Introducción a la programación en Python

Pedro Corcuera

Dpto. Matemática Aplicada y Ciencias de la Computación

Universidad de Cantabria

corcuerp@unican.es

Objetivos

- Programación orientada a objetos
- Estructuras de datos

Índice

- Programación orientada a objetos
- Búsqueda y ordenación
- Estructuras de datos

Limitaciones de la programación estructurada

- La descomposición de programas en funciones no es suficiente para conseguir la reusabilidad de código.
- Es difícil entender, mantener y actualizar un programa que consiste de una gran colección de funciones.
- Con la programación orientada a objeto se supera las limitaciones de la programación estructurada mediante un estilo de programación en el que las tareas se resuelven mediante la colaboración de objetos.

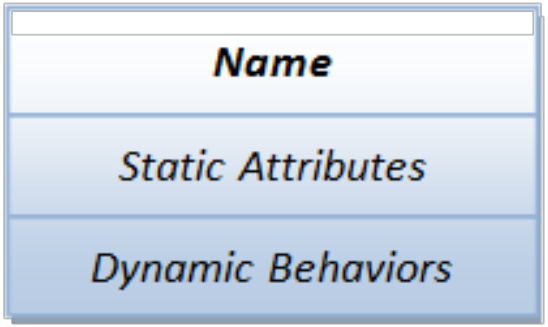
Programación orientada a objetos (OOP) en Python

- Una clase es una plantilla de entidades del mismo tipo. Una instancia es una realización particular de una clase. Python soporta instancias de clases y objetos.
- Un objeto contiene atributos: atributos de datos (o variables) y comportamiento (llamados métodos). Para acceder un atributo se usa el operador punto, ejemplo: nombre_instancia.nombre_atributo
- Para crear una instancia de una clase se invoca el constructor: nombre_instancia = nombre_clase(args)

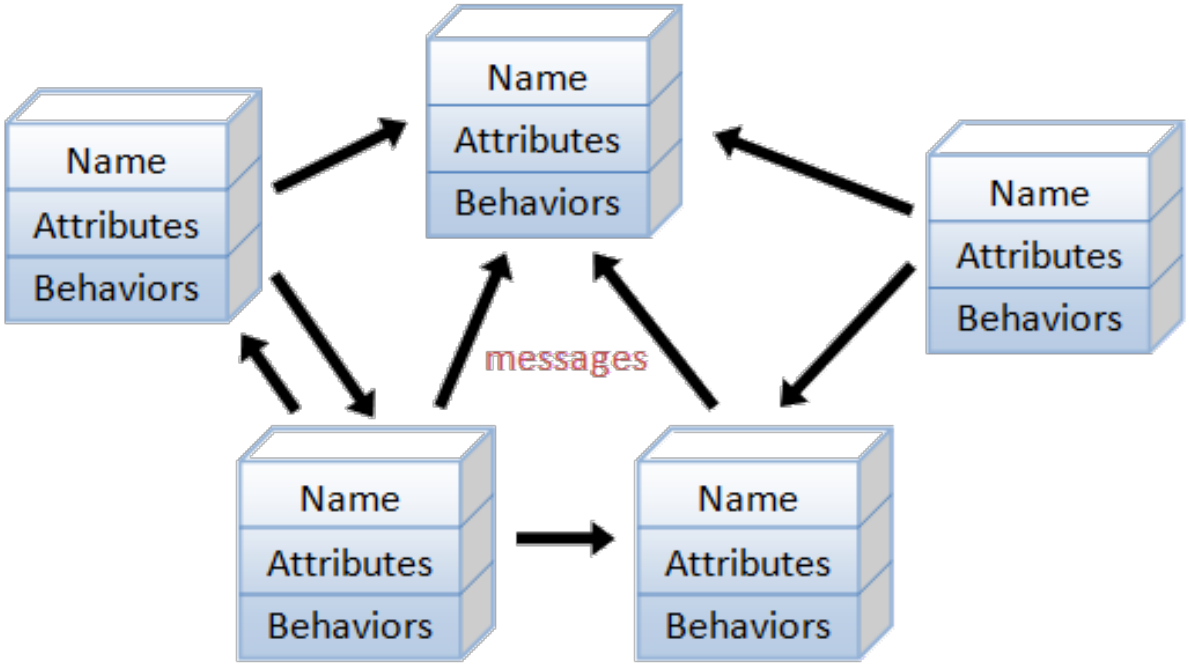
Objetos de clase vs Objetos de instancia

- Los objetos de clase sirven como factorías para generar objetos de instancia. Los objetos instanciados son objetos reales creados por una aplicación. Tienen su propio espacio de nombres.
- La instrucción `class` crea un objeto de clase con el nombre de clase. Dentro de la definición de la clase se puede crear variables de clase y métodos mediante `def` que serán compartidos por todas las instancias

Clases – representación UML



A class is a 3-compartment box



An object-oriented program consists of many well-encapsulated objects and interacting with each other by sending messages

Sintaxis de la definición de clase

- La sintaxis es:

```
class Class_name(superclass1, ...):
    """Class doc-string"""
    class_var1 = value1  # Class variables
    ......
    def __init__(self, arg1, ...):
        """Constructor"""
        self.instance_var1 = arg1  # inst var by assignment
        ......
    def __str__(self):
        """For printf() and str()"""
        ......
    def __repr__(self):
        """For repr() and interactive prompt"""
        ......
    def method_name(self, *args, **kwargs):
        """Method doc-string"""
        ......
```

Contructores

- Un constructor es un método que inicializa las variables de instancia de un objeto
  - Se invoca automáticamente cuando se crea un objeto.

```
register = CashRegister()
```

Crea una instancia de CashRegister

- Python usa el nombre especial `__init__` para el constructor porque el propósito es inicializar una instancia de la clase.

