Parte I: Programación en un lenguaje orientado a objetos

1. Introducción a los lenguajes de programación

- Lenguajes de programación. Compiladores e intérpretes. El lenguaje Python. Encapsulamiento de datos y algoritmos. Estructura de un programa. Funciones. Estilo de codificación.

2. Datos y expresiones

3. Clases

4. Estructuras algorítmicas

5. Estructuras de Datos

6. Tratamiento de errores

7. Entrada/salida

8. Herencia y polimorfismo

© Michael González Harbour, 1/feb/24

1.1. Lenguajes de programación

Un computador es una máquina capaz de almacenar *información* en su memoria y ejecutar una secuencia de *instrucciones*

- cada instrucción le dice al computador lo que debe hacer; por ejemplo, sumar dos números, restarlos o tomar una decisión en función de los datos disponibles

Las instrucciones de los computadores están escritas en *lenguajes de programación*

- el lenguaje natural, como el español, es demasiado complicado para un computador ordinario
  - aunque existen avanzados modelos de IA que entienden lenguaje natural
- los lenguajes de programación son más sencillos y los computadores ordinarios los comprenden

Números binarios

Los computadores se construyen mediante circuitos electrónicos digitales

- digital viene de dígito o número

La electrónica digital solo maneja dos números: el 0 y el 1

- normalmente uno es un valor de voltaje bajo, y el otro un voltaje alto

Los números compuestos por ceros y unos se llaman binarios

- Podemos poner muchas cifras binarias una a continuación de otra, para representar números tan grandes como necesitemos
- Por ejemplo, prueba a poner en google: número binario 10011010
  - obtendremos: 128 + 0 + 0 + 16 + 8 + 0 + 2 + 0 = 154

Afortunadamente, como veremos enseguida, aunque el computador usa números binarios nosotros no necesitamos usarlos

Instrucciones de un programa

Las instrucciones de un programa son códigos numéricos binarios almacenados en la memoria del computador

Ejemplo de lenguaje máquina para el microprocesador 68000: suma de dos enteros:

```
Dirección  Código Binario

$1000      0011101000111000
$1002      0001001000000000
$1004      1101101001111000
$1006      0001001000000010
$1008      0011000111000101
$100A      0001001000000100
```

Programación del computador

La programación mediante códigos numéricos se conoce como ***lenguaje máquina***

- es muy compleja para los humanos

Por ello se necesitan lenguajes de programación más cercanos a los programadores

© Michael González Harbour, 1/feb/24

Lenguajes de bajo nivel

Necesitamos escribir programas en un lenguaje más cómodo para los humanos

Una primera aproximación es el lenguaje de *bajo nivel* o *ensamblador*

- cada instrucción corresponde a una instrucción de lenguaje máquina
- es dependiente de la máquina: *no portable*

Ejemplo de lenguaje ensamblador: suma de dos enteros:

```
Dirección  Código Binario       Código Ensamblador

$1000      0011101000111000     MOVE.W $1200,D5
$1002      0001001000000000
$1004      1101101001111000     ADD.W  $1202,D5
$1006      0001001000000010
$1008      0011000111000101     MOVE.W $D5,$1204
$100A      0001001000000100
```

Lenguajes de alto nivel

Para evitar las desventajas de los lenguajes ensambladores, se han creado los lenguajes de *alto nivel*

- tienen instrucciones más abstractas y avanzadas
- son independientes de la máquina
- en la práctica, mucho más productivos

Ejemplo de instrucción en lenguaje de alto nivel: suma de dos enteros:

Dirección	Código Binario	Código Ensamblador	Alto Nivel
$1000	0011101000111000	MOVE.W $1200,D5	Z=X+Y
$1002	0001001000000000		
$1004	1101101001111000	ADD.W $1202,D5	
$1006	0001001000000010		
$1008	0011000111000101	MOVE.W $D5,$1204	
$100A	0001001000000100		

1.2 Compiladores e intérpretes

Son aplicaciones que *traducen* un programa escrito en un lenguaje de programación, a lenguaje máquina:

- lenguaje *ensamblador*: se traduce mediante un programa ensamblador
- lenguajes de *alto nivel*: se traducen mediante compiladores e intérpretes
  - los *compiladores* traducen el programa de aplicación antes de que éste se ejecute
  - los *intérpretes* van traduciendo el programa de aplicación a medida que se va ejecutando

