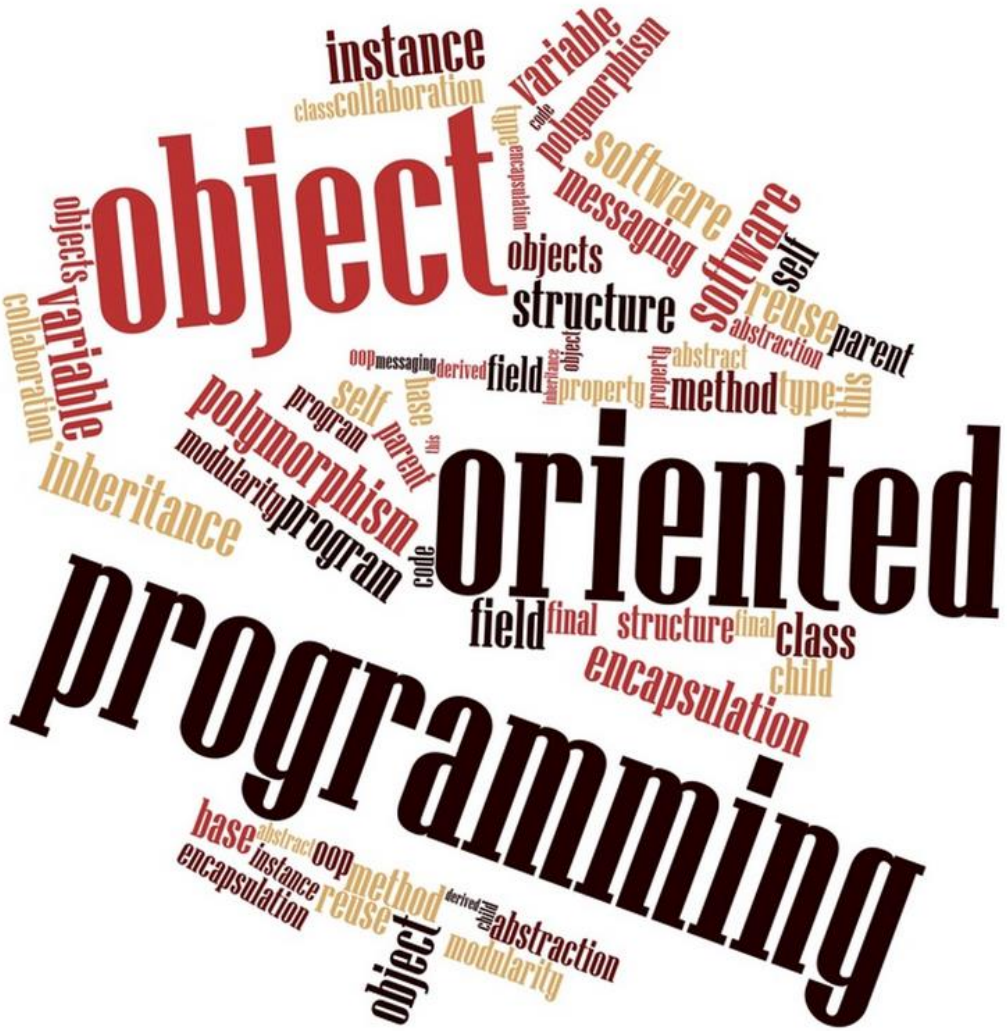

instance
variable
polymorphism
class
collaboration
code
encapsulation
object
software
messaging
objects
structure
software
self
reuse
abstraction
parent
objects
variable
collaboration
oop
messaging
derived
field
inheritance
object
property
abstract
method
type
this
self
base
program
parent
this
polymorphism
modularity
program
inheritance
code
oriented
field
final
structure
final
class
child
encapsulation
programming
base
abstract
oop
instance
encapsulation
method
reuse
derived
child
abstraction
object
modularity

OBJECTS

- Python supports many different kinds of data

```
1234        3.14159        "Hello"        [1, 5, 7, 11, 13]
{"CA": "California", "MA": "Massachusetts"}
```

- each is an **object**, and every object has:
  - a **type**
  - an internal **data representation** (primitive or composite)
  - a set of procedures for **interaction** with the object
- an object is an **instance** of a type
  - `1234` is an instance of an `int`
  - `"hello"` is an instance of a string

OBJECTS

- **EVERYTHING IN PYTHON IS AN OBJECT** (and has a type)
- can **create new objects** of some type
- can **manipulate objects**
- can **destroy objects**
  - explicitly using `del` or just "forget" about them
  - python system will reclaim destroyed or inaccessible objects – called "garbage collection"

