia respecto a otro Punto (objetos de la clase Punto).
   c) Que tenga un método __repr__, que represente al punto en este formato: (x,y)
2. Crear una clase Plano:
   a) Que tenga como atributo:
      a) Una lista de Puntos (objetos de la clase Punto anterior).
      b) distancia_maxima, una variable de tipo float que tenga la distancia entre cualquier par de puntos contenido en el plano.
   b) Un método agregar_punto que:
      a) Revise si el punto existe en el plano y solo en caso que no, lo agregue. Debe retornar True o False dependiendo si el el punto se pudo agregar al plano o no.
      b) Que calcule la distancia con todos los puntos que ya tiene el plano para ir actualizando el atributo distancia_máxima.
   c) Que tenga un método __str__, que retorne la lista de puntos del plano.

```
class Plano:
  def __init__(self):
    self.lista_puntos = []
    self.distancia_maxima = 0

  def __str__(self):
    return str(self.lista_puntos)+" Dist. Máx:
"+str(self.distancia_maxima)

  def agregar_punto(self,punto):
    existe = False
    for i in self.lista_puntos:
      if i.x == punto.x and i.y == punto.y:
        existe = True

    if not existe:
      for i in self.lista_puntos:
        if i.distancia_otro(punto) > self.distancia_ma-
xima:
          self.distancia_maxima = i.distan-
cia_otro(punto)
      self.lista_puntos.append(punto)
      return True
    else:
      return False
```

```
class Punto:
  def __init__(self,x,y):
    self.x = x
    self.y = y

  def distancia_otro(self,otro_punto):
    return math.sqrt((otro_punto.y-
self.y)**2+(otro_punto.x-self.x)**2)

  def __repr__(self):
    return
"("+str(self.x)+","+str(self.y)+")"
```

```
punto1 = Punto(0,0)
punto2 = Punto(1,1)
punto2_1 = Punto(1,1)
punto3 = Punto(3,4)


plano = Plano()
print(plano.agregar_punto(punto1))
print(plano)
print(plano.agregar_punto(punto2))
print(plano)
print(plano.agregar_punto(punto2_1))
print(plano)
print(plano.agregar_punto(punto3))
print(plano)
```

```
[(0,0)] Dist. Máx: 0
True
[(0,0), (1,1)] Dist. Máx: 1.4142135623730951
False
[(0,0), (1,1)] Dist. Máx: 1.4142135623730951
True
[(0,0), (1,1), (3,4)] Dist. Máx: 5.0
```

break

5 minutos

Ejercicio 4

- Crear tres clases:
  - Una clase Archivo, que tenga:
    - Atributos:
      - Contenido
      - Tamaño
  - Una clase Pendrive, que tenga:
    - Atributos:
      - Archivos (una lista de archivos).
      - Tamaño total
    - Métodos:
      - Agregar archivo
      - Eliminar archivo
      - Lleno
  - Una clase Computador, que tenga:
    - Atributos:
      - Archivos (una lista de archivos).
    - Métodos:
      - Transferir Archivos
      - Recibir Archivos
      - Mostrar Archivos

- Puede asumir que:
  - El contenido de los archivos solo son *string.* Cada uno tiene un tamaño de cantidad de caracteres de este string por 10.
  - Los computadores tienen un tamaño infinito para almacenar archivos.
  - Al comenzar hay un objeto `computador1` creado con N archivos.

  - Al agregar archivos al pendrive, se agregarán hasta que este se llene.

Implemente un código que mediante el método transferir_archivos pase archivos de un computador a otro ocupando el pendrive.

Ejercicio 4

```
class Archivo:
  def __init__(self,contenido):
    self.contenido = contenido
    self.tamaño = len(contenido)*10
```

Ejercicio 4

```python
class Pendrive:
  def __init__(self,tamaño_total):
    self.archivos = []
    self.tamaño_total = tamaño_total
    self.tamaño_ocupado = 0

  def agregar_archivo(self,archivo):
    if self.tamaño_ocupado + archivo.tamaño > self.tamaño_total:
      return False
    else:
      self.archivos.append(archivo)
      self.tamaño_ocupado += archivo.tamaño
      return True
```

```python
class Pendrive:
  …

  def eliminar_archivo(self):
    archivo = self.archivos.pop()
    self.tamaño_ocupado -= archivo.tamaño
    return archivo

  def lleno(self):
    if self.tamaño_ocupado >= self.tamaño_total:
      return True
    else:
      return False

  def __str__(self):
    str_aux = ""
    str_aux+="Tamaño total: "+str(self.tamaño_to-
tal)
    str_aux+=" Tamaño ocupado: "+str(self.ta-
maño_ocupado)
    str_aux+=" Cantidad archivos:
"+str(len(self.archivos))
    return str_aux
```

