

---

## **Tema 2: Clases**

- Programación Orientada a Objetos
- Herencia, Encapsulación, polimorfismo
- Programación de clases

# Programación orientada a objetos

---

- ◆ En Programación Orientada a Objetos (POO) se representa un modelo de la interacción de las cosas en el mundo real
  - Un programa consta de un conjunto de objetos que representan elementos del mundo real
  - Interesa *qué* hacen los objetos más que *cómo* se hace

# Programación orientada a objetos

---

- ◆ La programación procedural se basa en módulos de funciones
- ◆ Los datos y las funciones se consideran de forma separada.
- ◆ POO combina los datos y sus funciones

# Programación orientada a objetos

---

- ◆ La POO proporciona una forma de pensar más natural sobre el problema, no sobre funciones y procedimientos
- ◆ Permite reusabilidad de código y soporta modificaciones de requerimientos o tecnología.

# Programación orientada a objetos

---

- Cada objeto es responsable de unas tareas
- Los objetos interactúan entre sí por medio de mensajes
- Cada objeto pertenece a una clase (es un ejemplar de)
- Las clases se pueden organizar en una jerarquía con herencia

# Clases

---

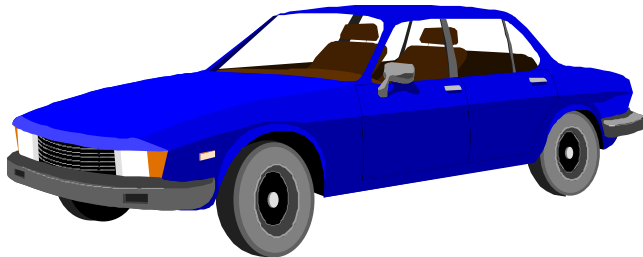
- ◆ Los objetos similares se agrupan en clases con estados similares y mismo comportamiento
- ◆ Las clases son “plantillas” que especifican el comportamiento y los atributos de los ejemplares (objetos) de la clase



# Datos o Atributos

---

- ◆ Valores o características de los objetos
- ◆ Permiten definir el estado del objeto u otras cualidades



- Velocidad
- Aceleración
- Capacidad de combustible



variables

- Marca
- Color
- Potencia
- Velocidad máxima
- Carburante



constantes

# Métodos

---

- ◆ Un objeto puede realizar una serie de acciones
  - Definen la funcionalidad y comportamiento de un objeto
  - Son los mensajes para realizar una acción en un objeto
  - Equivalen a las funciones en otros lenguajes de programación



- Arrancar motor
- Parar motor
- Acelerar
- Frenar
- Girar a la derecha (grados)
- Girar a la izquierda (grados)
- Cambiar de marcha (nueva marcha)

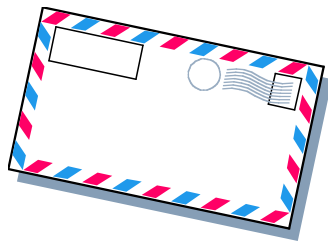
↑                      ↑  
método              argumentos



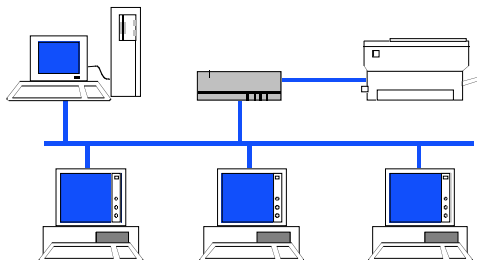
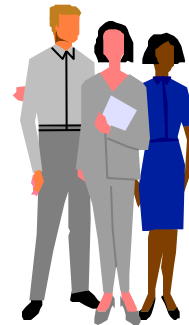
# Objetos

---

- ◆ Los objetos son cosas
- ◆ Los objetos pueden ser simples o complejos
- ◆ Los objetos pueden ser reales o imaginarios



