Programación orientada a objetos

...

Algoritmos y Estructuras de Datos
2023

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

¿Cómo trabajaría con tiempos?

¿Cómo trabajaría con tiempos?

- ¿Qué características tiene un tiempo o una hora determinada?
- ¿Qué elementos se necesitarían para su implementación?
- Hay operaciones que sean distintas o que tengan restricciones distintas a las de, por ejemplo, un número entero?

Sobre Objetos, Clases y Métodos

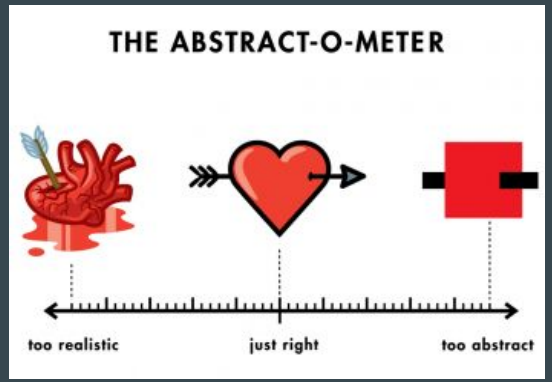
- Una **clase** es una plantilla que define las características de un tipo de datos:
  - Variables, estructuras de datos, otras clases
  - Funciones asociadas
- Un **objeto** es una instancia de una clase.
- Los **métodos** son todas las funciones definidas para esa clase

```
>>> a = "Algoritmos"
>>> type(a)
<class 'str'>
```

La variable ***a*** es una instancia de la clase ***string***

Principios

- **Modularidad:** un sistema se compone de muchos componentes independientes que interactúan entre sí.
- **Encapsulación:** los módulos deberían poder modificarse sin afectar otros componentes.
  - Los detalles internos de un componente no deberían ser accesibles por otros componentes, objetos o funciones.
- **Abstracción:** intentar limitar un problema a sus partes más fundamentales.



computersciencewiki.org/index.php/Abstraction

Objetivos del uso de objetos

- **Robustez:** capaz de manejar entradas inesperadas.
- **Adaptabilidad:** capacidad de correr en distintas plataformas y permitir la implementación de nuevas características acordes a los cambios en el ambiente.
- **Reusabilidad:** un mismo bloque de código debería poder usarse en diferentes aplicaciones

Creando clases

- Para definir una nueva clase debemos definirla usando la palabra reservada ***class***.
- Dentro de una clase primero se define el ***constructor*** con la función ***__init__***.
- ***self*** hace referencia a la instancia de la clase que invoca algún método.

```
1 class Tiempo:
2     def __init__ (self, h, m, s):
3         self._hora = h
4         self._minuto = m
5         self._segundo = s
6
7
8 t1 = Tiempo(17, 30, 40)
9 t2 = Tiempo(23, 59, 59)
```

Funciones

- Las funciones se definen con ***def*** dentro de la clase.
- Para acceder a los valores del objeto particular tenemos que usar la variable ***self***

```
1 class Tiempo:
2     def __init__ (self, h, m, s):
3         self._hora = h
4         self._minuto = m
5         self._segundo = s
6 
7     def devolver_minutos(self):
8         return self._minuto
9 
10 t1 = Tiempo(17, 30, 40)
11 print(t1.devolver_minutos())
```

```
>>> t1 = Tiempo(17, 30, 40)
>>> print(t1.devolver_minutos())
30
```

Funciones

- Las funciones se definen con ***def*** dentro de la clase.
- Para acceder a los valores del objeto particular tenemos que usar la variable ***self***

```
1 class Tiempo:
2     def __init__ (self, h, m, s):
3         self._hora = h
4         self._minuto = m
5         self._segundo = s
6
7     def devolver_minutos(self):
8         return self._minuto
9
10 t1 = Tiempo(17, 30, 40)
11 print(t1.devolver_minutos())
```

```
>>> i = 7
>>> type(i)
<class 'int'>
>>> i.devolver_minutos()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'devolver_minutos'
```

- Solo los objetos de una clase pueden acceder a las funciones que se declaren dentro.

