Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Clase 17: Programación Orientada a Objetos

Rodrigo Toro Icarte (rntoro@uc.cl)

IIC1103 Introducción a la Programación - Sección 5

18 de Mayo, 2015

Clases pasadas

Hemos visto distintos tipos de datos en Python:

`int`, `float`, `complex`, `bool`, `string`, `list` y `tuple`.

... de hecho podemos saber la *clase* de un objeto con el comando `type()`.

Clases pasadas

Hemos visto distintos tipos de datos en Python:

`int`, `float`, `complex`, `bool`, `string`, `list` y `tuple`.

... de hecho podemos saber la *clase* de un objeto con el comando `type()`.

```
1 a = 4.5; b = "4.5"
2 c = [1,2,3,4]
3 d = (1,2,3,4)
4
5 print(type(a)) # <class 'float'>
6 print(type(b)) # <class 'str'>
7 print(type(c)) # <class 'list'>
8 print(type(d)) # <class 'tuple'>
```

Programación Orientada a Objetos

En computación aspiramos a modelar el mundo para solucionar problemas reales.

Programación Orientada a Objetos

En computación aspiramos a modelar el mundo para solucionar problemas reales.

Programación Orientada a Objetos

... o para matar el tiempo.

Programación Orientada a Objetos

... o para matar el tiempo.

Programación Orientada a Objetos

Queremos modelar el mundo...

Programación Orientada a Objetos

Queremos modelar el mundo...

Miren a su alrededor... ¿qué ven?

Programación Orientada a Objetos

"Hay **Objetos** que pertenecen a cierta **Clase**".

Programación Orientada a Objetos

"Hay **Objetos** que pertenecen a cierta **Clase**".

Ejemplos:

- Puerta
- Ventana
- Plumón
- Persona

Programación Orientada a Objetos

"Hay **Objetos** que pertenecen a cierta **Clase**".

Ejemplos:

- Puerta
- Ventana
- Plumón
- Persona

Trabajemos un poco con la clase **persona**.

Programación Orientada a Objetos

Ejercicio: Diseñe un *mini-siding*, donde se pueda agregar notas a cada estudiante y calcular promedios.

Programación Orientada a Objetos

Ejercicio: Diseñe un *mini-siding*, donde se pueda agregar notas a cada estudiante y calcular promedios.

Solución: Tenemos que manejar una lista de estudiantes. ¿Cómo los representamos?

Programación Orientada a Objetos

Ejercicio: Diseñe un *mini-siding*, donde se pueda agregar notas a cada estudiante y calcular promedios.

Solución: Tenemos que manejar una lista de estudiantes. ¿Cómo los representamos?

Opción: Con tuplas (nombre, apellido, n° de alumno, notas)

```
1 estudiantes = []
2 estudiantes.append(('Juan','Águila','14000000',[]))
3 estudiantes.append(('Aldo','Verri','14000001',[]))
4 estudiantes.append(('María','Pinto','14000002',[]))
```

Programación Orientada a Objetos

```
estudiantes = []
estudiantes.append(('Juan','Águila','14000000',[]))
estudiantes.append(('Aldo','Verri','14000001',[]))
estudiantes.append(('María','Pinto','14000002',[]))

# Agrego notas
estudiantes[0][3].append(6.5)
estudiantes[0][3].append(7.0)
estudiantes[0][3].append(6.7)

estudiantes[1][3].append(3.0)
estudiantes[1][3].append(2.7)
estudiantes[1][3].append(3.8)

estudiantes[2][3].append(5.7)
estudiantes[2][3].append(7.0)
estudiantes[2][3].append(6.2)
```

Programación Orientada a Objetos

```
19 # Muestro promedios
20 for e in estudiantes:
21   promedio = sum(e[3])/len(e[3])
22   print(e[1],"\t=>",'%0.2f'%promedio)
23
24 # Salida:
25 #    >>> Águila   => 6.73
26 #    >>> Verri    => 3.17
27 #    >>> Pinto    => 6.30
```

Programación Orientada a Objetos

¿No les parecen incómodas las tuplas?

Programación Orientada a Objetos

¿No les parecen incómodas las tuplas?

