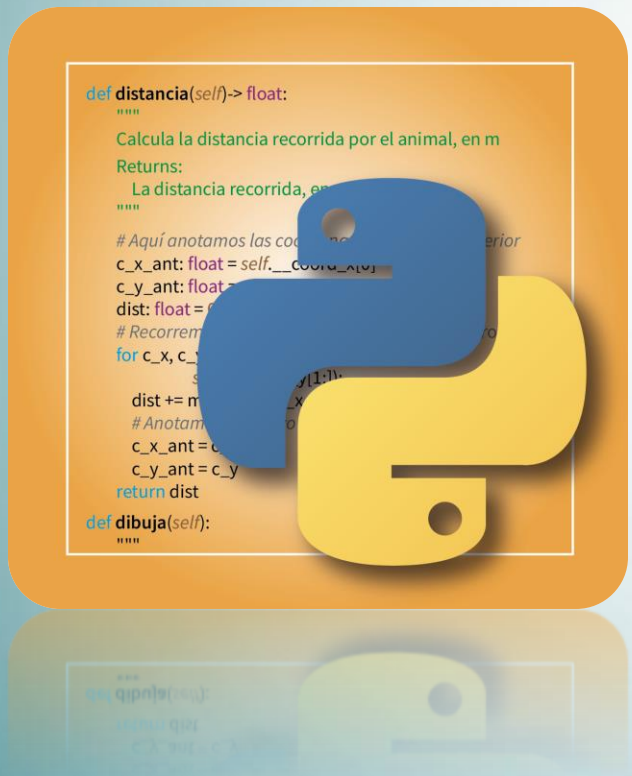
Programación

1. Introducción a los lenguajes de programación



Michael González Harbour

José Javier Gutiérrez García

José Carlos Palencia Gutiérrez

José Ignacio Espeso Martínez

Adolfo Garandal Martín

Departamento de Ingeniería Informática y Electrónica

Programación en Python

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

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
- los lenguajes de programación son más sencillos y el computador los comprende

Notas:

Un *computador* es una máquina capaz de ejecutar instrucciones y dotada de una *memoria* en la que puede guardar dos cosas:

- *datos*
- *instrucciones* para manipular esos datos

A la parte física de esa máquina le llamamos *hardware.*

Para que el computador pueda funcionar es preciso meter en su memoria un *programa*: una secuencia de instrucciones que le dirán lo que tiene que hacer.

- Esta es la parte lógica (no física) del computador y se denomina *software.*

Para crear programas de computador debemos usar unos lenguajes especiales, llamados *lenguajes de programación*.

- Son suficientemente sencillos como para que el computador los entienda y sea capaz de ejecutarlos sin ambigüedades.

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

Notas:

Los computadores modernos se construyen por medio de *circuitos electrónicos digitales*.

- Los circuitos electrónicos constan de multitud de dispositivos electrónicos llamados transistores, capaces de amplificar señales eléctricas.
- La palabra *digital*, se refiere a dígito o número. Es decir, los circuitos digitales son capaces de manejar números.
- La electrónica digital es más eficiente si se restringe a usar solo dos números: el *cero* y el *uno*.
- Nosotros estamos acostumbrados a usar los números *decimales*. Usamos solo diez cifras, del 0 al 9. Si necesitamos representar números grandes, lo hacemos poniendo juntas varias cifras, y asignándoles pesos diferentes: unidades, decenas, centenas, millares, etc. Esto se llama sistema de numeración *posicional*. El número 674 representa seis centenares, más siete decenas y cuatro unidades: $674 = 6 \cdot 100 + 7 \cdot 10 + 4 \cdot 1$. Estos pesos que usamos son las potencias de 10, que es la base de la numeración: 10^0, 10^1, 10^2, 10^3, ...
- Lo mismo podemos hacer con la base de numeración 2, que representa los llamados números *binarios*. Podemos poner muchas cifras juntas, cada una con un peso igual a una potencia de 2. Así, el número

$$10110 = 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

En definitiva, los computadores por dentro solo saben manejar *números binarios*: ceros y unos.