Fecha



*Hola Mundo*

# Orientación a objetos

---

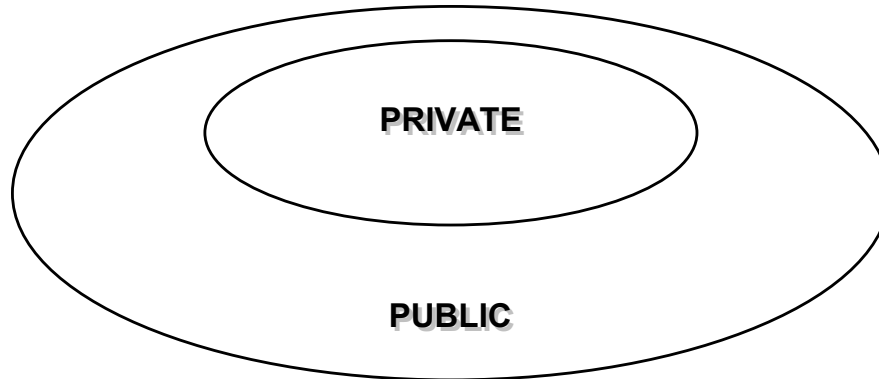
- ◆ Proporciona los siguientes mecanismos
  - Encapsulación
  - Herencia
  - Abstracción
  - Polimorfismo

# Encapsulación

---

## ◆ *Ocultación* de información

- Las partes necesarias para utilizar un objeto son visibles (*interfaz pública*): métodos
- Las demás partes son ocultas (privadas)



## ◆ En Python no hay encapsulación

# Variables Privadas

---

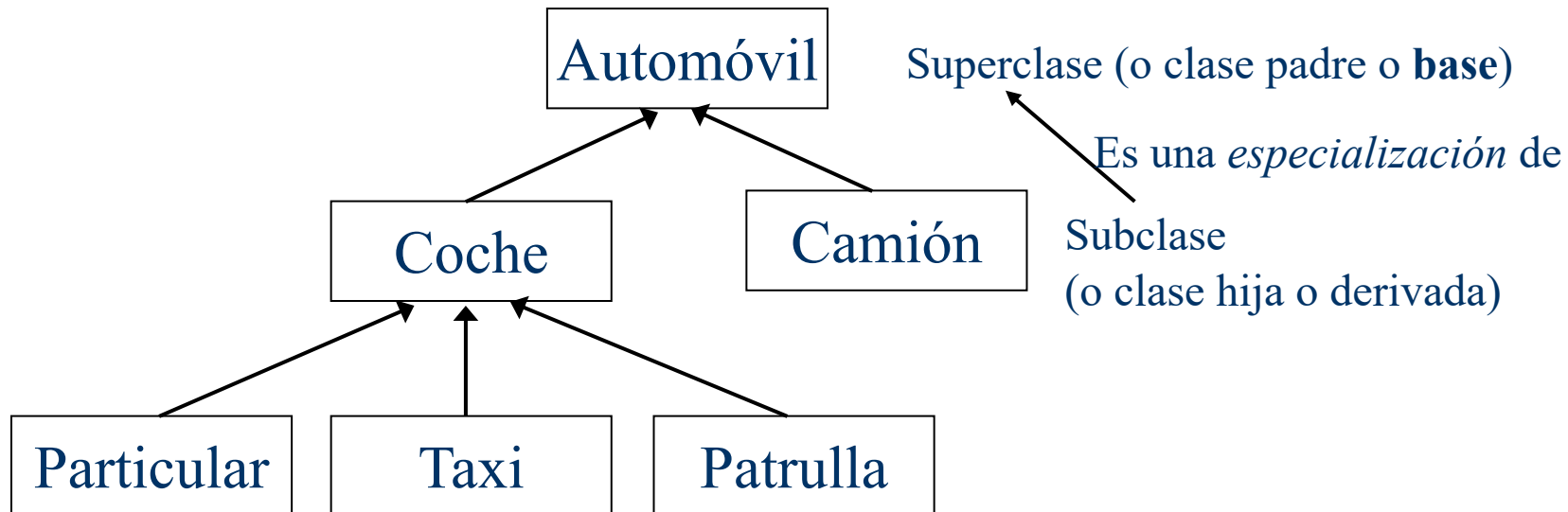
Las variables “privadas” de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python.

Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guión bajo debería tratarse como una parte no pública de la API y considerarse un detalle de implementación y que está sujeto a cambios sin aviso.