- Debemos recordar el significado de cada id.
- Con muchos atributos equivocarse con los id es fácil.
- No podemos cambiar los valores de los atributos.
- Se pueden agregar tuplas en orden incorrecto.

Programación Orientada a Objetos

¿No les parecen incómodas las tuplas?

- Debemos recordar el significado de cada id.
- Con muchos atributos equivocarse con los id es fácil.
- No podemos cambiar los valores de los atributos.
- Se pueden agregar tuplas en orden incorrecto.

... Sería útil si Python nos permitiera crear nuestros propios tipos de datos.

Programación Orientada a Objetos

Python nos permite definir nuestras propias **clases**.

Sintaxis

```
class nombre_clase:
    bloque_codigo_clase
```

Programación Orientada a Objetos

Python nos permite definir nuestras propias **clases**.

Sintaxis

class nombre_clase**:**
    bloque_codigo_clase

Ahora podemos crear un objeto del tipo `nombre_clase`.

Sintaxis

variable = **nombre_clase()**

Programación Orientada a Objetos

Ejemplo:

```
1 # Defino el tipo de dato (clase) "persona"
2 class persona:
3   pass
4
5 # Creo un objeto de la clase persona
6 p = persona()
7 print(type(p))
8 # >>> <class '__main__.persona'>
```

Programación Orientada a Objetos

Ejemplo:

```
1 # Defino el tipo de dato (clase) "persona"
2 class persona:
3   pass
4
5 # Creo un objeto de la clase persona
6 p = persona()
7 print(type(p))
8 # >>> <class '__main__.persona'>
```

... por ahora `persona` no hace nada.

Atributos

A cada **objeto** le podemos agregar **atributos** (variables) y ver/modificar sus valores.

Atributos

A cada **objeto** le podemos agregar **atributos** (variables) y ver/modificar sus valores.

Sintaxis

var = **nombre_clase()**
var.nombre_atributo = valor
print(**var.nombre_atributo**)

Atributos

Ejemplo: En nuestro problema cada persona tiene un nombre, apellido, número de alumno y lista de notas.

Atributos

Ejemplo: En nuestro problema cada persona tiene un nombre, apellido, número de alumno y lista de notas.

Primero defino la clase `persona`.

```
1 # Defino el tipo de dato (clase) "persona"
2 class persona:
3   pass
```

Atributos

... luego creo *personas* y agrego sus atributos.

```
5  # Creo una persona y le damos valor a sus atributos
6  juan = persona()
7  juan.nombre = 'Juan'
8  juan.apellido = 'Águila'
9  juan.n_alumno = '14000000'
10
11 # Creo otra persona
12 aldo = persona()
13 aldo.nombre = 'Aldo'
14 aldo.apellido = 'Verri'
15 aldo.n_alumno = '14000001'
16
17 # Creo último estudiante
18 maria = persona()
19 maria.nombre = 'María'
20 maria.apellido = 'Pinto'
21 maria.n_alumno = '14000002'
```

Atributos

Finalmente, podríamos agregarlos a una lista y mostrar sus nombres.

Atributos

Finalmente, podríamos agregarlos a una lista y mostrar sus nombres.

```
23 # Agrego estudiantes a mi lista
24 estudiantes = []
25 estudiantes.append(juan)
26 estudiantes.append(aldo)
27 estudiantes.append(maria)
28
29 # Muestro los nombres
30 for e in estudiantes:
31   print(e.nombre)
```

Atributos

Finalmente, podríamos agregarlos a una lista y mostrar sus nombres.

```
23 # Agrego estudiantes a mi lista
24 estudiantes = []
25 estudiantes.append(juan)
26 estudiantes.append(aldo)
27 estudiantes.append(maria)
28
29 # Muestro los nombres
30 for e in estudiantes:
31   print(e.nombre)
```

¿Qué cosas pueden fallar con este enfoque?

Constructores

Constructor: Es un método que se llama cuando se crea un nuevo objeto de la clase.

Sintaxis

```
class nombre_clase:
    def __init__(self, par_1, par_2, ...):
        Bloque_código_constructor
```

Constructores

Constructor: Es un método que se llama cuando se crea un nuevo objeto de la clase.