Contructor: Self

- El primer parámetro de un constructor debe ser `self`
- Cuando se invoca el constructor para crear un nuevo objeto, el parámetro `self` se asigna al objeto que esta siendo inicializado

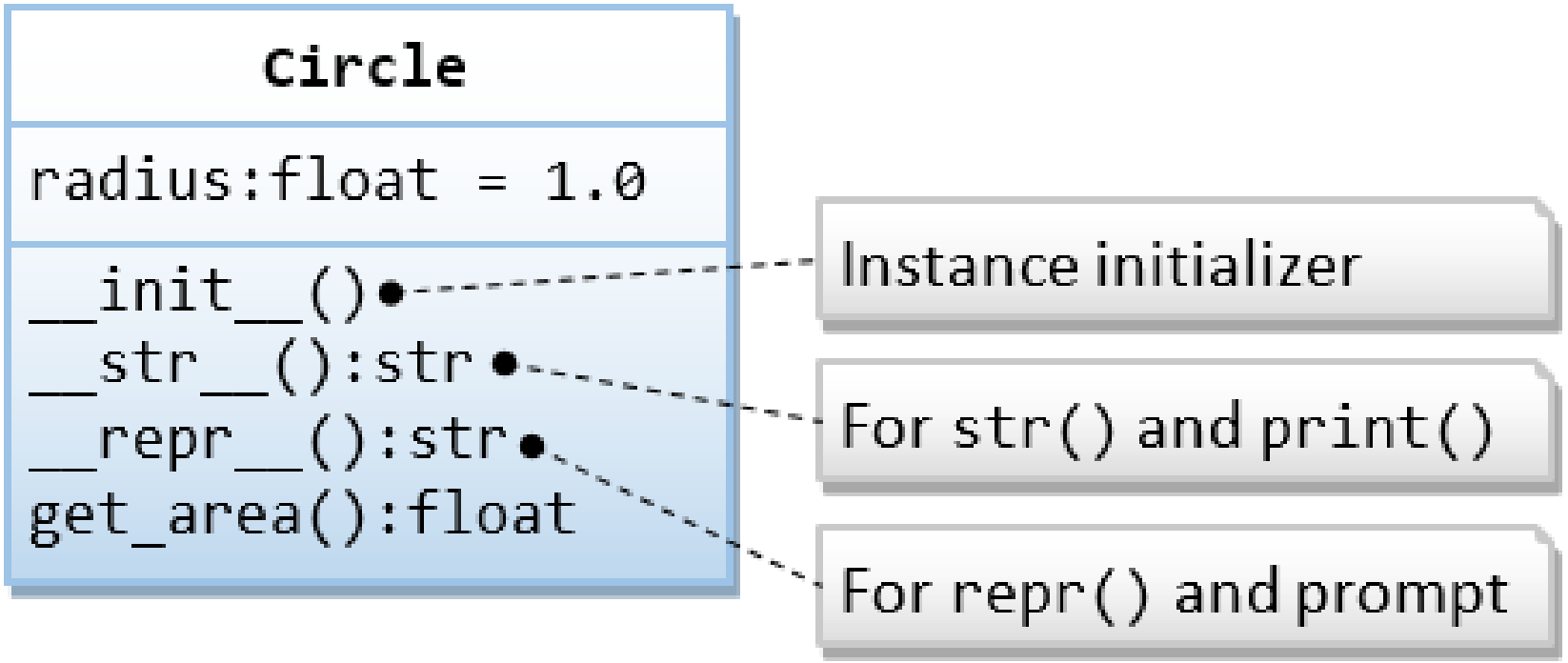
```
def __init__(self)
    self._itemCount = 0
    self._totalPrice = 0

register = CashRegister()
```

Referencia al objeto inicializado

Crea una instancia de CashRegister

Referencia al objeto creado cuando el contructor termina

Ejemplo – circle.py



Ejemplo – circle.py

- circle.py:

```
from math import pi

class Circle:
"""A Circle instance models a circle with a radius"""
    def __init__(self, radius=1.0):
        """Constructor with default radius of 1.0"""
        self.radius = radius  # Create an inst var radius
    def __str__(self):
        """Return string, invoked by print() and str()"""
        return 'circle with radius of %.2f' % self.radius
    def __repr__(self):
        """Return string used to re-create this instance"""
        return 'Circle(radius=%f)' % self.radius
    def get_area(self):
        """Return the area of this Circle instance"""
        return self.radius * self.radius * pi
```

Ejemplo – circle.py

- circle.py (cont.):

```
# if run under Python interpreter, __name__ is '__main__'.
# if imported into another module, __name__ is 'Circle'
if __name__ == '__main__':
    c1 = Circle(2.1)       # Construct an instance
    print(c1)              # Invoke __str__()
    print(c1.get_area())
    c2 = Circle()          # Default radius
    print(c2)
    print(c2.get_area())  # Invoke member method
    c2.color = 'red'  # Create new attribute via assignment
    print(c2.color)
    #print(c1.color)  # Error - c1 has no attribute color
    # Test doc-strings
    print(__doc__)                  # This module
    print(Circle.__doc__)           # Circle class
    print(Circle.get_area.__doc__)  # get_area() method
    print(isinstance(c1, Circle)) # True
```

Construcción de clase

- Para construir una instancia de una clase se realiza a través del constructor nombre_clase(…)

```
c1 = Circle(1.2)
c2 = Circle()     # radius default
```

- Python crea un objeto Circle, luego invoca el método _ _init_ _(self, radius) con self asignado a la nueva instancia
  - _ _init_ _() es un inicializador para crear variables de instancia
  - _ _init()_ _ nunca devuelve un valor
  - _ _init()_ _ es opcional y se puede omitir si no hay variables de instancia

Métodos de instancia

- Para invocar un método se usa el operador punto, en la forma nombre_obj.nombremetodo ( ). Python diferencia entre instancias de objetos y objetos de clase:
  - Para objetos de clase: se puede invocar

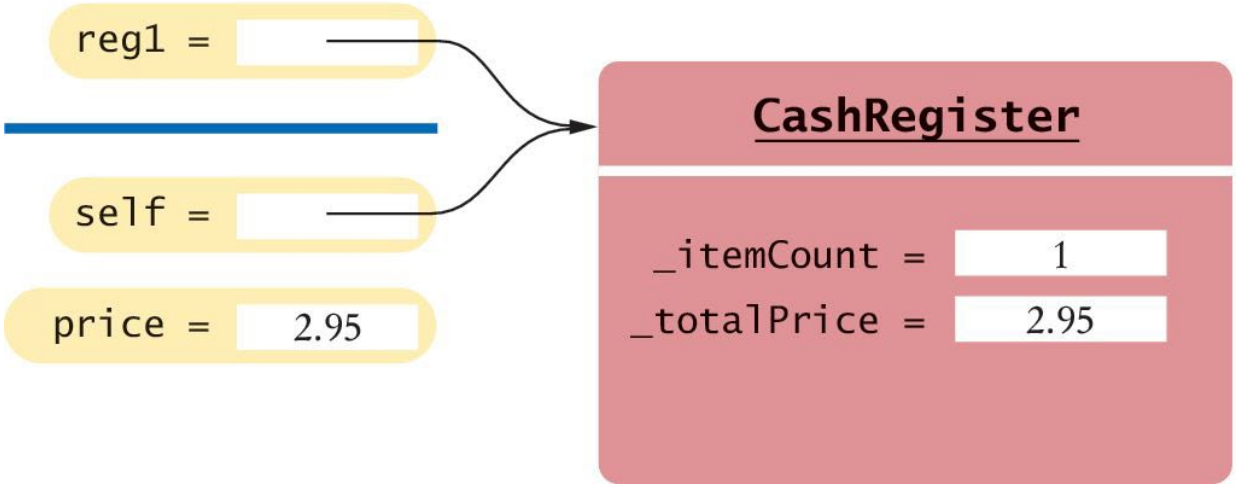
    class_name.method_name(instance_name, ...)
  - Para instancia de objetos:

    instance_name.method_name(...)

  donde instance_name es pasado en el método como el argumento 'self'

La referencia `self`

- Cada método tiene una referencia al objeto sobre el cual el método fue invocado, almacenado en la variable parámetro `self`.