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
```

Notas:

Los programas de computador internamente están compuestos por *instrucciones* que son números binarios.

- cada número o grupo de números representa una instrucción que el computador debe hacer siguiendo una *secuencia*:
- en el ejemplo que se muestra, cada dos números binarios representan una instrucción, y tenemos una secuencia de tres instrucciones con las que se consigue sumar dos datos:
  - mover un dato de un sitio a otro
  - sumar un número al dato anterior
  - mover el resultado a otro lugar

Lógicamente, este tipo de programación es extremadamente complicada para los humanos.

- Estamos acostumbrados a manejar un número más rico de símbolos: las letras del alfabeto y las cifras decimales.
- Además, agrupamos esos símbolos en palabras. Normalmente recordamos más fácilmente las palabras que los números.
- Trabajar con números binarios no es nuestro fuerte.

Afortunadamente no tendremos que manejar estos códigos binarios.

- Para ello se han inventado los lenguajes de programación.

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

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
```

Notas:

Para hacer accesible la programación de un computador a cualquier persona, existen lenguajes de programación que tienen como función evitar la dificultad del lenguaje máquina.

En los lenguajes ensambladores o de bajo nivel.

- Cada instrucción de lenguaje ensamblador se corresponde con una instrucción de lenguaje máquina.
- Pero en lugar de codificar mediante números se codifica mediante símbolos alfanuméricos, palabras, más fáciles de recordar.
- En el ejemplo podemos ver la secuencia de mover un dato (MOVE), sumar (ADD) y mover (MOVE).

Aunque es mucho más fácil de aprender y leer que el lenguaje máquina, en la práctica, programar en lenguaje ensamblador es tedioso, difícil, con mucha facilidad para cometer errores y, por tanto, poco productivo.

- Las instrucciones de lenguaje ensamblador, al igual que las de la máquina, son excesivamente simples. Construir programas complejos a partir de estas instrucciones es muy difícil. ¿Cómo hago una inversión de matrices de tamaño variable a base de sumas, productos y poco más?
- Por otro lado, los programas en ensamblador no son portables, pues dependen de la arquitectura del computador.
  - Cada tipo de computador tiene una arquitectura diferente.