Sintaxis

class nombre_clase**:**
    def __**init**__**(self**, par_1, par_2, ...**):**
        Bloque_código_constructor

Al hacer `var = nombre_clase(val_1,val_2,...)` se ejecuta `__init__()`.

Constructores

Ejemplo:

```
1 # Defino el tipo de dato (clase) "persona"
2 class persona:
3   # Constructor
4   def __init__(self, nombre):
5     print("Persona creada:",nombre)
6
7 # Creo una persona
8 j = persona("juan")
```

Constructores

Ejemplo:

```
1 # Defino el tipo de dato (clase) "persona"
2 class persona:
3   # Constructor
4   def __init__(self, nombre):
5     print("Persona creada:",nombre)
6
7 # Creo una persona
8 j = persona("juan")
```

Preguntas:

- ¿Qué muestra el código anterior?
- ¿Qué es `self`?

Constructores

self: Es una variable especial que contiene el objeto recién creado.

Constructores

self: Es una variable especial que contiene el objeto recién creado.

```
1  # Defino el tipo de dato (clase) "persona"
2  class persona:
3    # Constructor
4    def __init__(self, nombre):
5      print("Persona creada:",nombre)
6
7  # Creo una persona
8  j = persona("juan")
9  j.nombre = 'Juan'
10 j.apellido = 'Águila'
11 j.n_alumno = '14000000'
```

... en este caso, `self` equivale a `j`.

Constructores

¿De qué nos podría servir tener `self` (i.e. `j`) en el constructor?

```
1  # Defino el tipo de dato (clase) "persona"
2  class persona:
3    # Constructor
4    def __init__(self, nombre):
5      print("Persona creada:",nombre)
6
7  # Creo una persona
8  j = persona("juan")
9  j.nombre = 'Juan'
10 j.apellido = 'Águila'
11 j.n_alumno = '14000000'
```

Constructores

¿De qué nos podría servir tener `self` (i.e. `j`) en el constructor?

```
1  # Defino el tipo de dato (clase) "persona"
2  class persona:
3    # Constructor
4    def __init__(self, nombre):
5      print("Persona creada:",nombre)
6
7  # Creo una persona
8  j = persona("juan")
9  j.nombre = 'Juan'
10 j.apellido = 'Águila'
11 j.n_alumno = '14000000'
```

... podemos agregar los atributos en el constructor

Constructores

Con `self.var = valor` agregamos un atributo al objeto.

```
2 class persona:
3   # Constructor
4   def __init__(self, nombre, apellido, n_alumno):
5     # Agrego atributos a persona
6     self.nombre = nombre
7     self.apellido = apellido
8     self.n_alumno = n_alumno
9     self.notas = []
```

Constructores

Con `self.var = valor` agregamos un atributo al objeto.

```
2  class persona:
3    # Constructor
4    def __init__(self, nombre, apellido, n_alumno):
5      # Agrego atributos a persona
6      self.nombre = nombre
7      self.apellido = apellido
8      self.n_alumno = n_alumno
9      self.notas = []
```

Creo objetos de la clase persona mediante su constructor:

```
12 juan = persona('Juan', 'Águila', '14000000')
13 aldo = persona('Aldo', 'Verri', '14000001')
14 maria = persona('María', 'Pinto', '14000002')
```

Constructores

```
16 # Agrego notas
17 juan.notas.extend([6.5, 7.0, 6.7])
18 aldo.notas.extend([3.0, 2.7, 3.8])
19 maria.notas.extend([5.7, 7.0, 6.2])
20
21 # Formo lista con los estudiantes
22 estudiantes = [juan, aldo, maria]
23
24 # Muestro promedios
25 for e in estudiantes:
26   promedio = sum(e.notas)/len(e.notas)
27   print(e.apellido,"\t=>",'%0.2f'%promedio)
28
29 # Salida:
30 #    >>> Águila   => 6.73
31 #    >>> Verri    => 3.17
32 #    >>> Pinto    => 6.30
```

Constructores

Ventaja constructores:

- Lógica para asignar atributos dentro de la definición de la clase.
- Obligamos asignación de atributos.
- Podemos ejecutar código cada vez que se crea el objeto.