# Herencia: jerarquía de clases

---

- ◆ Permite definir una clase especializando otra ya existente
  - Se extiende un tipo de datos, heredando las características comunes y especificando las diferencias
  - Permite la reutilización de código

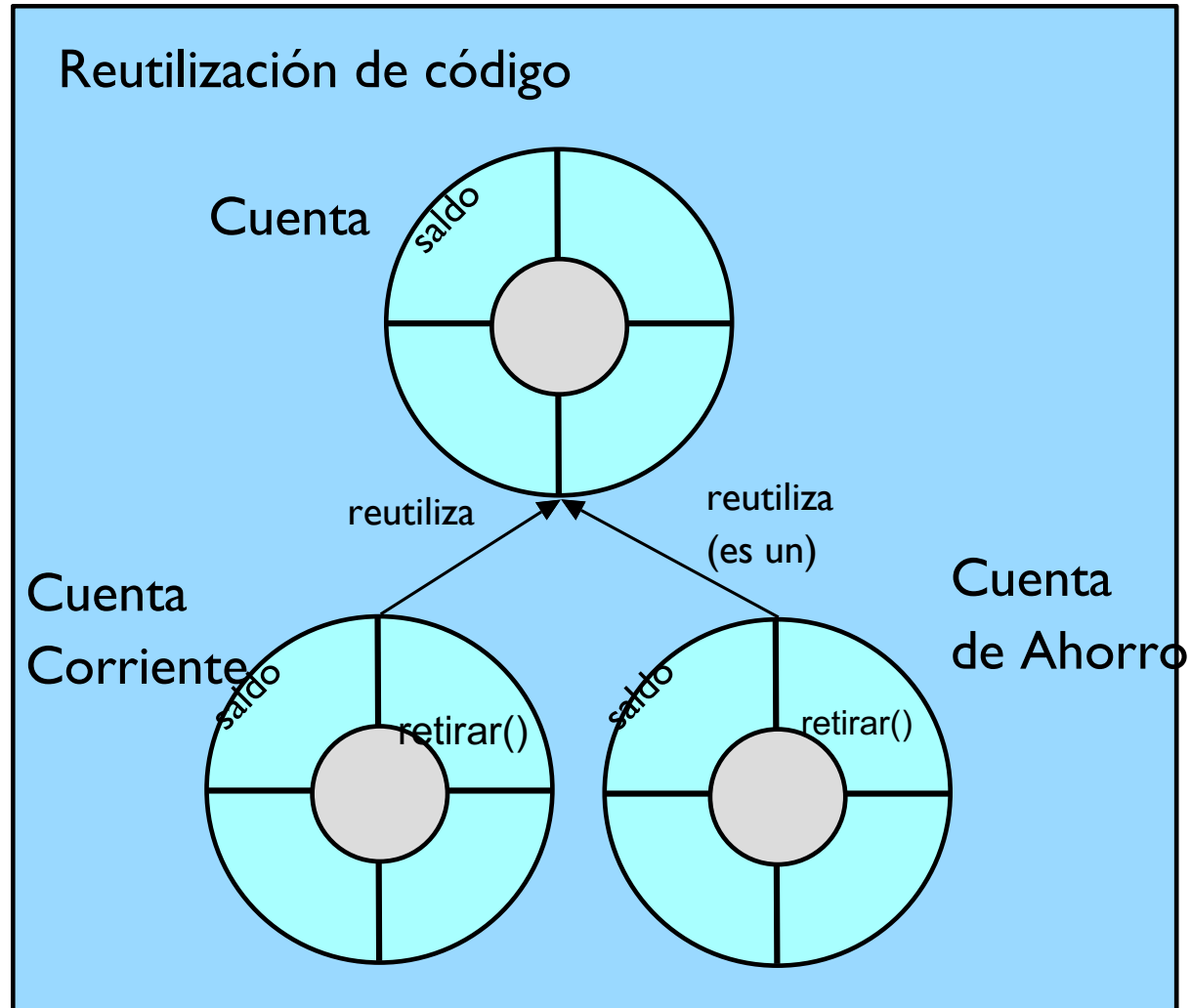


# Herencia de clases

---

- ◆ Los métodos y atributos de la superclase son heredados por las subclases, en la que
  - Se pueden añadir nuevos métodos y atributos
    - La clase Taxi tiene taxímetro y las operaciones poner en marcha taxímetro y para taxímetro
  - Se pueden redefinir los métodos
    - La clase Taxi redefine el método arrancar: si es recién subido un nuevo cliente, poner en marcha taxímetro