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
```

Notas:

Para hacer más productivo el proceso de programar, existen lenguajes de programación de *alto nivel* que tienen como función presentar al usuario el computador de acuerdo con un modelo abstracto (informático) sencillo e independiente de su estructura electrónica interna.

- Los lenguajes de alto nivel permiten programar utilizando un lenguaje más próximo al humano, e independiente de la máquina.
- Los lenguajes de alto nivel producen un código un poco menos eficiente que el ensamblador, pero más sencillo de escribir y mantener, con menos errores, y mucho más productivo.

Salvo para operaciones muy breves y especiales, casi nadie usa ensamblador. Usaremos siempre lenguajes de alto nivel.

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

Notas:

Un computador solo entiende lenguaje máquina. Para que comprenda programas hechos en otros lenguajes de programación es necesario *traducirlos*.

Los ensambladores, compiladores e intérpretes hacen esta traducción a lenguaje máquina. Según el lenguaje utilizado, las herramientas son:

- *Lenguajes de bajo nivel*: se utiliza un programa ensamblador, que traduce los símbolos alfanuméricos a código máquina, por medio de algoritmos muy simples.
- *Lenguajes de alto nivel*. La traducción a lenguaje máquina se hace mediante dos tipos de traductores:
  - Compiladores: traducen el programa escrito en lenguaje de alto nivel de forma completa antes de su ejecución.
  - Intérpretes: traducen cada instrucción mientras se ejecuta el programa.

Los intérpretes son más lentos en la ejecución, ya que tienen que hacer el trabajo de traducción cada vez que se ejecuta el programa. Es más habitual usar compiladores, pues producen un código mucho más eficiente.

Los lenguajes de programación de alto nivel han sido definidos como una solución intermedia entre los lenguajes naturales humanos y los lenguajes máquina de los computadores. Están bien definidos, en el sentido de que la tarea que se puede expresar con ellos no es ambigua y por lo tanto pueden ser traducidos en un programa máquina concreto, de forma automatizada y por el propio (aunque no necesariamente) computador que va a realizar la tarea.

Ejemplos de lenguajes de alto nivel: Los inicios

- ***Fortran***: 1956, para cálculo científico. Estándar actual: 2018
- ***Cobol***: 1960, para aplicaciones de gestión. Estándar actual: 2014
- ***Lisp***: 1958, para inteligencia artificial. Estándar actual: 2007
  - tiene un dialecto importante llamado *Scheme*
- ***Basic***: 1964, para docencia, interpretado
  - Visual Basic (Microsoft)

Notas:

Existen muchos lenguajes de alto nivel de propósito general. Sus principales diferencias se encuentran en que poseen un conjunto de órdenes mas adecuado para expresar tareas de un tipo concreto de problema o porque corresponden a distintos niveles de evolución de los computadores. Aquí vemos los lenguajes más primitivos:

FORTRAN (FORmula TRANslation). Su nombre evidencia la orientación matemática de uno de los lenguajes de alto nivel mas antiguos, que aún perduran. J. Backus lo desarrolló en 1956. Aunque ha perdido terreno frente a los lenguajes mas modernos, todavía es ampliamente utilizado en aplicaciones científicas de grandes cálculos numéricos, porque probablemente, es el lenguaje con mayor número de librerías, desarrolladas y comprobadas por mucha gente, a lo largo de su historia.