Constructores

Constructores conocidos

Constructores

Constructores conocidos

```
1  # input cualquiera
2  n = input("Ingrese input: ")
3
4  # constructores que hemos utilizado
5  i = int(n)
6  f = float(n)
7  c = complex(n)
8  b = bool(n)
9  s = str(n)
10 l = list()
11 t = tuple()
```

Métodos

Idea: Además de *atributos*, los objetos tienen *comportamiento*.

comportamiento == acción == método == código

Métodos

Ejemplo 1: ¿Qué objetos importantes hay aquí?

Métodos

¿Qué atributos y acciones nos interesan de ellos?

(a) Usuario.

(b) Guardia.

(c) Metro.

Métodos

usuario
- volumen - posición - destino
+ intentar_entrar() + intentar_salir()

Métodos

guardia
- volumen - posición
+ señal_cerrar_puertas()

Métodos

metro
- volumen_interior - personas_interior - posición
+ avanzar() + detenerse() + abrir_puertas() + cerrar_puertas()

Métodos

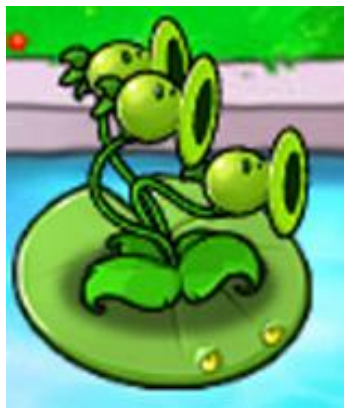
Ejemplo 2: ¿Qué objetos importantes hay aquí?

Métodos

¿Qué atributos y acciones nos interesan de ellos?

(a) Girasol.

(b) Zombie.



(c) Planta.

Métodos

girasol
- vida - frecuencia_sol - posición - dibujo
+ lanzar_sol()

Métodos

zombie
- vida - daño - velocidad - posición - dibujo
+ comer() + avanzar()

Métodos

planta
- vida - daño - frecuencia_ataque - posición - dibujo
+ atacar()

Métodos

Método: Función asociada a una clase particular.

Sintaxis

```
class nombre_clase:
    def __init__(self, par_1, par_2, ...):
        Bloque_código_constructor
    def nombre_método(self, par_1, par_2, ...):
        Bloque_código_método
```

Métodos

Método: Función asociada a una clase particular.

Sintaxis

```
class nombre_clase:
    def __init__(self, par_1, par_2, ...):
        Bloque_código_constructor
    def nombre_método(self, par_1, par_2, ...):
        Bloque_código_método
```

Para llamar al método:

Sintaxis

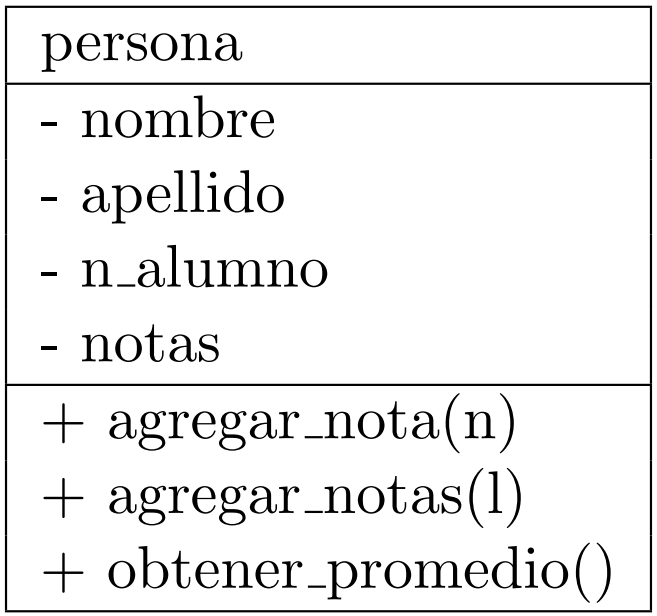
```
var = nombre_clase()
var.nombre_método(val_1, val_2, ...)
```

Métodos

En *mini-siding* ¿Qué métodos podríamos agregar a persona?