# Herencia de clases



# Polimorfismo

---

- ◆ Capacidad de solicitar la realización de una misma operación (mensaje) sobre distintos tipos de elementos
  - La realización concreta de la operación depende del objeto que reciba la petición
- ◆ Procesamiento genérico de objetos que:
  - Pertenecen a clases en una misma jerarquía
  - Pertenecen a clases que implementan un mismo interfaz



# Clases en Python

---

La forma más sencilla de definición de una clase se ve así:

```
class Clase:  
    <declaración-1>  
    .  
    .  
    .  
    <declaración-N>
```

# Clases y objetos

---

Ejemplo:

```
>>> class MiClase:
...     i=123
...     def f(self):
...         return "Hola Mundo"
...
>>> ob=MiClase()
>>> ob.i
123
>>> ob.f()
'Hola Mundo'
```

# Instanciando Objetos

---

Los objetos clase soportan dos tipos de operaciones: hacer referencia a atributos e instanciación.

La instanciación de clases usa la notación de funciones. El **constructor** es una función sin parámetros que devuelve una nueva instancia de la clase. Por ejemplo:

```
x = MiClase()
```

...crea una nueva instancia de la clase y asigna este objeto a la variable local x.

# Atributos de Objetos

---

Para hacer referencia a atributos se usa la sintaxis estándar de todas las referencias a atributos en Python:

`objeto.nombre`.

Los nombres de atributo válidos son todos los nombres que estaban en el espacio de nombres de la clase cuando ésta se creó.

En el ejemplo, `x.i` y `x.f` son referencias de atributos válidas del objeto `x`.

# Constructor

---

La operación de instanciación (“llamar” a un objeto clase) crea un objeto vacío. Muchas clases necesitan crear objetos con instancias en un estado inicial particular. Por lo tanto una clase puede definir un método especial llamado `__init__()`, de esta forma:

```
def __init__(self):  
    self.datos = []
```

Cuando una clase define un método `__init__()`, la instanciación de la clase automáticamente invoca a `__init__()` para la instancia recién creada cuando se llama a su constructor:

```
x = MiClase()
```

# Constructor con parámetros

---

El método `__init__()` puede tener argumentos para mayor flexibilidad. En ese caso, los argumentos que se pasen al operador de instanciación de la clase van a parar al método `__init__()`.

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

# Classes

---

```
class Punto:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '(' + str(self.x) + ', ' + str(self.y) + ')'
    def __add__(self, otro):
        return Punto(self.x + otro.x, self.y + otro.y)
```

# Ejemplo

---

```
class Number:
    def __init__(self, start):
        self.data = start;    # es definida la data
    def __add__(self, other): # numero + numero
        return Number (self.data + other.data)
    def __repr__(self):
        return `self.data`    # convierte a string
```



## Algunas clases pueden ser una coleccion

---

```
class collection:
    def __getitem__(self, i):
        return self.data[i] #la data es indexable

>>> X=collection()
>>> X.data = [1, 2, 3]
>>> for item in X:
    print item
```

# Herencia

---

La sintaxis para una definición de clase derivada se ve así:

```
Class DerivedClassName (BaseClassName) :
```

```
    <statement-1>
```

```
    .
```

```
    .
```

```
    .
```

```
    <statement-N>
```

# Herencia Múltiple

---

En Python una clase puede heredar de varias clases

```
class DerivedClassName(Base1, Base2, Base3):  
    <statement-1>  
    .  
    .  
    .  
    <statement-N>
```

# Clases y herencia

---

- ◆ Notaciones estándar: super clases, clases derivadas, **self** (como **this** en otros lenguajes), despacho dinámico
- ◆ Cada clase y cada objeto es un namespace (unidad de encapsulamiento) con un diccionario
- ◆ Para ubicar una operación, buscar en el diccionario del objeto (tabla de despacho). Si no lo encuentra, examina las superclases.
- ◆ Operador de sobrecarga se usa a través de los sgtes operadores predefinidos:

<code>__init__</code>	constructor
<code>__del__</code>	destructor
<code>__add__</code>	operador “+”
<code>__repr__</code>	printing, representación en salida

# Ejemplo Herencia

---

```
class Basic:
    def __init__(self, name):
        self.name = name
    def show(self):
        print 'Basic -- name: %s' % self.name

class Special(Basic): # entre paréntesis la clase base
    def __init__(self, name, edible):
        Basic.__init__(self, name)
        self.upper = name.upper() # clase base
        self.edible = edible
    def show(self):
        Basic.show(self)
        print 'Special -- upper name: %s.' % self.upper,
        if self.edible:
            print "It's edible."
        else:
            print "It's not edible."
    def edible(self):
        return self.edible
```

# Clases en Python 3.x

---

¿Cómo se define una clase de estilo nuevo?