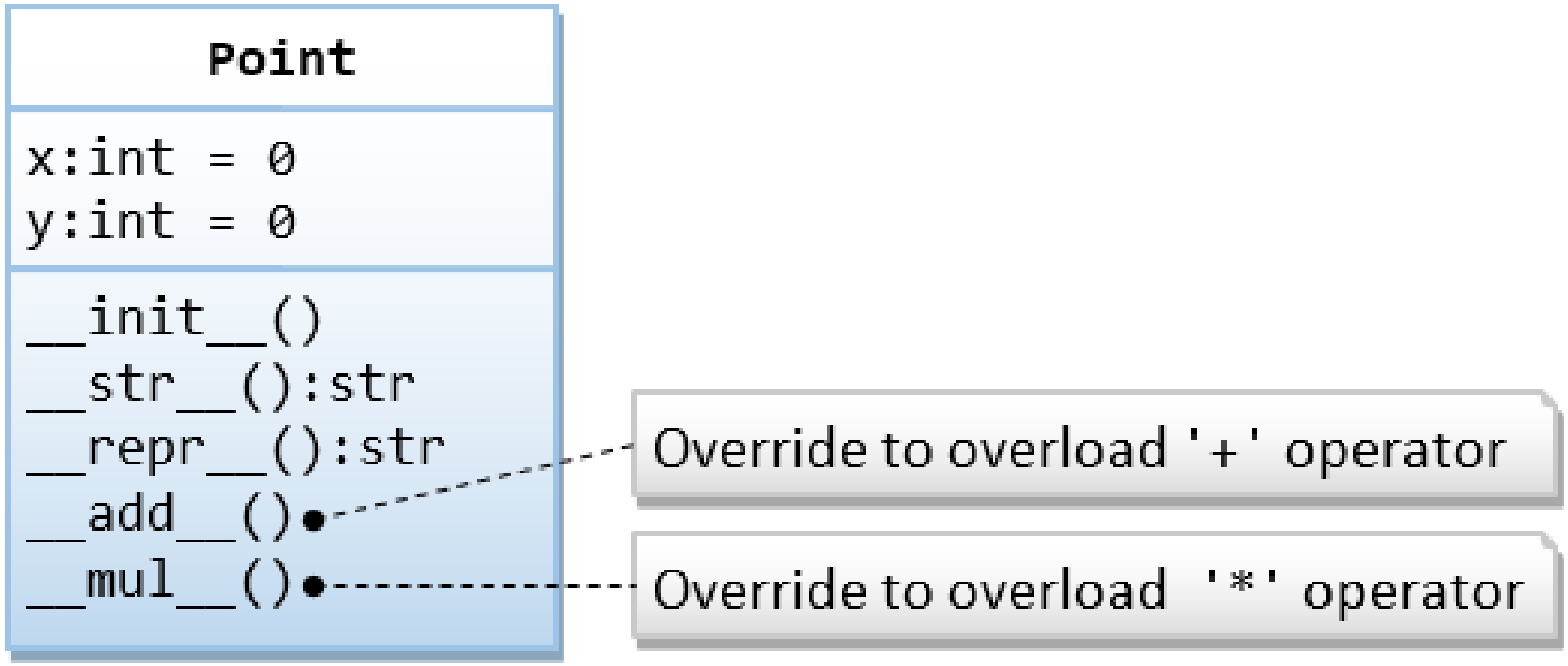



```
def addItem(self, price):
  self._itemCount = self._itemCount + 1
  self._totalPrice = self._totalPrice + price
```

Clase Point y sobrecarga de operadores

- Ejemplo: point.py que modela un punto 2D con coordenadas x e y.

  Se sobrecarga los operadores + y *:



Clase Point y sobrecarga de operadores

- point.py modela un punto 2D con coordenadas x e y. Se sobrecarga los operadores + y *:

```
""" point.py: point module defines the Point class"""
class Point:
    """A Point models a 2D point x and y coordinates"""
    def __init__(self, x=0, y=0):
        """Constructor x and y with default of (0,0)"""
        self.x = x
        self.y = y
    def __str__(self):
        """Return a descriptive string for this instance"""
        return '(%.2f, %.2f)' % (self.x, self.y)
    def __add__(self, right):
        """Override the '+' operator"""
        p = Point(self.x + right.x, self.y + right.y)
        return p
```

Clase Point y sobrecarga de operadores

```
    def __mul__(self, factor):
        """Override the '*' operator"""
        self.x *= factor
        self.y *= factor
        return self
# Test
if __name__ == '__main__':
    p1 = Point()
    print(p1)       # (0.00, 0.00)
    p1.x = 5        # direct access to property, bad idea
    p1.y = 6
    print(p1)       # (5.00, 6.00)
    p2 = Point(3, 4)
    print(p2)       # (3.00, 4.00)
    print(p1 + p2) # (8.00, 10.00) Same as p1.__add__(p2)
    print(p1)       # (5.00, 6.00)  No change
    print(p2 * 3)  # (9.00, 12.00) Same as p2.__mul__(3)
    print(p2)       # (9.00, 12.00) Changed
```

Métodos especiales (mágicos)

Magic Method	Invoked Via	Invocation Syntax
__lt__(self, right) __gt__(self, right) __le__(self, right) __ge__(self, right) __eq__(self, right) __ne__(self, right)	Comparison Operators	self < right self > right self <= right self >= right self == right self != right
__add__(self, right) __sub__(self, right) __mul__(self, right) __truediv__(self, right) __floordiv__(self, right) __mod__(self, right) __pow__(self, right)	Arithmetic Operators	self + right self - right self * right self / right self // right self % right self ** right

Métodos especiales (mágicos)