Ejemplos de lenguajes de alto nivel

Los inicios

- Fortran
- Cobol
- Lisp
- Basic

Programación estructurada

- Pascal
- C
- Ada 83

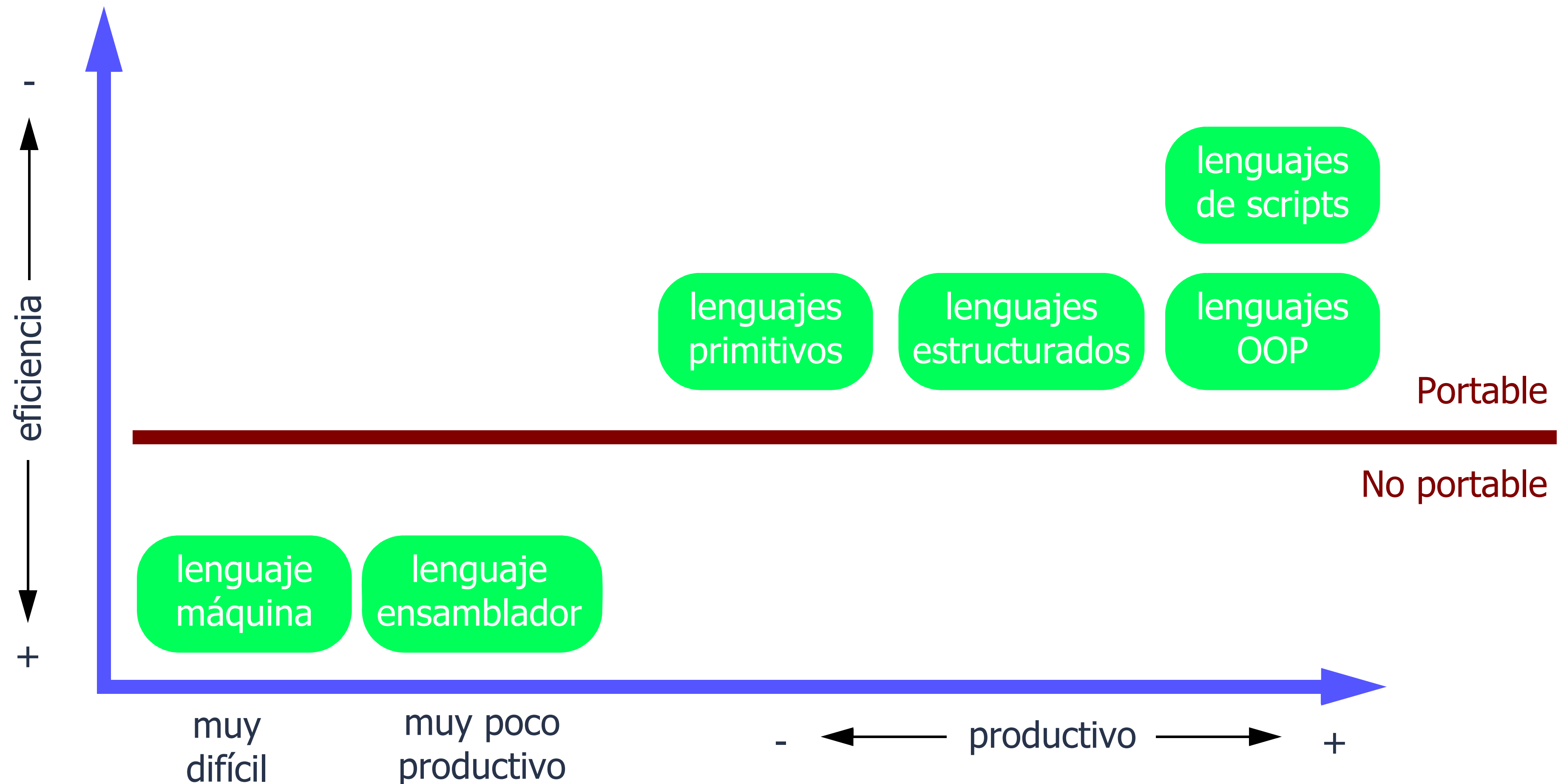
Programación orientada a objetos

- Smaltalk
- C++
- Java
- Ada
- C#
- Objective C
- Swift
- Go
- Rust

Lenguajes de scripts

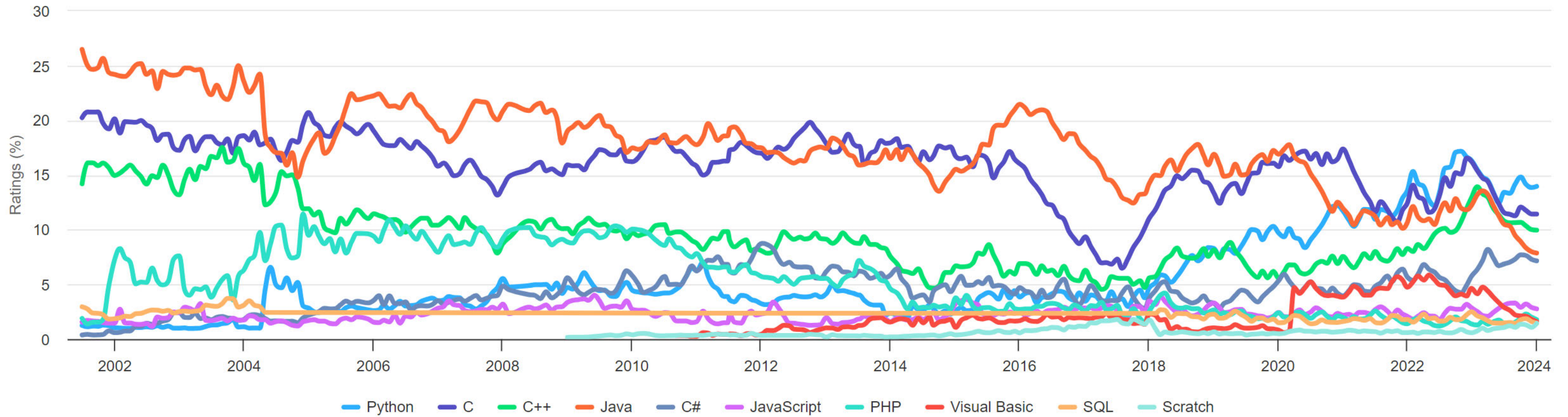
- Perl
- Python
- Javascript
- Ruby
- PHP
- R