Sintaxis (o “lo que dice el manual de buenas prácticas”)

- Un guión bajo al principio:
  - Partes ocultas => Encapsulación
  - Convención
- Dos guiones bajos al principio y al final:
  - Usos especiales en el lenguaje.
  - Define el comportamiento en funciones y operadores para una clase particular
  - Algunos ejemplos:
    - __init__ para modificar “=”
    - __len__ para modificar “len”

```
1 class Tiempo:
2     def __init__(self, h, m, s):
3         self._hora = h
4         self._minuto = m
5         self._segundo = s
6
7 t1 = Tiempo(17, 30, 40)
```

Polimorfismos en funciones binarias

- Algunos operadores tienen comportamientos distintos para distintas clases.
- Usar una misma entidad para representar diferentes operaciones en distintos escenarios se conoce como ***polimorfismo***.
- Se definen como una función más, pero con nombres especiales.

Operator	Magic Method
+	__add__(self, other)
–	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
//	__floordiv__(self, other)
%	__mod__(self, other)

Polimorfismos en funciones binarias

- Algunos operadores tienen comportamientos distintos para distintas clases.
- Usar una misma entidad para representar diferentes operaciones en distintos escenarios se conoce como ***polimorfismo***.
- Se definen como una función más, pero con nombres especiales.

```
1 class Tiempo:
2     # __init__ y demases
3
4     def __sub__(self, other):
5         h = self._hora - other._hora
6         m = self._minutos - other._minutos
7         s = self._segundos - other._segundos
8         return Tiempo(h, m, s)
6
7 t1 = Tiempo(17, 30, 40)
8 t2 = Tiempo(20, 32, 59)
9 t3 = t2 - t1
```

Errores (o “lo que dice el manual de buenas prácticas”)

- Para incrementar la robustez de un código es necesario asegurar que, al crear un objeto, se utilicen los datos correctos.
- La función ***raise*** permite informar al usuario qué error cometió

Class	Description
Exception	A base class for most error types
AttributeError	Raised by syntax obj.foo, if obj has no member named foo
EOFError	Raised if “end of file” reached for console or file input
IOError	Raised upon failure of I/O operation (e.g., opening file)
IndexError	Raised if index to sequence is out of bounds
KeyError	Raised if nonexistent key requested for set or dictionary
KeyboardInterrupt	Raised if user types ctrl-C while program is executing
NameError	Raised if nonexistent identifier used
StopIteration	Raised by next(iterator) if no element; see Section 1.8
TypeError	Raised when wrong type of parameter is sent to a function
ValueError	Raised when parameter has invalid value (e.g., sqrt(−5))
ZeroDivisionError	Raised when any division operator used with 0 as divisor

Errores (o “lo que dice el manual de buenas prácticas”)

- Para incrementar la robustez de un código es necesario asegurar que, al crear un objeto, se utilicen los datos correctos.
- La función ***raise*** permite informar al usuario qué error cometió

```
1 class Tiempo:
2     def __init__ (self, h, m, s):
3         if not isinstance (h, int):
4             raise TypeError ("Hora debe ser
int")
5
6         if h > 24 or h < 0:
7             raise ValueError ("Hora debe estar
entre 0 y 23")
8         self._hora = h
```

Herencia

- Permite definir nuevas clases a partir de clases ya definidas.
- Una subclase puede especializarse en un comportamiento existente, con métodos y variables nuevas, o expandir nuevos comportamientos.
- El mecanismo para definir una subclase se basa en el constructor ***super***.