Se hace heredando de una clase existente.

La mayoría de los tipos internos de Python, como enteros, listas, diccionarios, e incluso archivos son ahora clases de estilo nuevo.

Hay además una clase de estilo nuevo llamada 'object' que se convierte en la clase base para todos los tipos internos, de modo que si no queremos heredar de un nuevo tipo interno se puede heredar de este:

```
class MiClase(object):  
    def __init__(self):  
        pass
```

# Ejercicio: Fracción

---

```
class Fraccion:
    def __init__(self, numerador, denominador=1):
        m = mcd (numerador, denominador)
        self.numerador = numerador / m
        self.denominador = denominador / m

    def __mul__(self, otro):
        return Fraccion(self.numerador * otro.numerador,
self.denominador * otro.denominador)

    def __add__(self, otro):
        return Fraccion(self.numerador * otro.denominador
+ self.denominador * otro.numerador,
self.denominador * otro.denominador)
```

# Ejercicio: Fracción

---

```
class Fraccion:
    def __str__(self):
        return "%d/%d" % (self.numerador,
self.denominador)

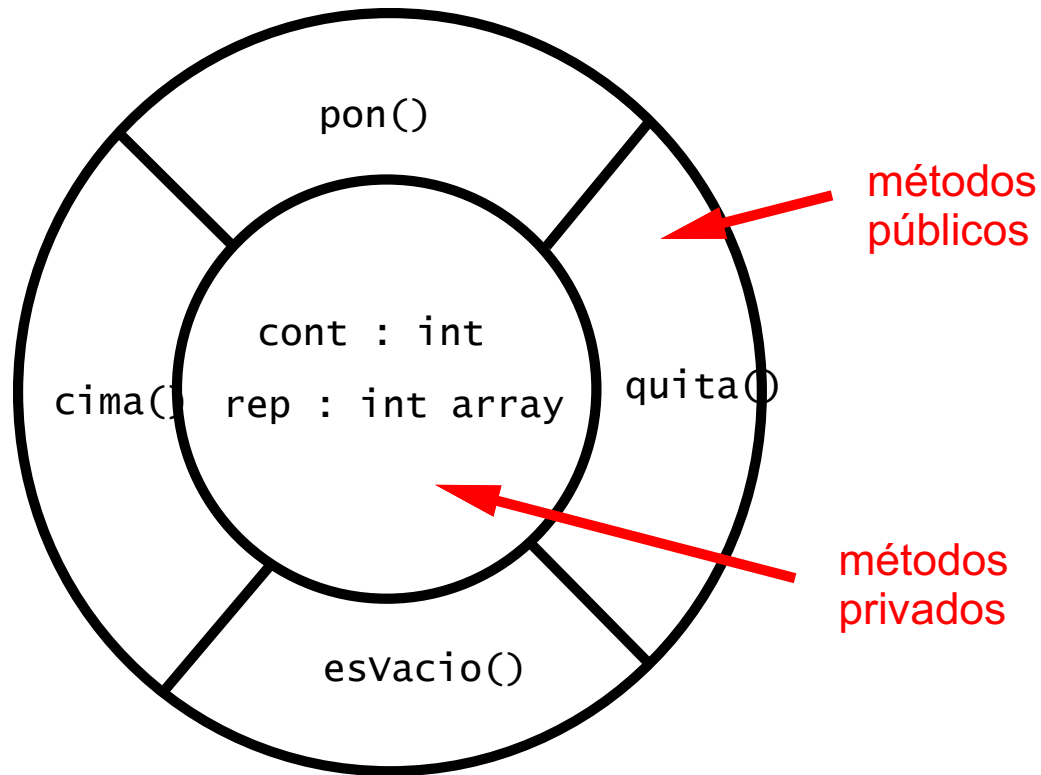
def mcd(m,n):
    if m % n == 0:
        return n
    else:
        return mcd(n,m%n)
```



## Ejercicio 7.3: Pila

---

Crea una clase Pila para apilar objetos.