Resumen de características de los lenguajes de programación

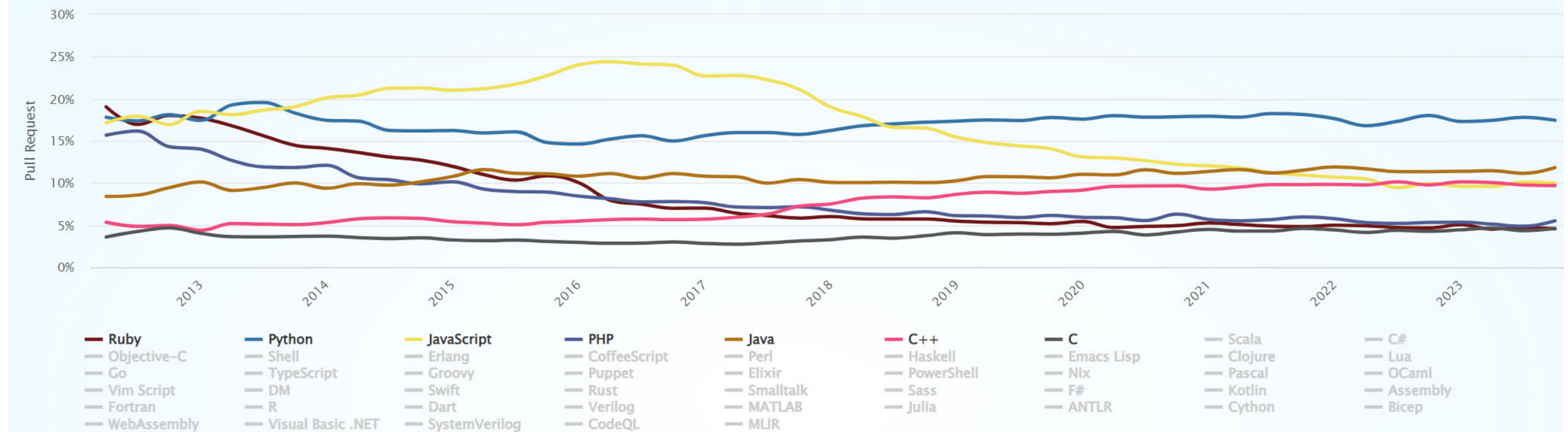


© Michael González Harbour, 1/feb/24

Ranking de lenguajes de programación

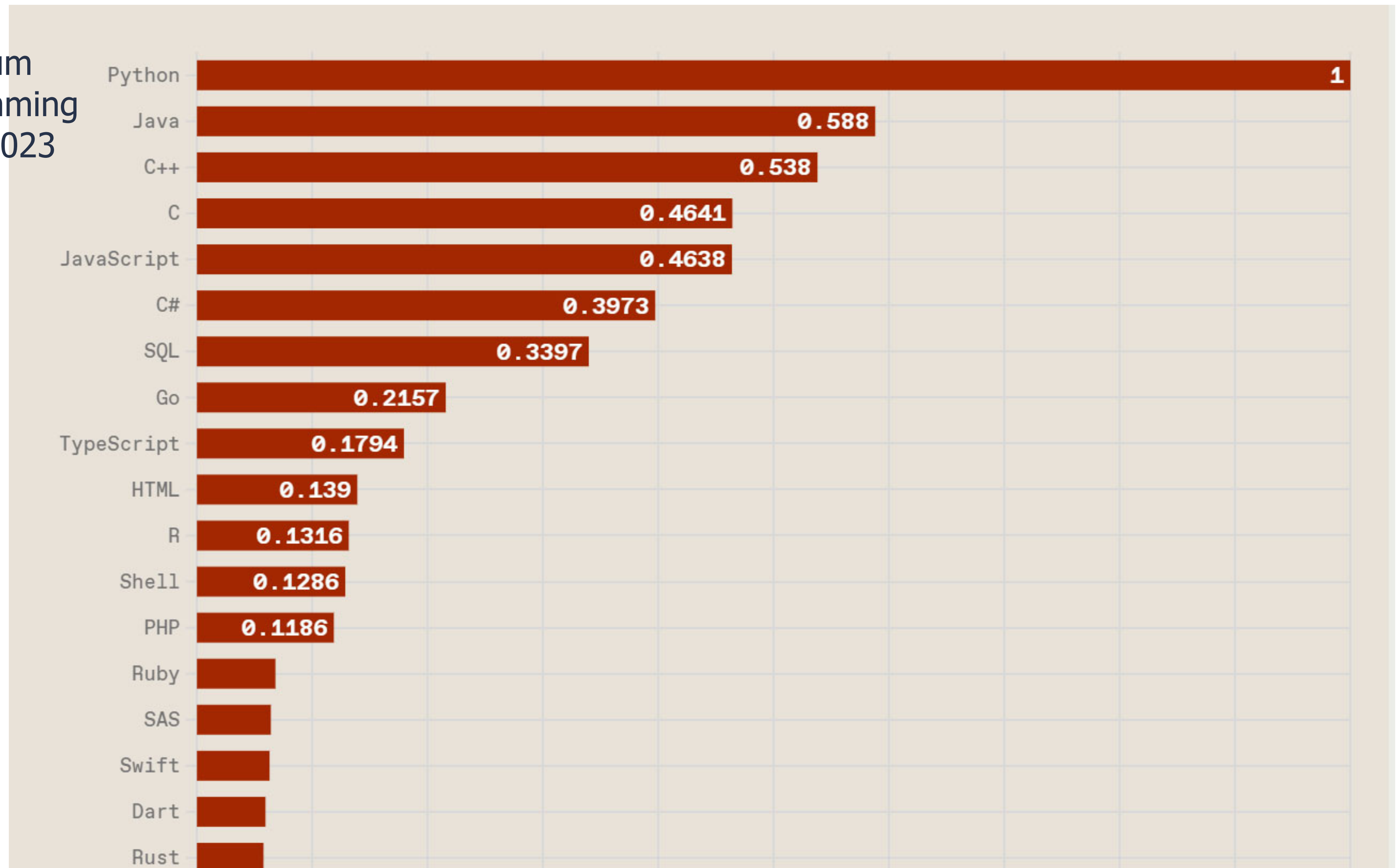


Ranking de lenguajes de programación



Ranking de lenguajes de programación

IEEE Spectrum
Top Programming Languages 2023



1.3 El lenguaje Python

Desarrollado por Guido van Rossum (NL) en 1991

Objetivos generales:

- Tipos dinámicos
- Gestión automática de memoria
- Objetos dinámicos, sin declaración
- Soporta varios paradigmas de programación
  - orientada a objetos, estructurada, funcional
- Tiene una amplia librería estándar y numerosas contribuciones de la comunidad de usuarios
- Las principales distribuciones son de código abierto

Versiones de Python

Hasta hace poco coexistían las versiones 2 y 3, que son incompatibles

- Inicialmente existía más software hecho para la versión 2
- Sin embargo, la versión 2 fue discontinuada con la distribución 2.7.18 (Abr 2020)

La versión actual es la 3.12.1 (Dic 2023)

Trabajaremos con la versión 3.11 que al principio del curso es la que instala por defecto la distribución recomendada:

- anaconda

Ventajas de Python

- muy legible (comparado con C/C++)
- código más compacto
- muchas librerías
- código abierto
- estructuras de datos integradas en la gramática del lenguaje
- alta productividad

Python es interpretado

Al ser interpretado es menos eficiente que otros lenguajes clásicos

- dependiendo del tipo de aplicación:
  - Java es entre 2 y 50 veces más rápido que CPython
  - C es entre 3 y 100 veces más rápido que CPython

Principales implementaciones:

- *CPython*: implementación de referencia
  - escrita en C
  - compila el código fuente a un lenguaje intermedio, más simple (llamado *bytecode*)
  - el *bytecode* es interpretado por una *máquina virtual*
- *PyPy*: intérprete más eficiente
  - usa la tecnología *just-in-time compile* (compilar sobre la marcha)
  - 7 veces más rápido que *CPython*, en promedio

1.4. Encapsulamiento de datos y algoritmos

Es el principio fundamental de la programación orientada a objetos (OOP)

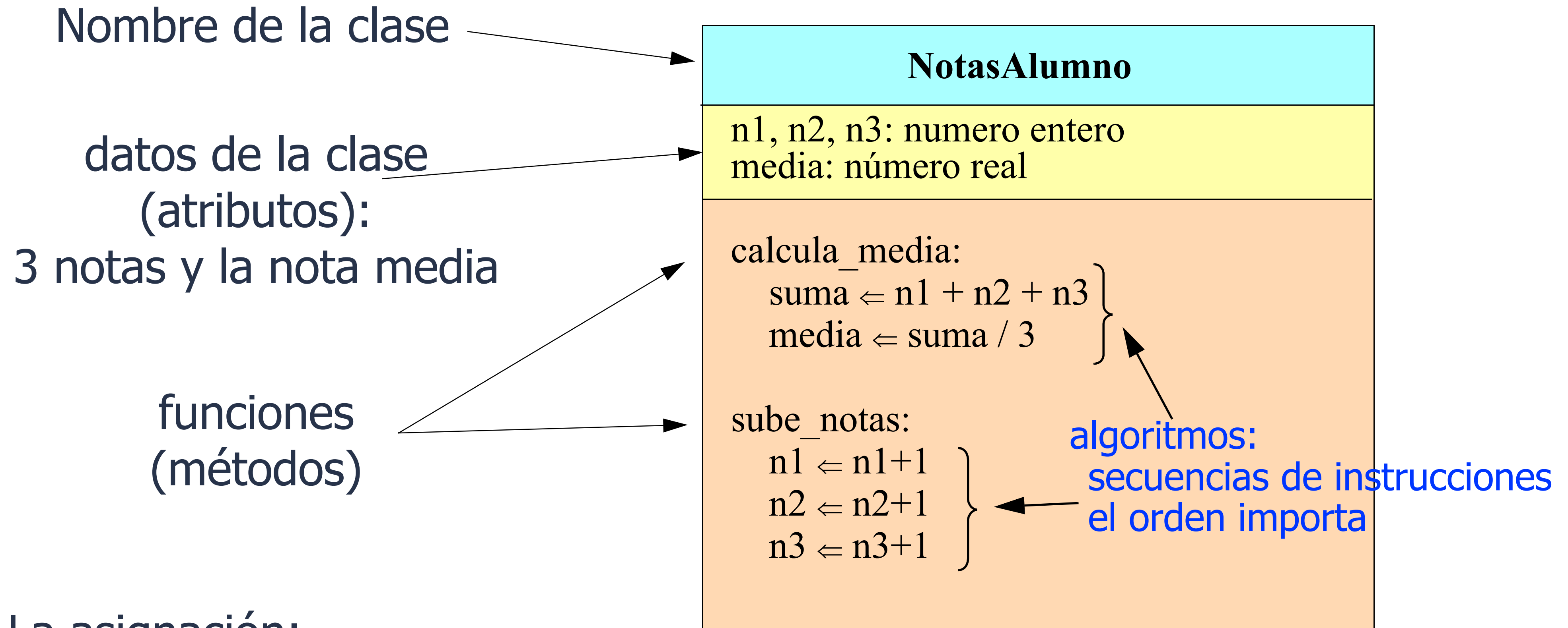
Cada "trozo" o *módulo* de programa contiene en su interior:

- *datos*
  - guardados en memoria en "recipientes" o variables de diversos tipos (números, texto, ...)
- *algoritmos* que trabajan con estos datos
  - secuencias de instrucciones descritas dentro de *funciones*

En OOP se llama *clase* a uno de estos módulos

- En Python un módulo puede contener una clase o varias

Ejemplo de clase con datos y algoritmos



La asignación:

⇐ operación de *asignación*:

copia el *valor* derecho en la *variable* o *dato* de la izquierda

en Python lo representaremos con el signo =

dato ⇐ valor

1.5. Estructura de un programa

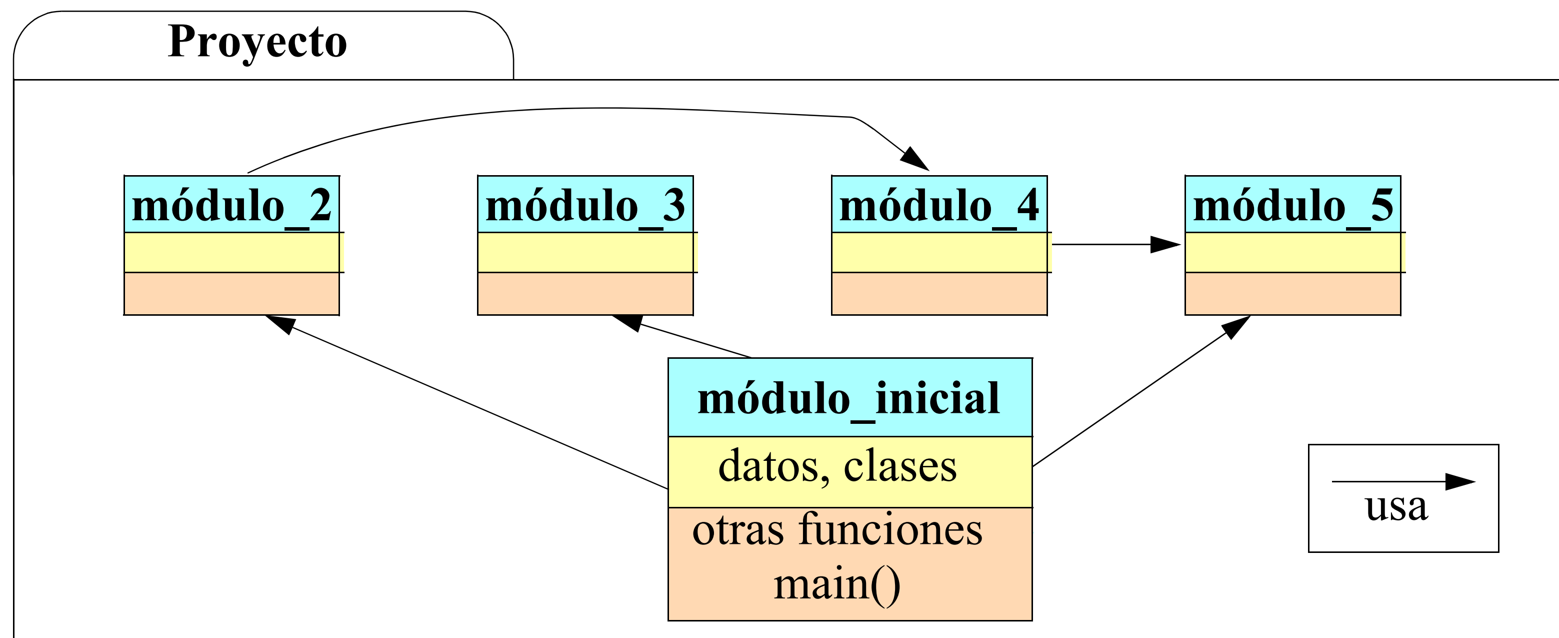
Un programa python es un conjunto de uno o varios módulos con:

- datos, funciones, clases e ~~instrucciones sueltas~~
- los módulos se pueden organizar en paquetes

¡No se pueden usar desde fuera!

Suele haber un módulo inicial, desde el que se cargan otros

- es habitual que sus instrucciones estén en una función llamada `main()`



© Michael González Harbour, 1/feb/24

Estructura de un módulo

El módulo más sencillo es un programa con una sola instrucción

- Hola Mundo: pone un mensaje de saludo en la pantalla

Escribiremos un módulo en un fichero llamado `hola_mundo.py`, con esta instrucción:

```
print("Hola, ¿qué tal?")
```

Función que pone en pantalla el mensaje indicado

El mensaje de texto se pone entre `" "` o `' '`

Al ejecutar el módulo en el intérprete, se muestra el mensaje

Un principio fundamental de la programación

Un programa se escribe una única vez y se lee muchas veces

Documentación de un módulo

Es recomendable indicar al principio de un módulo:

- qué sistema de codificación de caracteres se usa
  - especialmente importante en entornos con lenguas diferentes al inglés
- una breve descripción
- autor y fecha

Ejemplo

```
# -*- coding: utf-8 -*-
"""
Pone un mensaje de bienvenida en pantalla

@author: Michael González
@date  : 18/ene/2022
"""
print("Hola, ¿qué tal?")
```

 © Michael González Harbour, 1/feb/24

Carga y ejecución del módulo

La ejecución del módulo en el intérprete carga el módulo y ejecuta sus instrucciones

- usar el botón

Sin embargo, tras la carga, el módulo no podría ejecutarse desde otro

- Para resolverlo pondremos nuestra instrucción dentro de una *función*

1.6 Funciones

Son conjuntos de instrucciones agrupadas bajo un nombre

- se pueden invocar repetidas veces, para ejecutar sus instrucciones
- se les pueden pasar datos para operar con ellos
- pueden devolver un resultado

Habitualmente encapsulamos un algoritmo dentro de una función

© Michael González Harbour, 1/feb/24

Estructura de una función

Estructura:

```
def nombre_funcion(argumentos):
    """
    Breve descripción

    Descripción
    """

    instrucciones
```

Datos que necesita la función

Comentario de documentación, también llamado "docstring"

Sangrado

© Michael González Harbour, 1/feb/24

Invocar (ejecutar) una función

Para invocar la función se pone su nombre y los argumentos o datos concretos

```
nombre_funcion(datos_concretos)
```

Ejemplo:

```
print("Hola, ¿qué tal?")
```


© Michael González Harbour, 1/feb/24

Hola Mundo "bien estructurado"

Función escrita en el fichero hola_mundo.py:

No necesita datos

```python
def main():
    """Hola Mundo

    Este programa pone un mensaje en pantalla
    """

    print("Hola, ¿qué tal?")
```

Tras cargar el módulo con ▶, para invocar la función se debe escribir en el intérprete:

```python
main()
```