Métodos

En *mini-siding* ¿Qué métodos podríamos agregar a persona?



Métodos

En *mini-siding* ¿Qué métodos podríamos agregar a persona?

persona
- nombre - apellido - n_alumno - notas
+ agregar_nota(n) + agregar_notas(l) + obtener_promedio()

¿Cómo programarían `obtener_promedio()`?

Métodos

```
1  # Defino el tipo de dato (clase) "persona"
2  class persona:
3    # Constructor
4    def __init__(self, nombre, apellido, n_alumno):
5      # Atributos de persona
6      self.nombre = nombre
7      self.apellido = apellido
8      self.n_alumno = n_alumno
9      self.notas = []
10   # Métodos
11   def agregar_nota(self,n):
12     self.notas.append(n)
13   def agregar_notas(self,l):
14     self.notas.extend(l)
15   def get_promedio(self):
16     return sum(self.notas)/len(self.notas)
```

Métodos

```
20 # Creo las personas y doy valores a sus atributos
21 juan = persona('Juan', 'Águila', '14000000')
22 aldo = persona('Aldo', 'Verri', '14000001')
23 maria = persona('María', 'Pinto', '14000002')
24
25 # Agrego notas
26 juan.agregar_notas([6.5, 7.0, 6.7])
27 aldo.agregar_notas([3.0, 2.7, 3.8])
28 maria.agregar_notas([5.7, 7.0, 6.2])
29
30 # Formo lista y muestro promedios
31 estudiantes = [juan, aldo, maria]
32 for e in estudiantes:
33   print(e.apellido,"\t=>",'%0.2f'%e.get_promedio())
34
35 # Salida:
36 #    >>> Águila   => 6.73
37 #    >>> Verri    => 3.17
38 #    >>> Pinto    => 6.30
```

Métodos

Self es una variable especial que contiene al objeto sobre el que se ejecuta la función.

Métodos

Self es una variable especial que contiene al objeto sobre el que se ejecuta la función.

Definición método `agregar_notas(l)`.

```
13   def agregar_notas(self,l):
14     self.notas.extend(l)
```

Llamados a `agregar_notas(l)` son sin el `self`.

```
26 juan.agregar_notas([6.5, 7.0, 6.7])
27 aldo.agregar_notas([3.0, 2.7, 3.8])
28 maria.agregar_notas([5.7, 7.0, 6.2])
```

Métodos

Self es una variable especial que contiene al objeto sobre el que se ejecuta la función.

Definición método `agregar_notas(l)`.

```
13   def agregar_notas(self,l):
14     self.notas.extend(l)
```

Llamados a `agregar_notas(l)` son sin el `self`.

```
26 juan.agregar_notas([6.5, 7.0, 6.7])
27 aldo.agregar_notas([3.0, 2.7, 3.8])
28 maria.agregar_notas([5.7, 7.0, 6.2])
```

Obs: Necesitamos `self` para acceder a atributos y métodos.

Métodos

Ventajas métodos:

- Asociamos funcionalidades propias de una clase, a su definición.
- Ganamos semántica.
- Código reutilizable.

Métodos

Métodos conocidos

Métodos

Métodos conocidos

Algunos métodos sobre strings...

```
2 s = "hola"
3 s.replace('h','')
4 s.lower()
5 s.count('a')
```

Algunos métodos sobre listas...

```
8 l = [1,2,3,4,5]
9 l.append(6)
10 l.extend([7,8,9])
11 l.sort()
12 l.reverse()
```

Clases vs Objetos

¿Cuál es la diferencia entre la clase y el objeto?

Clases vs Objetos

¿Cuál es la diferencia entre la clase y el objeto?

Clase

La clase es el template del objeto ($\approx$ su molde).

Clases vs Objetos

¿Cuál es la diferencia entre la clase y el objeto?

Clase

La clase es el template del objeto ($\approx$ su molde).