# Ejercicio 3: Pila

---

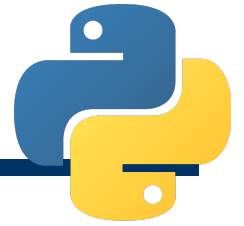
```
class Pila : # implem. con listas de Python
    def __init__(self) :
        self.elementos = []
    def push(self, elemento) :
        self.elementos.append(elemento)
    def pop(self) :
        return self.elementos.pop()
    def isEmpty(self) :
        return (self.elementos == [])
```

# Use of Classes and Objects: Machine learning steps

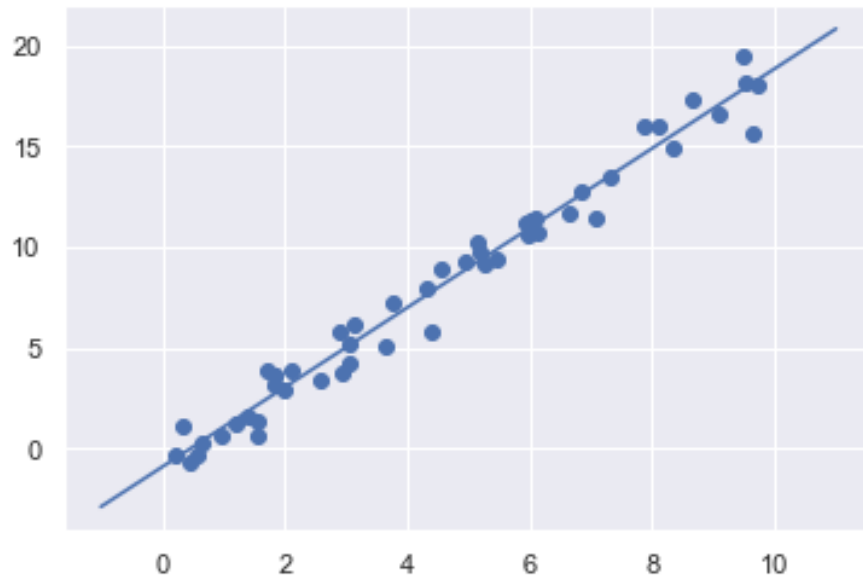
---

Most commonly, the ML are as follows:

1. Choose a class of **model** by importing the appropriate estimator
2. Choose model hyperparameters by instantiating this class with desired values.
3. Arrange data into a features **matrix and target vector**
4. Fit the model to your data by calling the **fit()** method of the model instance.
5. Apply the Model to new data:
  1. For supervised learning, often we predict labels for unknown data using the **predict()** method.
  2. For unsupervised learning, we often transform or **infer properties** of the data using the **transform() or predict()** method.



# Linear Regression (supervised)



# Appying a supervised ML model

---

Basic recipe for applying a supervised machine learning model:

1. Choose a class of model
2. Choose model hyperparameters
3. Fit the model to the training data
4. Use the model to predict labels for new data

# Linear Regression (supervised)

As an example of this process, let's consider a simple linear regression—that is, the common case of fitting a line to (x,y) data. **import matplotlib.pyplot as plt**

**import numpy as np**

`rng = np.random.RandomState(42)`

`x = 10 * rng.rand(50)`

`y = 2 * x - 1 + rng.randn(50)`

`plt.scatter(x, y);`

```
In [20]: import matplotlib.pyplot as plt
import numpy as np
```

```
rng = np.random.RandomState(42)
x = 10 * rng.rand(50)
y = 2 * x - 1 + rng.randn(50)
plt.scatter(x, y);
```



```
In [21]: x
```

```
Out[21]: array([3.74540119, 9.50714306, 7.31993942, 5.98658484, 1.5601864 ,
1.5599452 , 0.58083612, 8.66176146, 6.01115012, 7.08072578,
```