OBJECTS

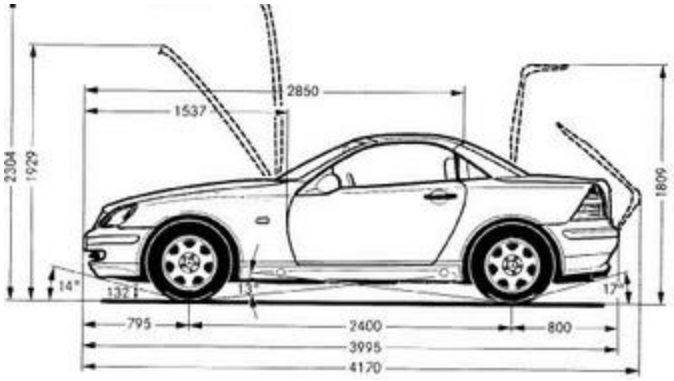
- objects are **a data abstraction** that captures…

(1) an **internal representation**

- through data attributes

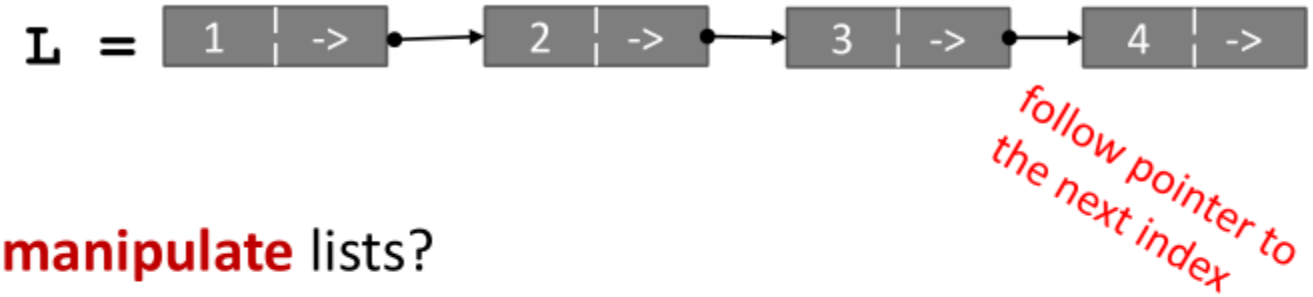
(2) an **interface** for interacting with object

- through methods (aka procedures/functions)
- defines behaviors but hides implementation



OBJECTS

- how are lists **represented internally**? linked list of cells



- how to **manipulate** lists?
  - `L[i], L[i:j], +`
  - `len(), min(), max(), del(L[i])`
  - `L.append(),L.extend(),L.count(),L.index(), L.insert(),L.pop(),L.remove(),L.reverse(), L.sort()`
- internal representation should be private
- correct behavior may be compromised if you manipulate internal representation directly

VENTAJAS DE POO

Agrupar datos en paquetes junto con procedimientos que funcionan en ellos a través de interfaces bien definidas.

Desarrollo de divide y vencerás:

- implementar y probar el comportamiento de cada clase por separado.
- Mayor modularidad reduce la complejidad.

Las clases facilitan la reutilización del código:

- muchos módulos de Python definen nuevas clases.
- cada clase tiene un entorno separado (sin colisión en los nombres de las funciones).
- La herencia permite que las subclases redefinan o amplíen un subconjunto seleccionado de un comportamiento de superclase.

CREATING AND USING YOUR OWN TYPES WITH CLASSES

Hacer una distinción entre crear una clase y usar una instancia de la clase.

crear la clase implica:

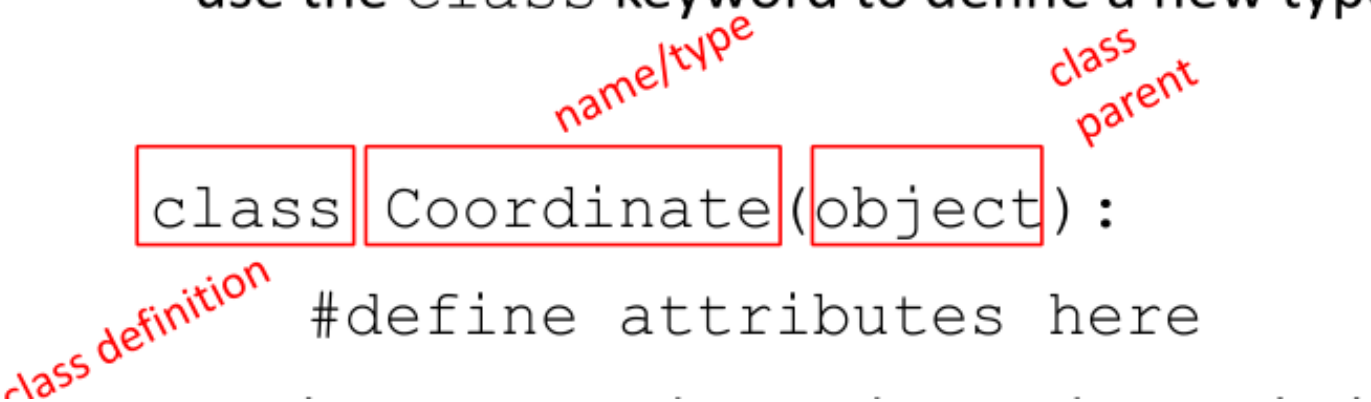
- Definiendo el nombre de la clase
- Definiendo los atributos de la clase
- por ejemplo, alguien escribió código para implementar una clase de lista

usar la clase implica:

- creando nuevas instancias de objetos.
- haciendo operaciones sobre las instancias.
- por ejemplo, L = [1,2] y len (L).

DEFINE YOUR OWN TYPES

- use the `class` keyword to define a new type



similar a def, indentar para indicar qué declaraciones son parte de la definición de clase.

la palabra object significa que Coordinate es un objeto de Python y hereda todos sus atributos

- Coordinate es una subclase de objeto.
- objeto es una superclase de Coordinate.

WHAT ARE ATTRIBUTES?

datos y procedimientos que pertenecen a la clase.

atributos de datos:

- Piense en los datos como otros objetos que componen la clase.
- por ejemplo, una coordenada se compone de dos números.

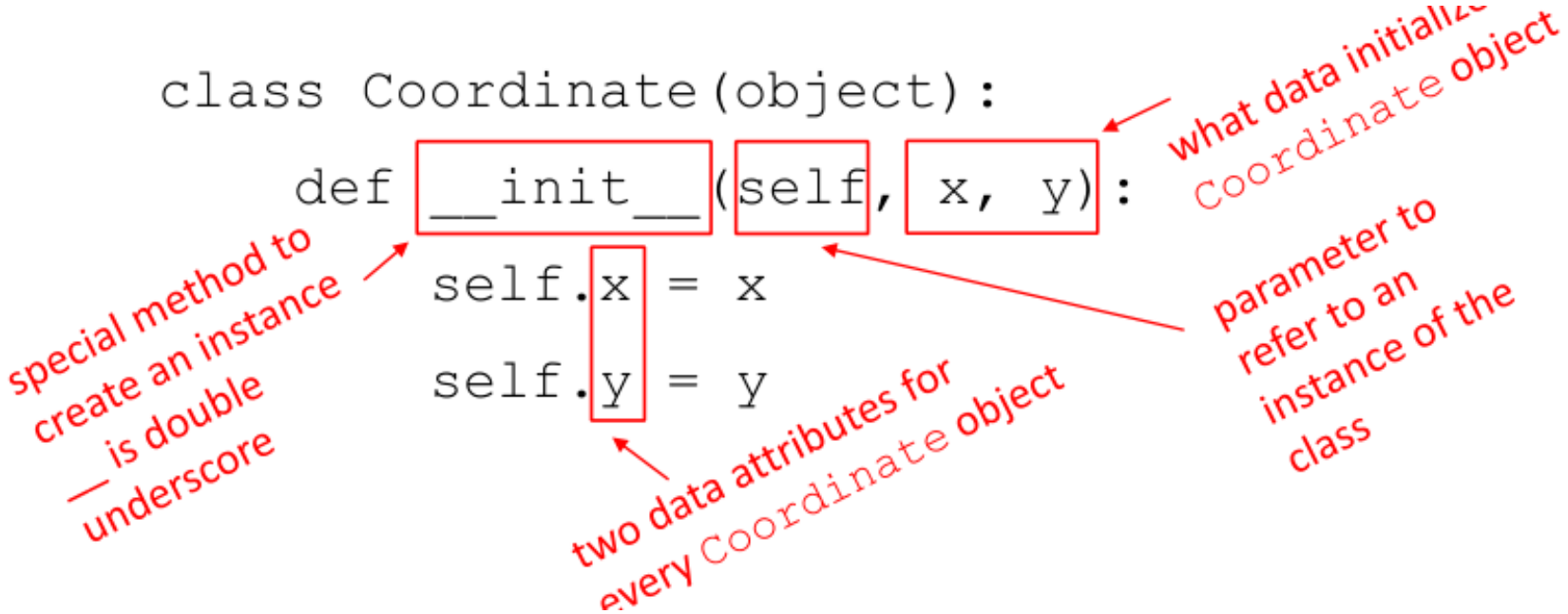
métodos (atributos de procedimiento):

- Piense en los métodos como funciones que solo funcionan con esta clase.
- cómo interactuar con el objeto.
- por ejemplo, puede definir una distancia entre dos objetos de coordenadas.