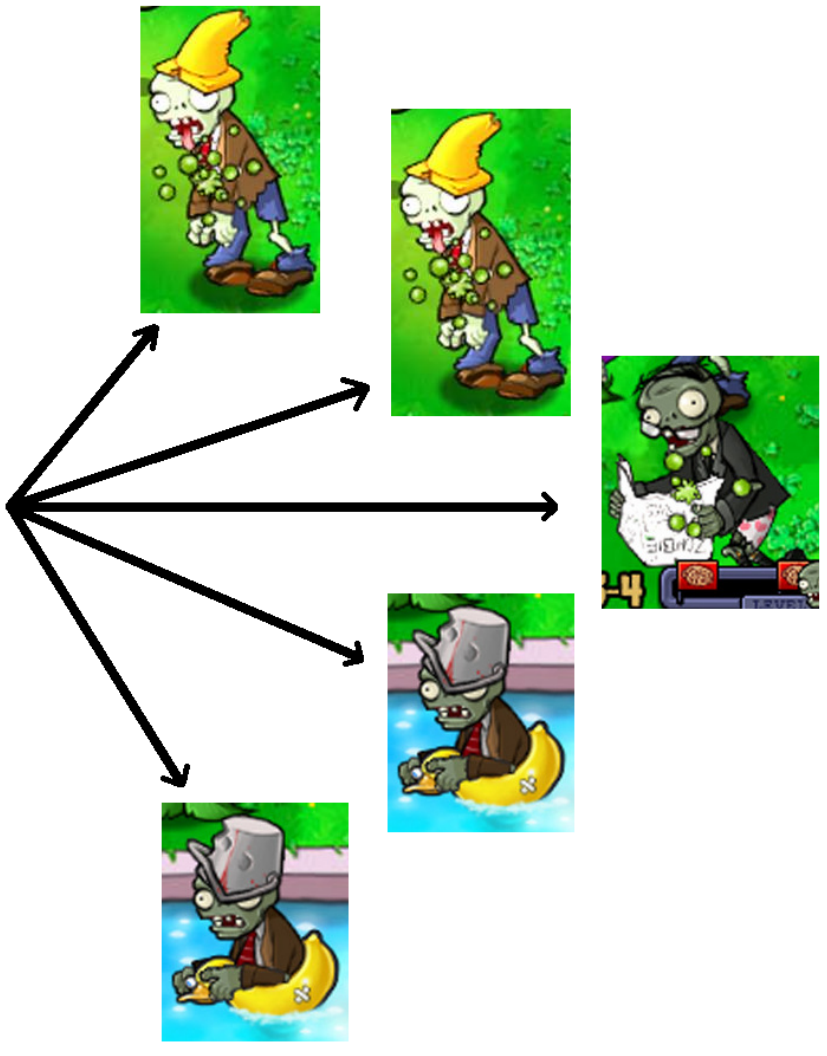
A partir del *molde* podemos *construir* objetos.

Objeto

Un objeto es una instancia particular de una clase.

Clases vs Objetos

zombie
- vida - daño - velocidad - dibujo
+ atacar() + avanzar()



Clases vs Objetos

Clase:

```
2  class persona:
3    # Constructor
4    def __init__(self, nombre, apellido, n_alumno):
5      # Agrego atributos a persona
6      self.nombre = nombre
7      self.apellido = apellido
8      self.n_alumno = n_alumno
9      self.notas = []
```

Objetos (instancias) de la clase:

```
12 juan = persona('Juan', 'Águila', '14000000')
13 aldo = persona('Aldo', 'Verri', '14000001')
14 maria = persona('María', 'Pinto', '14000002')
```

Ejemplo

Programe un simulador de batallas entre:

Ejemplo

Programe un simulador de batallas entre:

(a) Súperman.

Ejemplo

Programe un simulador de batallas entre:

(a) Súperman.

(b) Gokú.

Ejemplo

Programe un simulador de batallas entre:

(a) Súperman.

(b) Gokú.

(c) Chuck Norris.

Ejemplo

Usaremos una sola clase:

guerrero
- nombre - vida - fuerza - precisión - velocidad - defensa
+ golpear(g)

Ejemplo

Consideremos que están luchando j_1 contra j_2:

- Se golpea por turnos.
- Comienza el jugador con mayor velocidad.
- Si j_1 intenta golpear a j_2, la probabilidad de acierto es:

$$\frac{j_1.precision - j_2.velocidad}{100}$$

- Si j_1 golpea a j_2, el daño será:

$$\max(j_1.fuerza - j_2.defensa + randrange(-10, 11), 1)$$

- Pelea finaliza cuando algún guerrero llega a vida 0.