Magic Method	Invoked Via	Invocation Syntax
__and__(self, right) __or__(self, right) __xor__(self, right) __invert__(self) __lshift__(self, n) __rshift__(self, n)	Bitwise Operators	self & right self | right self ^ right ~self self << n self >> n
__str__(self) __repr__(self) __sizeof__(self)	Function call	str(self), print(self) repr(self) sizeof(self)
__len__(self) __contains__(self, item) __iter__(self) __next__(self) __getitem__(self, key) __setitem__(self, key, value) __delitem__(self, key)	Sequence Operators & Functions	len(self) item in self iter(self) next(self) self[key] self[key] = value del self[key]

Métodos especiales (mágicos)

Magic Method	Invoked Via	Invocation Syntax
__int__(self) __float__(self) __bool__(self) __oct__(self) __hex__(self)	Type Conversion Function call	int(self) float(self) bool(self) oct(self) hex(self)
__init__(self, *args) __new__(cls, *args)	Constructor	x = ClassName(*args)
__del__(self)	Operator del	del x
__index__(self)	Convert this object to an index	x[self]
__radd__(self, left) __rsub__(self, left) ...	RHS (Reflected) addition, subtraction, etc.	left + self left - self ...
__iadd__(self, right) __isub__(self, right) ...	In-place addition, subtraction, etc	self += right self -= right ...

Métodos especiales (mágicos)

Magic Method	Invoked Via	Invocation Syntax
__pos__(self) __neg__(self)	Unary Positive and Negative operators	+self -self
__round__(self) __floor__(self) __ceil__(self) __trunc__(self)	Function Call	round(self) floor(self) ceil(self) trunc(self)
__getattr__(self, name) __setattr__(self, name, value) __delattr__(self, name)	Object's attributes	self.name self.name = value del self.name
__call__(self, *args, **kwargs)	Callable Object	obj(*args, **kwargs);
__enter__(self), __exit__(self)	Context Manager with-statement	

Variables de clase

- Son valores que pertenecen a la clase y no al objeto de la clase.
- Las variables de clase se llaman también “variables estáticas”
- Las variables de clase se declaran al mismo nivel que los métodos.
- Las variables de clase siempre deben ser privadas para asegurar que métodos de otras clases no cambian sus valores. Las *constantes* de clase pueden ser públicas

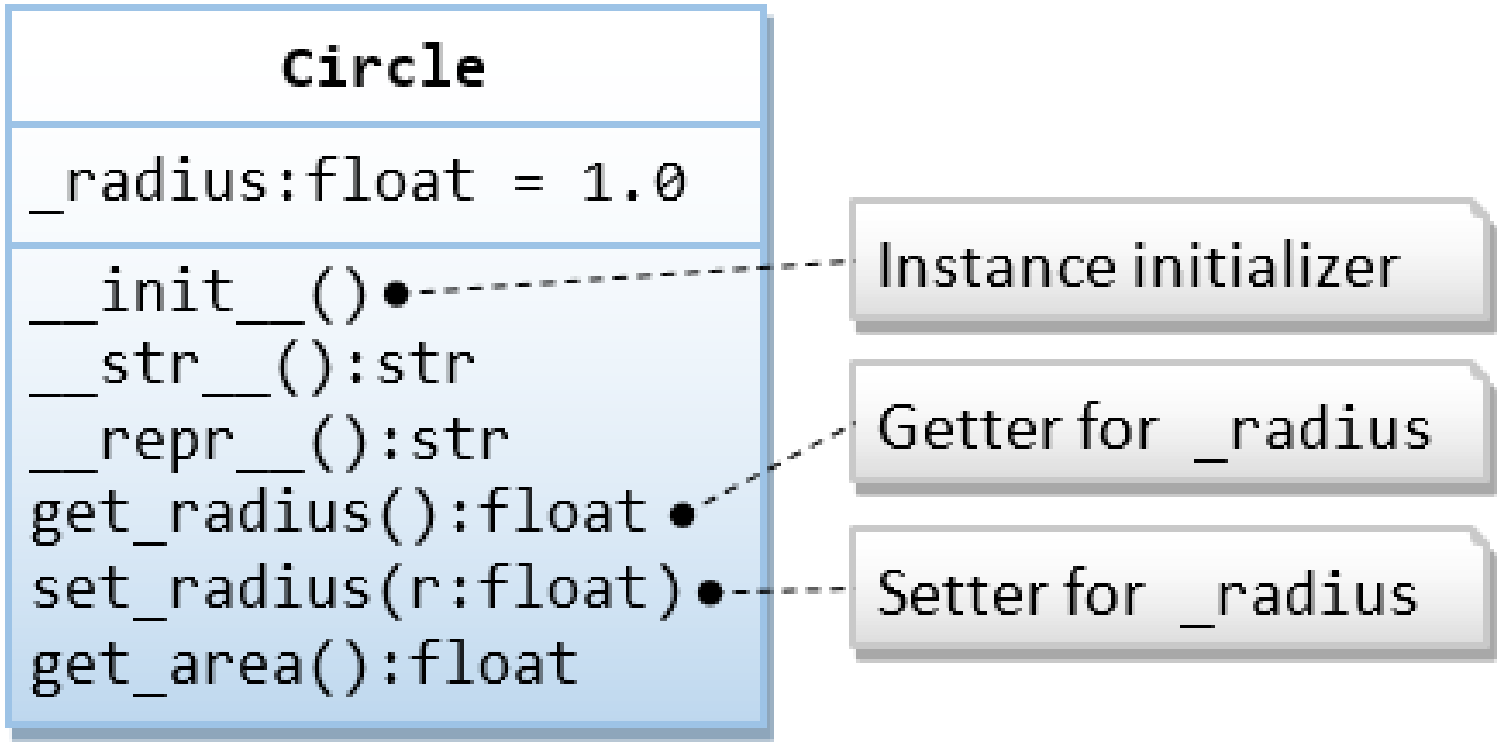
Variables de clase - Ejemplo

- Se desea asignar números de cuenta de banco secuencialmente empezando por 1001.

```
class BankAccount :
    _lastAssignedNumber = 1000    # A class variable
    OVERDRAFT_FEE = 29.95         # Class constant
    def _ _init_ _(self) :
        self._balance = 0
        BankAccount._lastAssignedNumber =
           BankAccount._lastAssignedNumber + 1
        self._accountNumber =
           BankAccount._lastAssignedNumber
```

Métodos de acceso (get) y asignación (set)

- Mejora de la clase Circle con métodos get y set para acceder las variables de instancia (circle1.py)