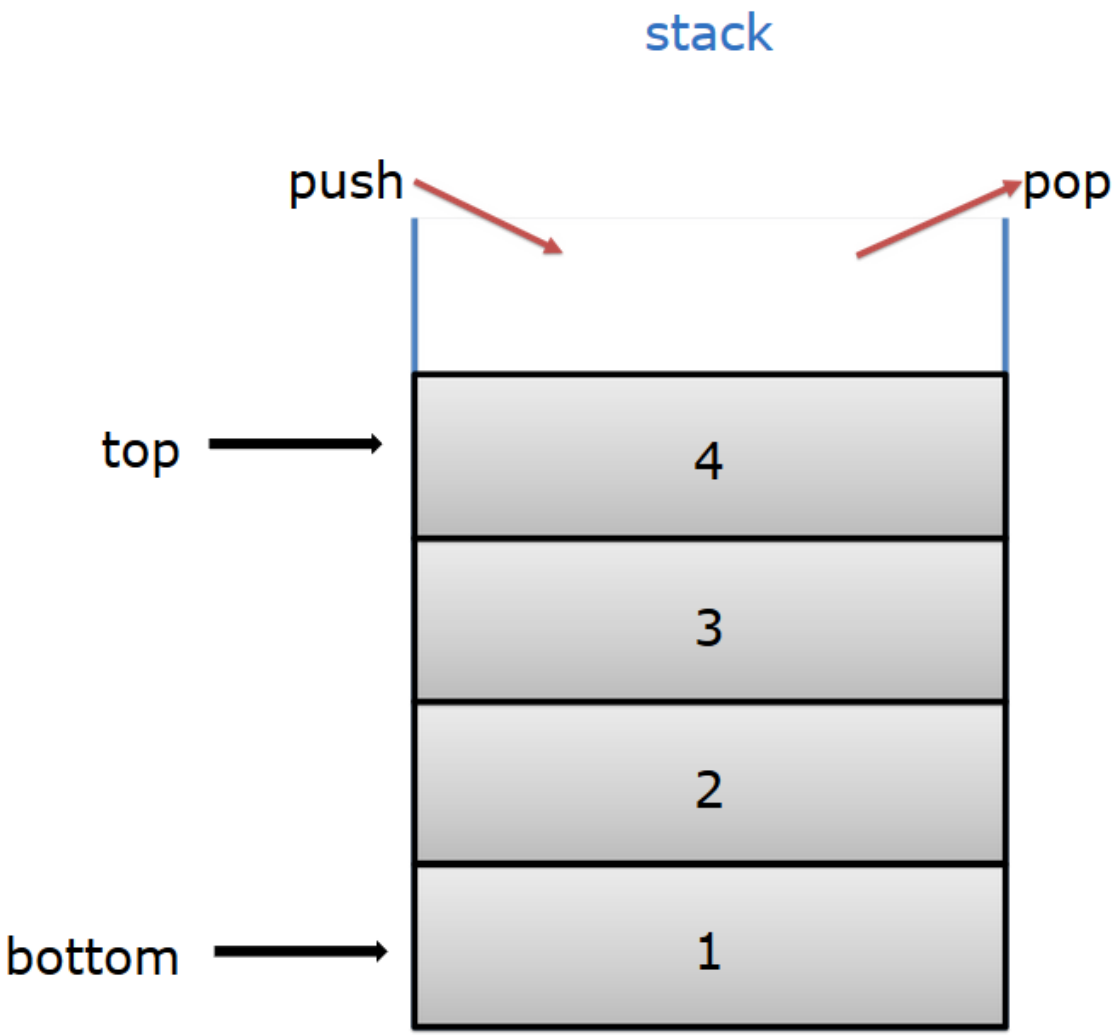
DEFINING HOW TO CREATE AN INSTANCE OF A CLASS

Primero tienes que definir cómo crear una instancia de objeto.

use un método especial llamado __init__ para inicializar algunos atributos de datos