# Install sklearn

## pip install sklearn

---

```
In [23]: pip install sklearn
```

```
Collecting sklearn
  Downloading sklearn-0.0.tar.gz (1.1 kB)
Requirement already satisfied: scikit-learn in c:\users\luis\anaconda3\lib\site-packages (from sklearn) (0.22.1)
Requirement already satisfied: scipy>=0.17.0 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.4.1)
Requirement already satisfied: numpy>=1.11.0 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (1.18.1)
Requirement already satisfied: joblib>=0.11 in c:\users\luis\anaconda3\lib\site-packages (from scikit-learn->sklearn) (0.14.1)
Building wheels for collected packages: sklearn
  Building wheel for sklearn (setup.py): started
  Building wheel for sklearn (setup.py): finished with status 'done'
  Created wheel for sklearn: filename=sklearn-0.0-py2.py3-none-any.whl size=1320 sha256=264abce7084e9dc81b0a059edb1b2ac99385b71d0ca342a6e5b642fed1b82dc7
  Stored in directory: c:\users\luis\appdata\local\pip\cache\wheels\46\ef\c3\157e41f5ee1372d1be90b09f74f82b10e391eaacca8f22d33e
Successfully built sklearn
Installing collected packages: sklearn
Successfully installed sklearn-0.0
Note: you may need to restart the kernel to use updated packages.
```

```
In [24]: from sklearn.linear_model import LinearRegression
```

# Read documentation

[https://scikit-learn.org/stable/modules/linear\\_model.html](https://scikit-learn.org/stable/modules/linear_model.html)

scikit-learn

Install User Guide API Examples More ▾

Prev Up Next

scikit-learn 0.24.2  
Other versions

Please [cite us](#) if you use the software.

**1.1. Linear Models**

- 1.1.1. Ordinary Least Squares
- 1.1.2. Ridge regression and classification
- 1.1.3. Lasso
- 1.1.4. Multi-task Lasso
- 1.1.5. Elastic-Net
- 1.1.6. Multi-task Elastic-Net
- 1.1.7. Least Angle Regression
- 1.1.8. LARS Lasso
- 1.1.9. Orthogonal Matching Pursuit (OMP)
- 1.1.10. Bayesian Regression
- 1.1.11. Logistic regression
- 1.1.12. Generalized Linear Regression
- 1.1.13. Stochastic Gradient Descent - SGD
- 1.1.14. Perceptron
- 1.1.15. Passive Aggressive

## 1.1. Linear Models

The following are a set of methods intended for regression in which the target value is expected to be a linear combination of the features. In mathematical notation, if  $\hat{y}$  is the predicted value.

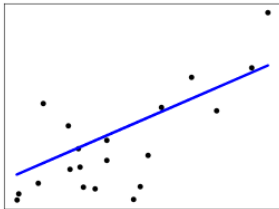
$$\hat{y}(w, x) = w_0 + w_1x_1 + \dots + w_px_p$$

Across the module, we designate the vector  $w = (w_1, \dots, w_p)$  as `coef_` and  $w_0$  as `intercept_`.

To perform classification with generalized linear models, see [Logistic regression](#).

### 1.1.1. Ordinary Least Squares

**LinearRegression** fits a linear model with coefficients  $w = (w_1, \dots, w_p)$  to minimize the residual sum of squares between the observed targets in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w ||Xw - y||_2^2$$




# Choose model hyperparameters

---

An important point is that *a class of model is not the same as an instance of a model*.

Once we have decided on our model class, there are still some options open to us. Depending on the model class we are working with, we might need to answer one or more questions like the following:

- ◆ Would we like to fit for the offset (i.e.,  $y$ -intercept)?
- ◆ Would we like the model to be normalized?
- ◆ Would we like to preprocess our features to add model flexibility?
- ◆ What degree of regularization would we like to use in our model?
- ◆ How many model components would we like to use?

# 1 - Instantiate the model

---

We would like to fit the intercept using the `fit_intercept` hyperparameter:

```
from sklearn.linear_model import LinearRegression
model = LinearRegression(fit_intercept=True)
model
```

```
In [24]: from sklearn.linear_model import LinearRegression
```

```
In [25]: model = LinearRegression(fit_intercept=True)
model
```

```
Out[25]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

## 2 - Arrange data into matrix X and target vector y

We need to coerce these x values into a `[n_samples, n_features]` matrix