Métodos de acceso (get) y asignación (set)

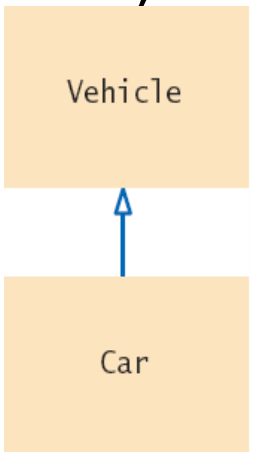
```
"""circle.py: The circle module, which defines the Circle class"""
from math import pi

class Circle:
    """A Circle instance models a circle with a radius"""
    def __init__(self, _radius = 1.0):
        """Initializer with default radius of 1.0"""
        # Change from radius to _radius (meant for internal use)
        # You should access through the getter and setter.
        self.set_radius(_radius)   # Call setter
    def set_radius(self, _radius):
        """Setter for instance variable radius with input validation"""
        if _radius < 0:
            raise ValueError('Radius shall be non-negative')
        self._radius = _radius
    def get_radius(self):
        """Getter for instance variable radius"""
        return self._radius
    def get_area(self):
        """Return the area of this Circle instance"""
        return self.get_radius() * self.get_radius() * pi  # Call getter
```

Métodos de acceso (get) y asignación (set)

```
    ...
    def __repr__(self):
        """Return a command string to recreate this instance"""
        # Used by str() too as __str__() is not defined
        return 'Circle(radius={})'.format(self.get_radius())  # Call getter

if __name__ == '__main__':
    c1 = Circle(1.2)         # Constructor and Initializer
    print(c1)                # Invoke __repr__(). Output:
Circle(radius=1.200000)
    print(vars(c1))          # Output: {'_radius': 1.2}
    print(c1.get_area())     # Output: 4.52389342117
    print(c1.get_radius())   # Run Getter. Output: 1.2
    c1.set_radius(3.4)       # Test Setter
    print(c1)                # Output: Circle(radius=3.400000)
    c1._radius = 5.6         # Access instance variable directly (NOT
                             # recommended but permitted)
    print(c1)                # Output: Circle(radius=5.600000)
    c2 = Circle()  # Default radius
    print(c2)      # Output: Circle(radius=1.000000)

    c3 = Circle(-5.6)  # ValueError: Radius shall be non-negative
```

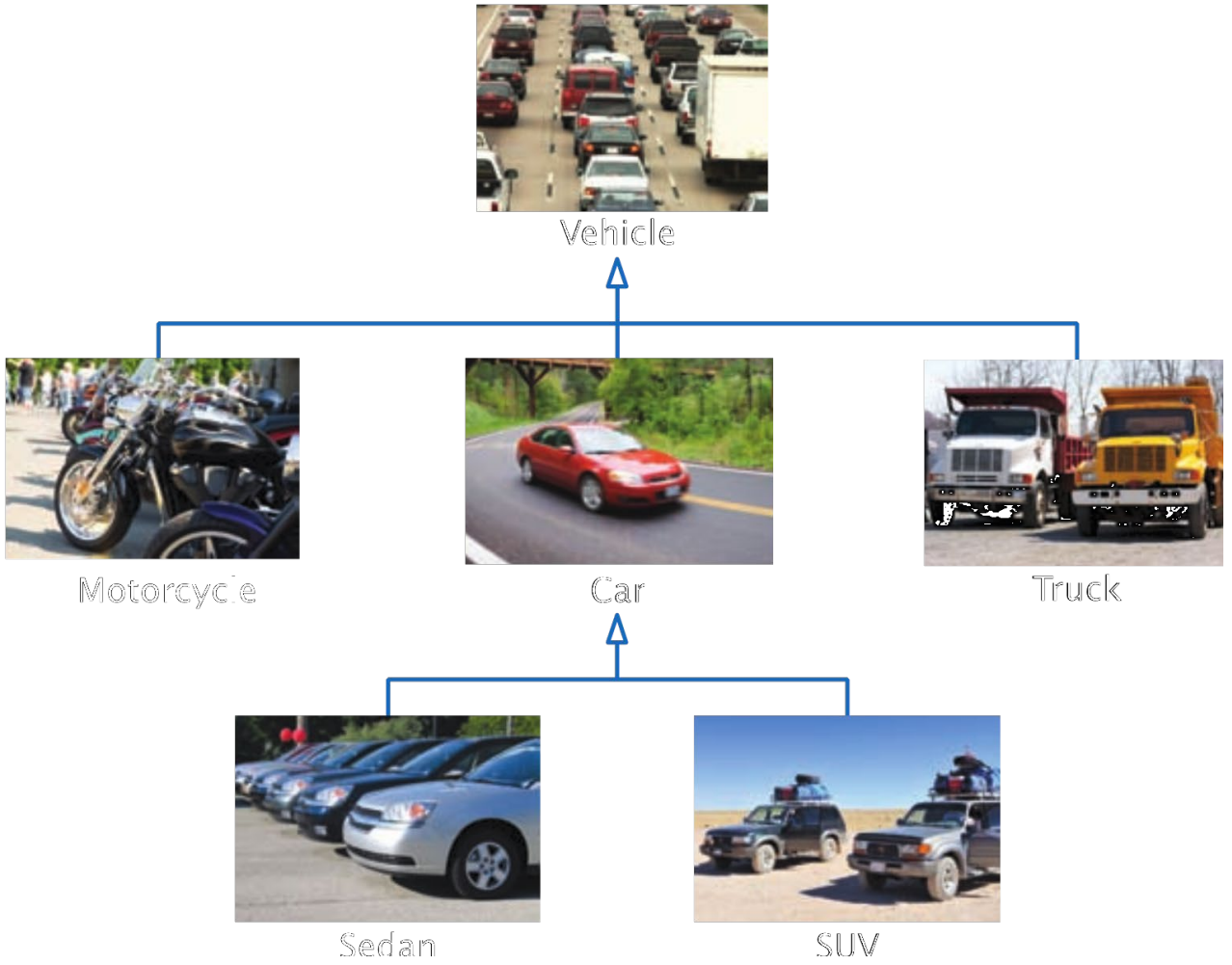
Herencia - Jerarquía

- En programación orientada a objetos, la **herencia** es una relación entre:
  - Una superclase: clase más general
  - Una subclase: clase más especializada
- La subclase 'hereda' los datos (variables) y comportamiento (métodos) de la superclase.