Ejemplo

```
1  import random
2
3  class guerrero:
4    def __init__(self,nombre,vida,fuerza,precision,
       velocidad,defensa):
5      self.nombre = nombre; self.vida = vida
6      self.fuerza = fuerza; self.precision = precision
7      self.velocidad = velocidad; self.defensa = defensa
8
9    def golpear(self,g):
10     # veo si acierto el golpe
11     if(random.random() <= (self.precision - g.
       velocidad) / 100):
12       # en caso de acertar, agrego daño al oponente
13       g.vida -= max([(self.fuerza - g.defensa)
14                      + random.randrange(-10,11),1])
15       print("Golpe certero de", self.nombre)
16     else:
17       print(g.nombre, "esquiva el golpe!")
```

Ejemplo

Función que simula la batalla:

```
20 def simular_batalla(j1,j2):
21   # comienza jugador más veloz
22   golpeador,receptor = j1, j2
23   if(j1.velocidad < j2.velocidad):
24     golpeador,receptor = j2,j1
25   # se golpean hasta que alguno tenga vida cero
26   while(j1.vida > 0 and j2.vida > 0):
27     print("\n" + j1.nombre,j1.vida,"vs",
28           j2.vida,j2.nombre)
29     golpeador.golpear(receptor)
30     # cambio de turnos
31     golpeador,receptor = receptor,golpeador
32   # fin
33   print("\n" + j1.nombre,j1.vida,"vs",
34         j2.vida,j2.nombre)
35   print("Ganador:",receptor.nombre)
```

Ejemplo

Creamos objetos y simulamos una batalla:

```
37 # batalla de ejemplo
38 superman = guerrero('Superman',100,50,80,30,20)
39 goku = guerrero('Gokú',100,60,80,40,20)
40 chuck = guerrero('Chuck Norris',200,99,99,99,99)
41
42 # simula batalla
43 simular_batalla(goku,chuck)
```

Ejercicios

1) Agregue a la clase `persona` del *mini-siding* un método que borre las notas de un estudiante.

2) Agregue un nuevo guerrero al simulador de batallas con los siguientes atributos:

- **Nombre**: Aldo Verri
- **Vida**: 10
- **Fuerza**: 1
- **Precisión**: 1
- **Velocidad**: 1
- **Resistencia**: 1

... luego haga que pelee contra Chuck Norris.

Ejercicios

3) Cree una clase ampolleta con un método para cambiar su estado (si estaba apagada pasa a estar prendida y viceversa). Luego cree 4 ampolletas y que el usuario sea capaz de prenderlas y apagarlas.

4) Cree un *mini-plants vs zombies*. Para ello debe programar una clase planta y una clase zombie. Ambos tienen vida y se encuentran a `n` metros de distancia. En cada turno, la planta ataca al zombie (desde lejos) y el zombie se acerca a la planta (un metro). Cuando el zombie se encuentre a 1 metro de la planta, puede atacarla (2 mordiscos por turno). El daño de cada ataque de la planta es `p.ataque + random.randrange(-10,11)`, y el daño del ataque del zombie es `z.ataque + 2*random.randrange(-10,11)`. El juego acaba cuando el zombie se come a la planta o la planta mata al zombie.

Ejercicios

5) Cree un tablero de $n \times n$ donde existan 5 objetos: un lápiz, un cuaderno, un computador, un pase escolar y una mochila. También hay una persona en alguna posición del tablero. En cada turno, la persona se puede mover una casilla en cualquier dirección (izquierda, derecha, arriba, abajo). Para ello utilice las teclas `asdw`. El juego consiste en obtener cada objeto y dejarlo en la mochila. La persona sólo puede cargar un objeto a la vez y no puede mover de lugar la mochila. El juego termina cuando todos los objetos están en la mochila. La idea es hacer esto en el mínimo número de pasos. Para mostrar los distintos elementos del tablero en consola, use letras. Ejemplo: La persona es una **O**, la mochila una **M**, etc...