```
X = x[:, np.newaxis]
```

```
X.shape
```

```
In [28]: x
```

```
Out[28]: array([3.74540119, 9.50714306, 7.31993942, 5.98658484, 1.5601864 ,
                1.5599452 , 0.58083612, 8.66176146, 6.01115012, 7.08072578,
                0.20584494, 9.69909852, 8.32442641, 2.12339111, 1.81824967,
                1.8340451 , 3.04242243, 5.24756432, 4.31945019, 2.9122914 ,
                6.11852895, 1.39493861, 2.92144649, 3.66361843, 4.56069984,
                7.85175961, 1.99673782, 5.14234438, 5.92414569, 0.46450413,
                6.07544852, 1.70524124, 0.65051593, 9.48885537, 9.65632033,
                8.08397348, 3.04613769, 0.97672114, 6.84233027, 4.40152494,
                1.22038235, 4.9517691 , 0.34388521, 9.09320402, 2.58779982,
                6.62522284, 3.11711076, 5.20068021, 5.46710279, 1.84854456])
```

```
In [29]: X = x[:, np.newaxis]
         X.shape
```

```
Out[29]: (50, 1)
```

```
In [30]: X
```

```
Out[30]: array([[3.74540119],
                [9.50714306],
                [7.31993942],
                [5.98658484],
                [1.5601864 ],
                [1.5599452 ],
                [0.58083612],
                [8.66176146],
```

# 3 - Fit de model

---

`model.fit(X, y)`

slope and intercept of the simple linear fit:

`model.coef_`

`model.intercept_`

```
In [31]: model.fit(X, y)
```

```
Out[31]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
In [32]: model.coef_
```

```
Out[32]: array([1.9776566])
```

```
In [33]: model.intercept_
```

```
Out[33]: -0.9033107255311164
```

# 4 - Predict labels for unknown data

---

```
xfit = np.linspace(-1, 11)
```

we need to coerce these x values into a `[n_samples, n_features]`

```
Xfit = xfit[:, np.newaxis]
```

feed it to the model

```
yfit = model.predict(Xfit)
```

```
In [41]: xfit = np.linspace(-1, 11)
```

```
In [42]: Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

```
In [43]: Xfit.shape
```

```
Out[43]: (50, 1)
```

```
In [44]: yfit.shape
```

```
Out[44]: (50,)
```

```
In [45]: Xfit
```

```
Out[45]: array([[ -1.          ],
                 [-0.75510204],
                 [-0.51020408],
                 [-0.26530612],
                 [-0.02040816],
                 [ 0.2244898 ],
                 [ 0.4489796 ],
                 [ 0.6734694 ],
                 [ 0.8979592 ],
                 [ 1.1224489 ],
                 [ 1.3469387 ],
                 [ 1.5714286 ],
                 [ 1.7959184 ],
                 [ 2.0204082 ],
                 [ 2.244898 ],
                 [ 2.4693878 ],
                 [ 2.6938776 ],
                 [ 2.9183674 ],
                 [ 3.1428571 ],
                 [ 3.3673469 ],
                 [ 3.5918367 ],
                 [ 3.8163265 ],
                 [ 4.0408163 ],
                 [ 4.2653061 ],
                 [ 4.4897959 ],
                 [ 4.7142857 ],
                 [ 4.9387755 ],
                 [ 5.1632653 ],
                 [ 5.3877551 ],
                 [ 5.6122449 ],
                 [ 5.8367347 ],
                 [ 6.0612245 ],
                 [ 6.2857143 ],
                 [ 6.5102041 ],
                 [ 6.7346939 ],
                 [ 6.9591837 ],
                 [ 7.1836735 ],
                 [ 7.4081633 ],
                 [ 7.6326531 ],
                 [ 7.8571429 ],
                 [ 8.0816327 ],
                 [ 8.3061224 ],
                 [ 8.5306122 ],
                 [ 8.755102 ],
                 [ 8.9795918 ],
                 [ 9.2040816 ],
                 [ 9.4285714 ],
                 [ 9.6530612 ],
                 [ 9.877551 ],
                 [10.102041 ],
                 [10.326531 ],
                 [10.5510204],
                 [10.7755102],
                 [11.0        ]])
```

# 5 - visualize the data

---

visualize the results by plotting first the raw data, and then this model fit

```
plt.scatter(x, y)
```

```
plt.plot(xfit, yfit);
```

```
In [46]: plt.scatter(x, y)  
plt.plot(xfit, yfit);
```