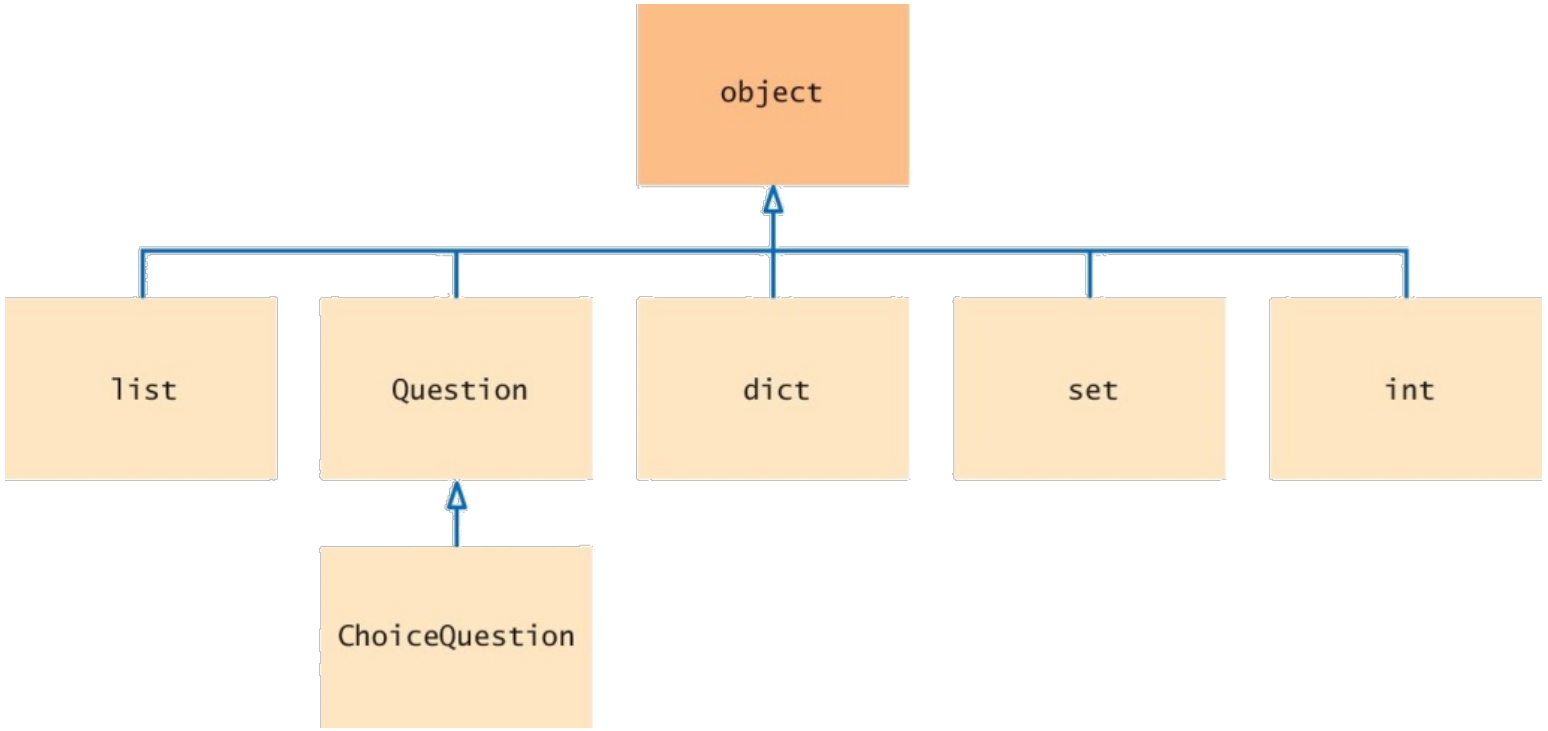
Jerarquía de la clase Vehicle

- General
- Especializado
- Más específico



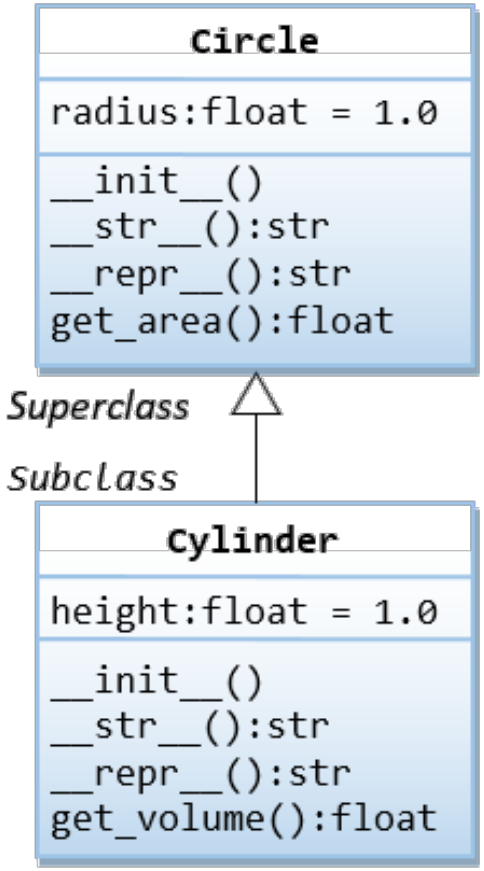
La superclase `object`

- En Python, cada clase que es declarada sin una superclase explícita, automáticamente extiende la clase object.



Herencia y polimorfismo

- Se puede definir una clase Cylinder como una subclase de Circle.



cylinder.py

Hereda atributo radius y método get_area()

Herencia

- cylinder.py un cilindro se puede derivar de un circulo

```
"""cylinder.py: which defines the Cylinder class"""
from circle import Circle  # Using the Circle class
class Cylinder(Circle):
    """The Cylinder class is a subclass of Circle"""
    def __init__(self, radius = 1.0, height = 1.0):
        """Constructor"""
        super().__init__(radius)  # Invoke superclass
        self.height = height
    def __str__(self):
        """Self Description for print()"""
        return 'Cylinder(radius=%.2f,height=%.2f)' %\
                (self.radius, self.height)
    def get_volume(self):
        """Return the volume of the cylinder"""
        return self.get_area() * self.height
```