COBOL (COmmon Business Oriented Language). Se trata del lenguaje que ha alcanzado una mayor resonancia en las tareas de gestión. Su desarrollo fue promovido por el Departamento de Defensa de EEUU, en 1960. El lenguaje ha sufrido muchas extensiones, y ha sido actualizado en 2014.

LISP (LISt Processing). El Massachusetts Institute of Technology creó, en 1959, este lenguaje de alto nivel orientado a aplicaciones de inteligencia artificial. La programación de procesos recurrentes (edificados sobre datos procesados en los pasos anteriores) es uno de los puntos fuertes del LISP.

BASIC (Beginners All-purpose Symbolic Instruction Code). Nació entre 1964 y 1965 en el Dartmouth College como una herramienta para la enseñanza. Con el tiempo han ido proliferando los dialectos y versiones, hasta el punto de que es raro el fabricante que no desarrolla un dialecto para sus propios equipos. Fue muy popular por su sencillez, pero tiene carencias importantes.

Ejemplos de lenguajes: programación estructurada

La programación estructurada mejora la claridad, calidad y tiempo de desarrollo recurriendo únicamente a instrucciones con un comienzo y un final claros

- ***Pascal***: 1969, para docencia, programación estructurada
  - Ahora está reapareciendo como *Delphi (2018)*, una versión orientada a objetos
- ***C***: 1972, para programación del software del sistema. Estándar actual: 2018
- ***Ada 83***: 1983, para sistemas de alta integridad, incluyendo sistemas de tiempo real

Notas:

PASCAL (En honor del matemático francés Blaise Pascal). Es un lenguaje de programación desarrollado por el profesor Nicklaus Wirth, en 1969, en el Instituto Federal de Tecnología de Zurich partiendo de los fundamentos del ALGOL. Fue uno de los primeros lenguaje que incorporaron los conceptos de programación estructurada. Aunque fue muy popular, la dificultad para partir el programa en módulos y la falta de estandarización han hecho decaer su uso.

C. Es un lenguaje de programación desarrollado por la Bell Laboratories, en principio para trabajar con el sistema operativo UNIX. Quizás por ello, la popularidad del 'C' es muy alta. Es un lenguaje que, al mismo tiempo que permite una programación en alto nivel, permite una gran aproximación a la máquina. Muchos lo consideran un lenguaje intermedio entre alto y bajo nivel. Como estos últimos, presenta alta eficiencia y escasa fiabilidad. Es fácil cometer errores en C.

Ejemplos de lenguajes: programación orientada a objetos

La programación orientada a objetos (OOP) aumenta la reutilización de código basándose en técnicas como la abstracción, la herencia y el polimorfismo

- ***Smaltalk***: 1980, creado para uso educativo. Estándar actual 1998
- ***C++***: 1987, extensión mejorada del C. Estándar actual 2020
- ***Java***: 1995, mejora la fiabilidad y añade programación distribuida. Estándar *de facto*
- ***Ada***: se añade programación orientada a objetos. Estándar actual 2016
- ***C#*** (C sharp): Similar a Java, divergen a partir de 2005. Estándar actual 2003

Notas:

SMALTALK: Lenguaje de programación orientada a objetos puro y con tipos dinámicos. Es muy ineficiente con respecto a lenguajes procedurales como el C o el Ada, pero es cómodo de usar y de programar en él.

JAVA: Lenguaje derivado del C en cuanto a sintaxis, pero hace más énfasis en la fiabilidad promoviendo comprobaciones de tipos que hace el compilador y gestión de memoria automática. Además, soporta programación concurrente. Está pensado para su ejecución en sistemas distribuidos (internet). Existe un código intermedio, bien definido, que puede intercambiarse entre computadores diferentes para luego ser traducido y ejecutado, consiguiendo así portabilidad entre diferentes arquitecturas de computadores.