© Michael González Harbour, 1/feb/24

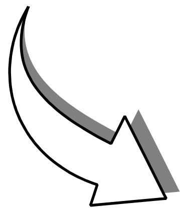
Observaciones

Sobre el ejemplo anterior

- sangrado
  - marca la estructura del programa y es obligatorio
- función sin argumentos (o parámetros)
- comentario de documentación, escrito entre `""" ..."""`

Resumen del programa "Hola Mundo"

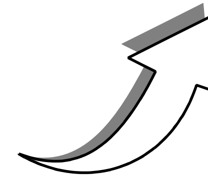
```python
print("Hola, ¿qué tal?")
```



```python
# -*- coding: utf-8 -*-
"""
Pone un mensaje en la pa

@author: Michael Gonzále
@date  : 18/ene/2022
"""

print("Hola, ¿qué tal?")
```



```python
# -*- coding: utf-8 -*-
"""
Programa principal

@author: Michael González
@date  : 18/ene/2022
"""

def main():
    """Hola mundo

    Pone un mensaje en pantalla
    """

    print("Hola, ¿qué tal?")
```

1.7 Estilo de codificación

La facilidad de leer código es fundamental

Un estilo uniforme y agradable ayuda mucho a entender el código

Python define unas normas de estilo en su documento PEP 8

- `https://www.python.org/dev/peps/pep-0008/`

Aquí mostramos un resumen de las normas más importantes

Normas de estilo

Sangrado

- usar 4 espacios, sin tabuladores

Longitud de las líneas

- que no superen los 79 caracteres

Separar bloques visualmente

- usar líneas en blanco para separar funciones y clases, y bloques más grandes de código dentro de las funciones

© Michael González Harbour, 1/feb/24

Normas de estilo: comentarios

Utilizar *docstrings* (comentarios de documentación)

- los consideramos obligatorios para módulos, funciones y clases
  - el del módulo lo ponemos al principio y los de las funciones y clases justo debajo de sus respectivos encabezamientos

Utilizar *comentarios internos*

- comienzan por el símbolo # y son efectivos hasta el final de la línea

Lugar de los comentarios

- poner comentarios internos en una línea propia, encima del código documentado
- evitar comentarios innecesarios, pues dificultan la lectura del código

Normas de estilo: espaciado

Espaciado

- Usar espacios alrededor de los operadores de asignación (=)

```
x = y+1    # si
x=y+1      # no
```

- y después de las comas

```
dibuja(inicio, fin)  # si
dibuja(inicio,fin)   # no
```

- pero no justo antes o después de paréntesis, corchetes o llaves

```
come(jamon[1], {chorizo: 2})          # si
come ( jamon[ 1 ], { chorizo: 2 } )  # no
```

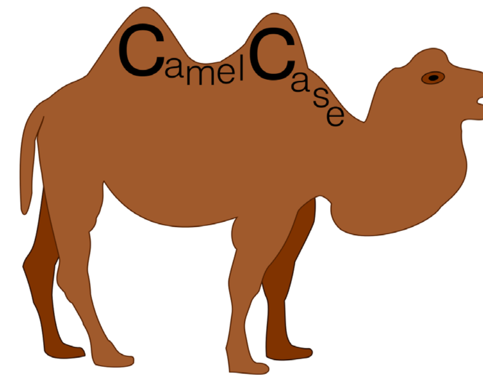

© Michael González Harbour, 1/feb/24

Normas de estilo: nombres

Convenio de nombres

- Clases: `CamelCase`
- funciones, datos, módulos, paquetes: `snake_case`
- constantes: `TODO_MAYUSCULAS`

Fuente: https://commons.wikimedia.org/wiki/File:CamelCase.svg



Conjunto de caracteres

- no usar letras acentuadas o ñ en los identificadores (nombres de cosas)
- se pueden usar en textos (*strings*) y comentarios

© Michael González Harbour, 1/feb/24

Normas de estilo: anotaciones de tipos

Ayudan a entender el tipo de cada dato (número real, número entero, texto, ...)

Anotaciones de tipos en funciones y datos

- aunque no son obligatorias en Python, debemos usarlas siempre
  - al crear funciones
  - al crear datos
- ayudan a entender los argumentos y respuestas retornadas por las funciones
- ayudan a entender los tipos de las variables
- ayudan a detectar errores
- las veremos más adelante