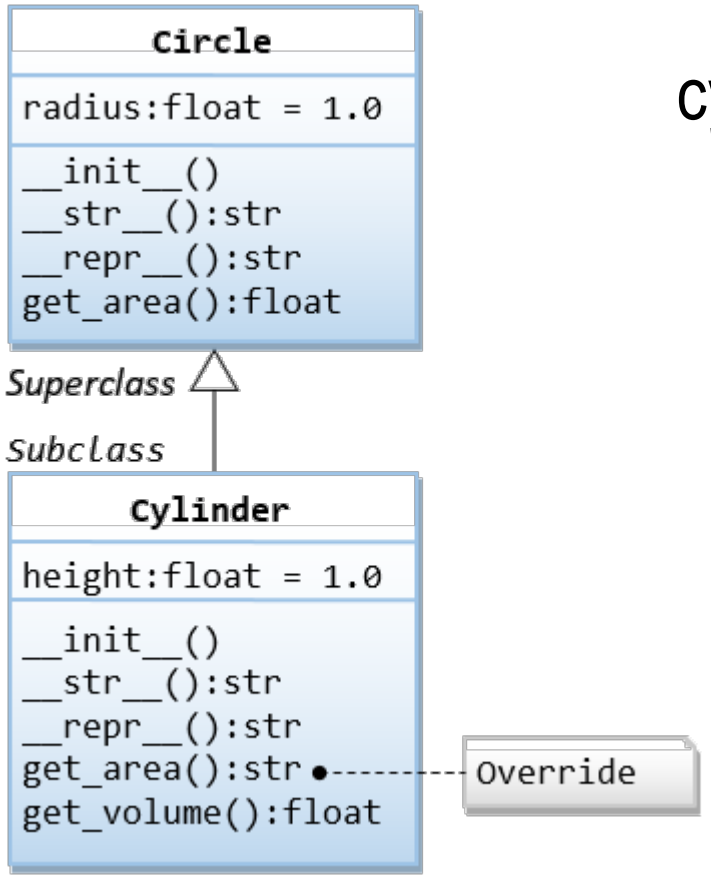
Herencia

- cylinder.py (cont.)

```
if __name__ == '__main__':
    cy1 = Cylinder(1.1, 2.2)
    print(cy1)
    print(cy1.get_area())    # inherited superclass' method
    print(cy1.get_volume())   # Invoke its method
    cy2 = Cylinder()           # Default radius and height
    print(cy2)
    print(cy2.get_area())
    print(cy2.get_volume())
    print(dir(cy1))
    print(Cylinder.get_area)
    print(Circle.get_area)
    c1 = Circle(3.3)
    print(c1)    # Output: circle with radius of 3.30
    print(issubclass(Cylinder, Circle))  # True
    print(issubclass(Circle, Cylinder))  # False
    print(isinstance(cy1, Cylinder))     # True
```

Sobreescritura (overriding) de métodos

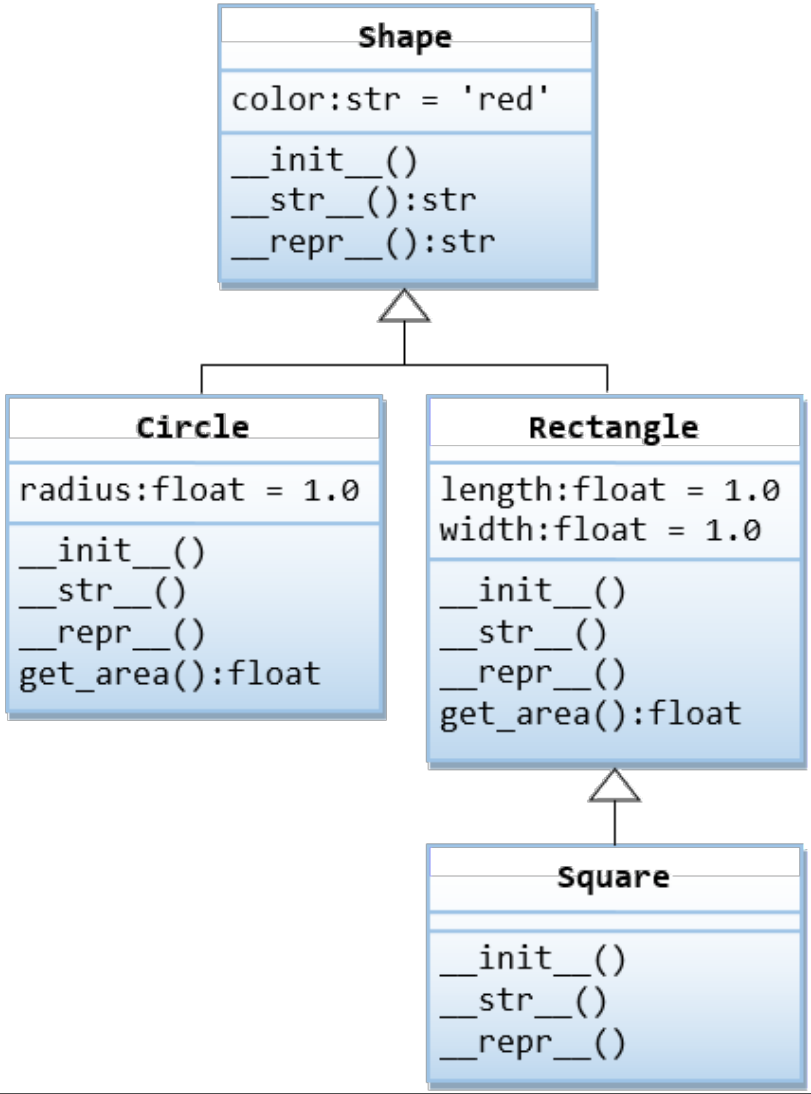
- Se puede *sobreescribir* el método get_area( ) para calcular la superficie del cilindro. También __str__( )