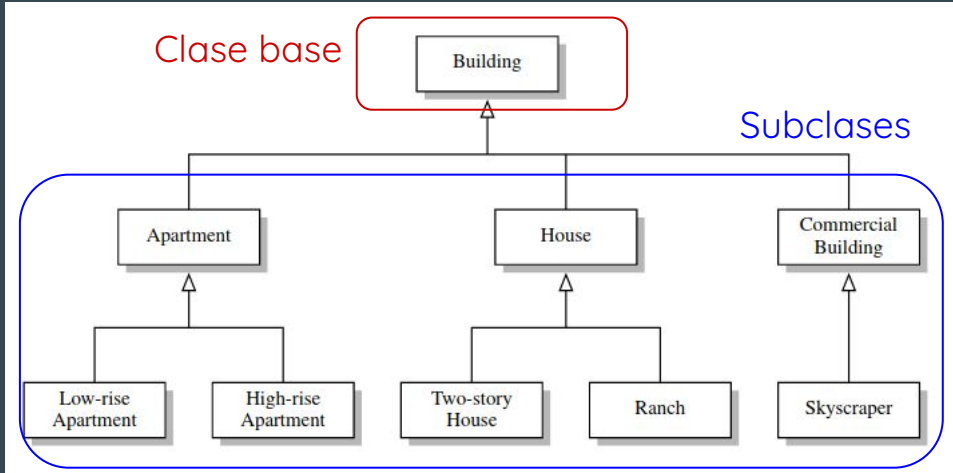
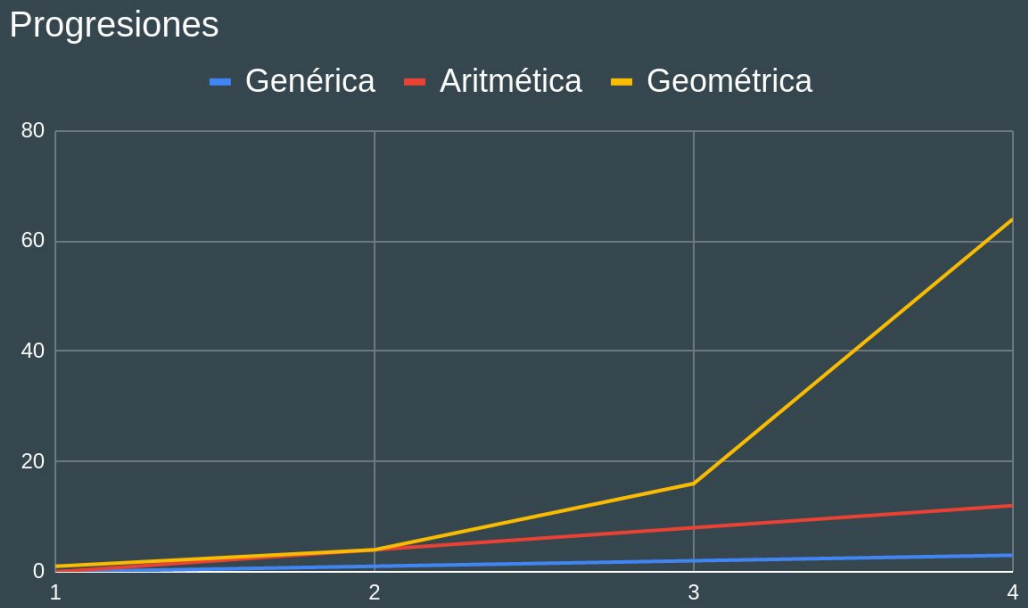


Figure 2.4: An example of an "is a" hierarchy involving architectural buildings.

Ejemplo: series numéricas

- Una seria genérica comienza en 0 y avanza sumando 1
  - 0, 1, 2, 3, 4, ...
- Una serie aritmética puede avanzar con un paso (por ejemplo, 4):
  - 0, (0+4), (4+4), (8+4), (12+4), ... => 0, 4, 8, 12, 16, ...
- Una serie geométrica comienza en 1, posee una base y avanza por multiplicaciones (por ejemplo, para base = 4)
  - 1, (1*4), (4*4), (16*4), (64*4), ... => 1, 4, 16, 64, 256,...
- Todas las series tienen un comienzo y funciones similares:
  - Calcular el siguiente elemento
  - Mostrar/Devolver una cierta cantidad de elementos

Ejemplo: series numéricas



Ejemplo: series numéricas

```
1  class Progression:
2      def __init__ (self, start=0):
3          self._current = start
4
5      def next (self):
6          answer = self._current
7          self._advance()
8          return answer
9
10     def _advance(self):
11         self._current += 1
12
13     def get_progression( self, n):
14         answer = []
15         for i in range(n):
16             answer.append( self.next())
17         return answer
```

```
19 class ArithmeticProgression(Progression):
20     def __init__(self, increment=1, start=0):
21         super( ).__init__(start)
22         self._increment = increment
23
24     def _advance(self):
25         self._current += self._increment
```

```
27 class GeometricProgression(Progression):
28     def __init__ (self, base=1, start=1):
29         super().__init__(start)
30         self._base = base
31
32     def _advance(self):
33         self._current *= self._base
```

Ejemplo: series numéricas

```
35  serie_generica = Progression()
36  serie_aritmetica = ArithmeticProgression(4)
37  serie_geometrica = GeometricProgression(4)
38
39  print (serie_generica.get_progression(10))
40  print (serie_aritmetica.get_progression(10))
41  print (serie_geometrica.get_progression(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 4, 8, 12, 16, 20, 24, 28, 32, 36]
[1, 4, 16, 64, 256, 1024, 4096, 16384, 65536, 262144]
```

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 1