Ejercicio 4

```
class Computador:
  def __init__(self,id):
    self.archivos = []
    self.id = id

  def transferir_archivos(self,pendrive):
    while not pendrive.lleno() and len(self.archivos)>0 :
      archivo = self.archivos.pop()
      if pendrive.agregar_archivo(archivo):
        print("Transferido con éxito")
      else:
        print("¡Pendrive lleno!")
```

```
class Computador:
  …
  def recibir_archivos(self,pendrive):
    while len(pendrive.archivos) > 0:
      archivo = pendrive.eliminar_archivo()
      self.archivos.append(archivo)
      print("Archivo recibido con éxito")

  def imprimir_archivos(self):
    if len(self.archivos) > 0:
      counter = 0
      for i in self.archivos:
        print("Archivo:",counter,i.contenido)
        counter += 1
    else:
      print("¡Vacío!")
```

`__str(self)__`

- Recordemos la definición del método __str__
- Es posible “sobrecargar” el método (str) de un objeto para que retorna lo que nosotros queramos.
- En este caso, definimos el siguiente método de una clase:

```
def __str__(self):
    …
    return (string)
```

- Esto permitirá poder ocupar `print(objeto)` de esa clase y obtener lo que retorne este método. También permitirá obtener información cada vez que hagamos `str()` a este objeto.

¿Podemos sobrecargar otros métodos?

¡Sí!

Veremos un ejemplo con el operador suma.

Ejercicio 5

- Creemos una clase fracción, que nos permita sumar dos fracciones.

Ejercicio 5

- Creemos una clase fracción, que nos permita sumar dos fracciones.

```
class Fraccion:
  def __init__(self,numerador,denominador):
    self.numerador = numerador
    self.denominador = denominador

  def simplificar(self):
    if self.numerador > self.denominador:
      for i in range(2,numerador+1):
        if self.numerador%i==0 and self.denominador%i==0:
          self.numerador=self.numerador/i
          self.denominador=self.denominador/i
    elif self.numerador <= self.denominador:
      for i in range(2,self.denominador+1):
        if self.numerador%i==0 and self.denominador%i==0:
          self.numerador=self.numerador/i
          self.denominador=self.denominador/i

  def suma(self,fraccion):
    fraccion =
Fraccion(self.numerador*fraccion.denominador+self.denominador*fraccion.numerador,self.denominador*fraccion.denom
inador)
    fraccion.simplificar()
    return fraccion

  def __str__(self):
    return str(self.numerador)+"/"+str(self.denominador)
```

Ejercicio 5

- Creemos una clase fracción, que nos permita sumar dos fracciones.

```
def __add__(self,other):
  fraccion = Fraccion(self.numerador*other.denominador+self.denominador*other.numerador,self.denominador*other.denominador)
  fraccion.simplificar()
  return fraccion
```

Bibliografía

- http://runest.ing.puc.cl/class.html
- A. B. Downey. Think Python: How to think like a computer scientist. Green Tea Press, 2013 -> Capítulo 15

Links

- https://repl.it/@FelipeLopez/IIC1103POOII que contiene todos los ejemplos de la clase.

Resumen

- Las clase en son representaciones abstractas en Python.
- Una instancia de una clase con atributos específicos es un objeto.
- Se puede interactuar con una clase mediante sus métodos. Estos también permiten la interacción entre distintas clases.

- En Python se definen de la siguiente manera:

```
class Nombre_clase:
  def __init__(self,…):
        self.atributo1
        self.atributo2
                …
  def metodo1(self,…):

  def metodo2(self,…):
```

Resumen

- Es posible “sobrecargar” el método (str) de un objeto para que imprima lo que nosotros queramos.
- En este caso, definimos el siguiente método de una clase:

```
def __str__(self):
        …
        return (string)
```

- Esto permitirá poder ocupar `print(objeto)` de esa clase y obtener lo que retorne este método. También permitirá obtener información cada vez que hagamos `str()` a este objeto.
- No obstante, el método __str__, no funciona al imprimir objetos dentro de una lista.

- Para eso podemos ocupar el método __repr__

```
def __repr__(self):
        …
        return (string)
```

- Que cumple las mismas funciones de __str__, además de permitir imprimir objetos dentro de una lista.

Clase 15 – POO II

IIC1103 Sección 9 – 2019-2

Profesor: Felipe López

21-11-2019