cylinder1.py

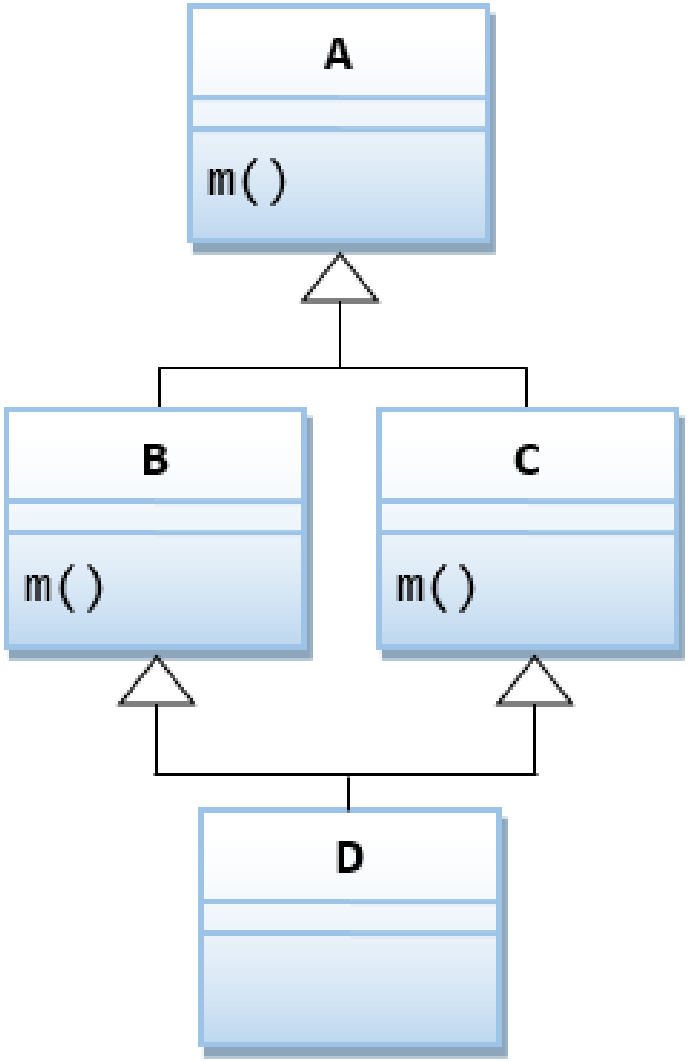
Ejemplo shape y subclases

- shape.py



Herencia múltiple

- minh.py
- minh1.py
- minh2.py
- minh3.py



Tipo de dato definido por el usuario – Clase Charge

- Se define un tipo Charge para partículas cargadas.
- Se usa la ley de Coulomb para el cálculo del potencial de un punto debido a una carga V=kq/r , donde q es el valor de la carga, r es la distancia del punto a la carga y k=8.99 × 10^9 N m^2/C^2.

Constructor

operation	*description*
`Charge(x0, y0, q0)`	*a new charge centered at (x0, y0) with charge value q0*
`c.potentialAt(x, y)`	*electric potential of charge c at point (x, y)*
`str(c)`	*'q0 at (x0, y0)' (string representation of charge c)*

API for our user-defined Charge data type

Convenciones sobre ficheros

- El código que define el tipo de dato definido por el usuario Charge se coloca en un fichero del mismo nombre (sin mayúscula) charge.py
- Un programa cliente que usa el tipo de dato Charge se pone en el cabecero del programa:

```
from charge import Charge
```

Elementos básicos de un tipo de dato

- API

operation	*description*
`Charge(x0, y0, q0)`	*a new charge centered at* (x_0, y_0) *with charge value* q_0
`c.potentialAt(x, y)`	*electric potential of charge* `c` *at point* `(x, y)`
`str(c)`	`'q0 at (x0, y0)'` *(string representation of charge* `c`*)*

API for our user-defined `Charge` *data type*

- Clase. Fichero charge.py. Palabra reservada class
- Constructor. Método especial __init__(), self
- Variable de instancia. _nombrevar
- Métodos. Variable de instancia self
- Funciones intrínsecas. __str()__
- Privacidad

Implementación de clase Charge – fichero charge.py

```
import sys
import math

class Charge:
    def __init__(self, x0, y0, q0):
        self._rx = x0  # x value of the query point
        self._ry = y0  # y value of the query point
        self._q = q0   # Charge

    def potentialAt(self, x, y):
        COULOMB = 8.99e09
        dx = x - self._rx
        dy = y - self._ry
        r = math.sqrt(dx*dx + dy*dy)
        if r == 0.0: # Avoid division by 0
            if self._q >= 0.0:
                return float('inf')
            else:
                return float('-inf')
        return COULOMB * self._q / r

    def __str__(self):
        result = str(self._q) + ' at (' + str(self._rx) + ', ' + str(self._ry) + ')'
        return result
```

Creación de objetos, llamada de métodos y representación de String

```
#-----------------
# chargeclient.py
#-----------------
import sys
from charge import Charge
# Acepta floats x e y como argumentos en la línea de comandos. Crea dos objetos
# Charge con posición y carga. Imprime el potencial en (x, y) en la salida estandard
x = float(sys.argv[1])
y = float(sys.argv[2])
c1 = Charge(.51, .63, 21.3)
c2 = Charge(.13, .94, 81.9)
v1 = c1.potentialAt(x, y)
v2 = c2.potentialAt(x, y)
print('potential at (%.2f, %.2f) due to\n', x, y)
print('  ' + str(c1) + ' and')
print('  ' + str(c2))
print('is %.2e\n', v1+v2)
#-----------------------------------------------------------------------
# python chargeclient.py .2 .5
# potential at (0.20, 0.50) due to
#   21.3 at (0.51, 0.63) and
#   81.9 at (0.13, 0.94)
# is 2.22e+12
```

Clase Complex

- Métodos especiales. En Python la expresión a * b se reemplaza con la llamada del método a.__mul__(b)

client operation	*special method*	*description*
`Complex(x, y)`	`__init__(self, re, im)`	*new* `Complex` *object with value* `x+yi`
`a.re()`		*real part of* `a`
`a.im()`		*imaginary part of* `a`
`a + b`	`__add__(self, other)`	*sum of* `a` *and* `b`
`a * b`	`__mul__(self, other)`	*product of* `a` *and* `b`
`abs(a)`	`__abs__(self)`	*magnitude of* `a`
`str(a)`	`__str__(self)`	`'x + yi'` *(string representation of* `a`*)*

API for a user-defined `Complex` *data type*

Sobrecarga de operadores

client operation	*special method*	*description*
`x + y`	`__add__(self, other)`	*sum of x and y*
`x - y`	`__sub__(self, other)`	*difference of x and y*
`x * y`	`__mul__(self, other)`	*product of x and y*
`x ** y`	`__pow__(self, other)`	*x to the yth power*
`x / y`	`__truediv__(self, other)`	*quotient of x and y*
`x // y`	`__floordiv__(self, other)`	*floored quotient of x and y*
`x % y`	`__mod__(self, other)`	*remainder when dividing x by y*
`+x`	`__pos__(self)`	*x*
`-x`	`__neg__(self)`	*arithmetic negation of x*

Special methods for arithmetic operators

Algoritmos de búsqueda

- Secuencial $O(n)$
- Búsqueda binaria $O(\log n)$

Algoritmos de ordenación

- Burbuja $O(n^2)$
- Inserción $O(n^2)$
- Selección $O(n^2)$
- Mezcla $O(n \log n)$
- Quicksort $O(n \log n)$

Estructuras de datos

- Pilas. Lista LIFO
- Colas. Lista FIFO
- Listas enlazadas
- Arboles
- Tabla de símbolos
- Grafos