STACK - LIFO



STACK - LIFO

Ejercicio 1 - Funciones

```
stack = []
```

```
def push(val):
    stack.append(val)
```

```
def pop(val):
    val = stack[-1]
    del stack[-1]
    return val
```

```
stack = []

def push(val):
    stack.append(val)

def pop():
    val = stack[-1]
    del stack[-1]
    return val

push(3)
push(2)
push(1)
print(pop())
print(pop())
print(pop())
```

CLASS

Esta función se llama constructor, ya que su propósito general es construir un nuevo objeto. El constructor debe saber todo sobre la estructura del objeto y debe realizar todas las inicializaciones necesarias.

```
class Stack:
   def __init__(self):
        print('Hi')

stack = Stack()
```

← constructor

← Instanciación de la clase Stack

```
class Stack:
   def __init__(self):
        self.stk = []

stack = Stack()
print(len(stack.stk))
```

← Los atributos debiesen ser siempre privados, para eso es importante trabajar la encapsulación de los datos

CLASS

```
class Stack:
    def __init__(self):
        self.__stk = []

stack = Stack()
print(len(stack.__stk))
```

El __, indica que la variable es privada y solo puede ser accedida desde la clase

CLASS

```
class Stack:
   def __init__(self):
        self.__stk = []

   def push(self, val):
        self.__stk.append(val)

   def pop(self):
        val = self.__stk[-1]
        del self.__stk[-1]
        return val

stack = Stack()
stack.push(3)
stack.push(2)
stack.push(1)
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

CLASS

```
class Stack:
def __init__(self):
   self.__stk = []

def push(self, val):
       self.__stk.append(val)

def pop(self):
       val = self.__stk[-1]
       del self.__stk[-1]
       return val

stack1 = Stack()
stack2 = Stack()
stack1.push(3)
stack2.push(stack1.pop())
print(stack2.pop())
```

CLASS

```
class Stack:
def __init__(self):
  self.__stk = []

def push(self, val):
      self.__stk.append(val)

def pop(self):
      val = self.__stk[-1]
      del self.__stk[-1]
      return val

little_stack = Stack()
another_stack = Stack()
funny_stack = Stack()
little_stack.push(1)
another_stack.push(little_stack.pop() + 1)
funny_stack.push(another_stack.pop() - 2)
print(funny_stack.pop())
```

SUBCLASS

Ejercicio 2 - Herencia

```
class AddingStack(Stack):
  def __init__(self):
      Stack.__init__(self)
      self.__sum = 0
```

```
def push(self, val):
  self.__sum += val
  Stack.push(self,val)
```

```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

```
def getSum(self):
  return self.__sum
```

INSTANCE VARIABLES

```
class Class:
   def __init__(self,val=1):
        self.First = val

   def setSecond(self,val):
        self.Second = val

object1 = Class()
object2 = Class(2)
object2.setSecond(3)
object3 = Class(4)
object3.Third = 5

print(object1.__dict__)
print(object2.__dict__)
print(object3.__dict__)
```

CLASS VARIABLES

Ejercicio 3 – Variables de clases

```
class Class:
        Counter = 0
        def __init__(self,val=1):
            self.__First = val
            Class.Counter += 1

object1 = Class()
object2 = Class(2)
object3 = Class(4)

print(object1.__dict__, object1.Counter)
print(object2.__dict__, object2.Counter)
print(object3.__dict__, object3.Counter)
```

CHECKING ATTRIBUTES

Ejercicio 4 – Variables de clases

```
class Class:
        def __init__(self,val):
                if val % 2 != 0:
                        self.a = 1
                else:
                        self.b = 1

object = Class(1)
print(object.a)
print(object.b)
```

CHECKING ATTRIBUTES

Ejercicio 5 – Variables de clases

```
class Class:
        def __init__(self,val):
                if val % 2 != 0:
                        self.a = 1
                else:
                        self.b = 1

object = Class(1)
print(object.a)
try:
        print(object.b)
except AttributeError:
        pass
```

CHECKING ATTRIBUTES

Ejercicio 6 – Variables de clases

```
class Class:
        def __init__(self,val):
                if val % 2 != 0:
                        self.a = 1
                else:
                        self.b = 1

object = Class(1)
print(object.a)
if hasattr(object,'b'):
        print(object.b)
```

Hasattr, espera que se le pasen dos argumentos: la clase o el objeto que se está comprobando; el nombre de la propiedad cuya existencia se debe informar (debe ser cadena). La función devuelve Verdadero o Falso.

```
class Class:
        Attr = 1

print(hasattr(Class,'Attr'))
print(hasattr(Class,'Prop'))
```