C++: Extensión del lenguaje C que mejora algunos de sus inconvenientes, y añade construcciones de programación orientada a objetos. Entre las mejoras destacan una mayor comprobación de los tipos de datos por parte del compilador, las excepciones, y las plantillas genéricas.

Ada (En honor de Lady Augusta Ada Byron). El Ada es un lenguaje inspirado en el PASCAL, que fue promovido por el Departamento de Defensa de Los EEUU. El objetivo de su desarrollo era conseguir un lenguaje con posibilidades de convertirse en un estándar universal y que facilitara la ingeniería de software y el mantenimiento de los programas. Entre sus campos de aplicación se incluyen los sistemas de tiempo real y los sistemas de alta integridad. El 1995 se revisó el lenguaje para mejorarlo y para añadirle construcciones de programación orientada al objeto. Se espera que en 2021 se publique una nueva versión del estándar.

C#. Al pertenecer Java a la empresa Oracle otras empresas desarrollaron sus propios lenguajes. C sharp fue desarrollado por Microsoft y es bastante similar a Java. Está estandarizado, pero las versiones actuales están por delante de la especificación estándar.

Ejemplos de lenguajes: programación orientada a objetos

- ***Objective-C***: es un superconjunto de C con programación orientada a objetos
  - se origina en los años 80 por Apple. Versión actual: 2007
- ***Swift***: Creado por Apple en 2014 a partir de Objective C, con mejoras y mayor fiabilidad
- ***GO***: Creado por Google en 2009, a partir de C, para hacer programación con mayor fiabilidad

Notas:

Objective C: Versión del lenguaje C con algunos añadido que fue hasta 2014 el principal lenguaje para programar aplicaciones para los computadores Mac y teléfonos con sistema operativo IoS.

Swift: Actualmente Apple va abandonando Objective C en favor del lenguaje Swift, que permite una programación más libre de fallos y añade algunas facilidades nuevas de programación.

GO: Cada gran empresa quiere su propio lenguaje. Este ha sido desarrollado por Google, con la intención de añadir al lenguaje C mayor fiabilidad, así como soportar programación funcional, concurrente y orientada a objetos, aunque esta última se consigue de una forma diferente a la habitual, al no existir jerarquías de tipos.

Ejemplos de lenguajes
Lenguajes de guiones o “scripts”:

Normalmente interpretados, pensados para aplicaciones no muy grandes o para hacer tareas en el entorno de otras aplicaciones

Algunos de estos lenguajes han evolucionado hasta convertirse en lenguajes de programación de propósito general

- ***PHP***: desarrollado para hacer páginas Web dinámicas
- ***Perl***: eficaz para operaciones de manipulación de textos
- ***Python***: hace énfasis en la legibilidad
- ***JavaScript***: soportado por muchos navegadores Web para hacer páginas web interactivas
- ***Ruby***: derivado de Lisp, con OOP del estilo de Smalltalk
- ***R***: lenguaje para cálculo estadístico y representación gráfica

Notas:

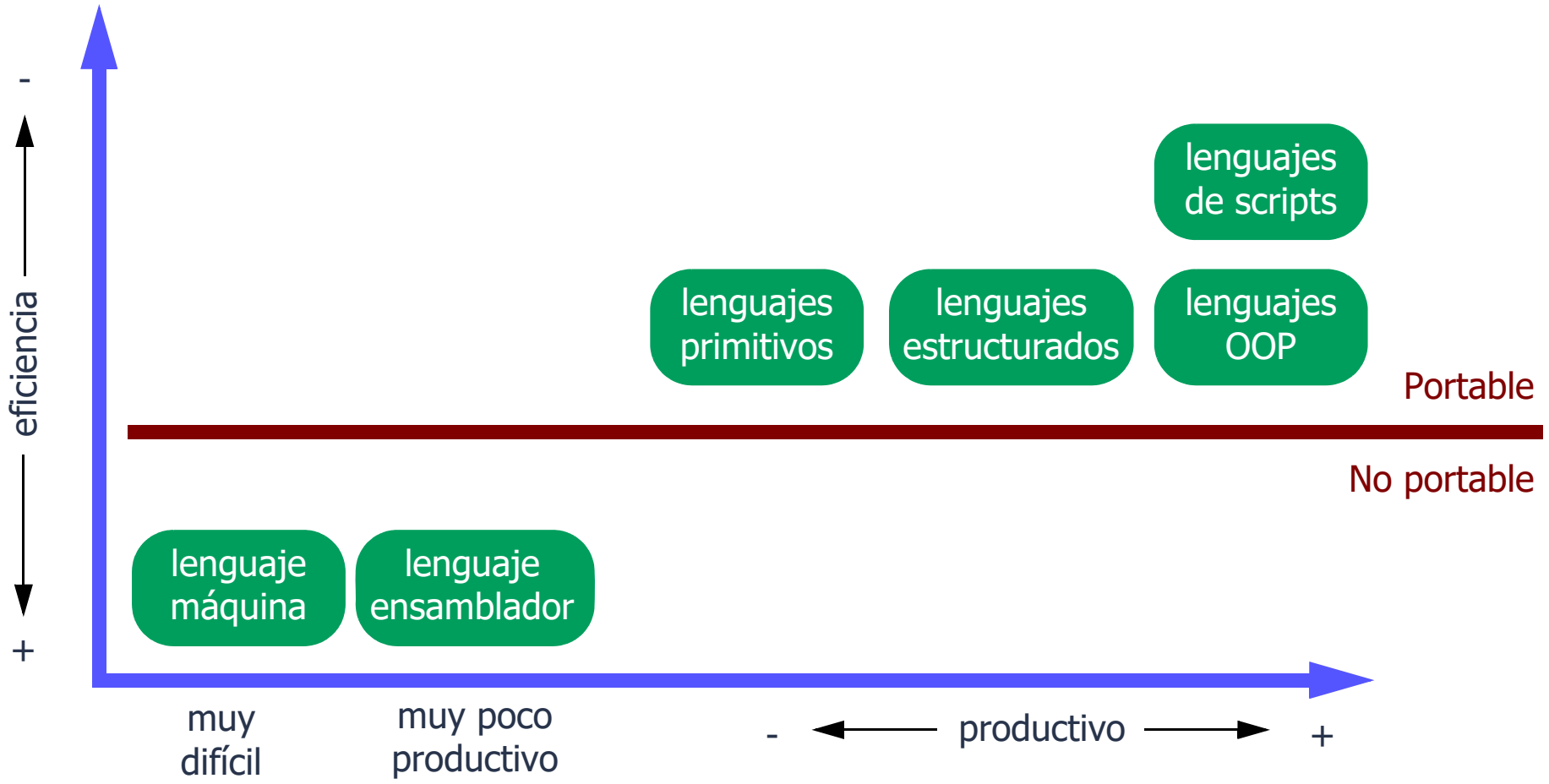
Un lenguaje de "scripts" o de guiones es un lenguaje de programación para controlar la ejecución de otras aplicaciones, o para realizar tareas no muy complejas. Suelen ser interpretados. Tienen mucho éxito para programar aplicaciones web y en entornos de propósito especial.

Los lenguajes de scripts son habitualmente interpretados. Algunos de ellos han evolucionado hasta convertirse en lenguajes de propósito general, y se han desarrollado para ellos compiladores que permiten una ejecución más eficiente.

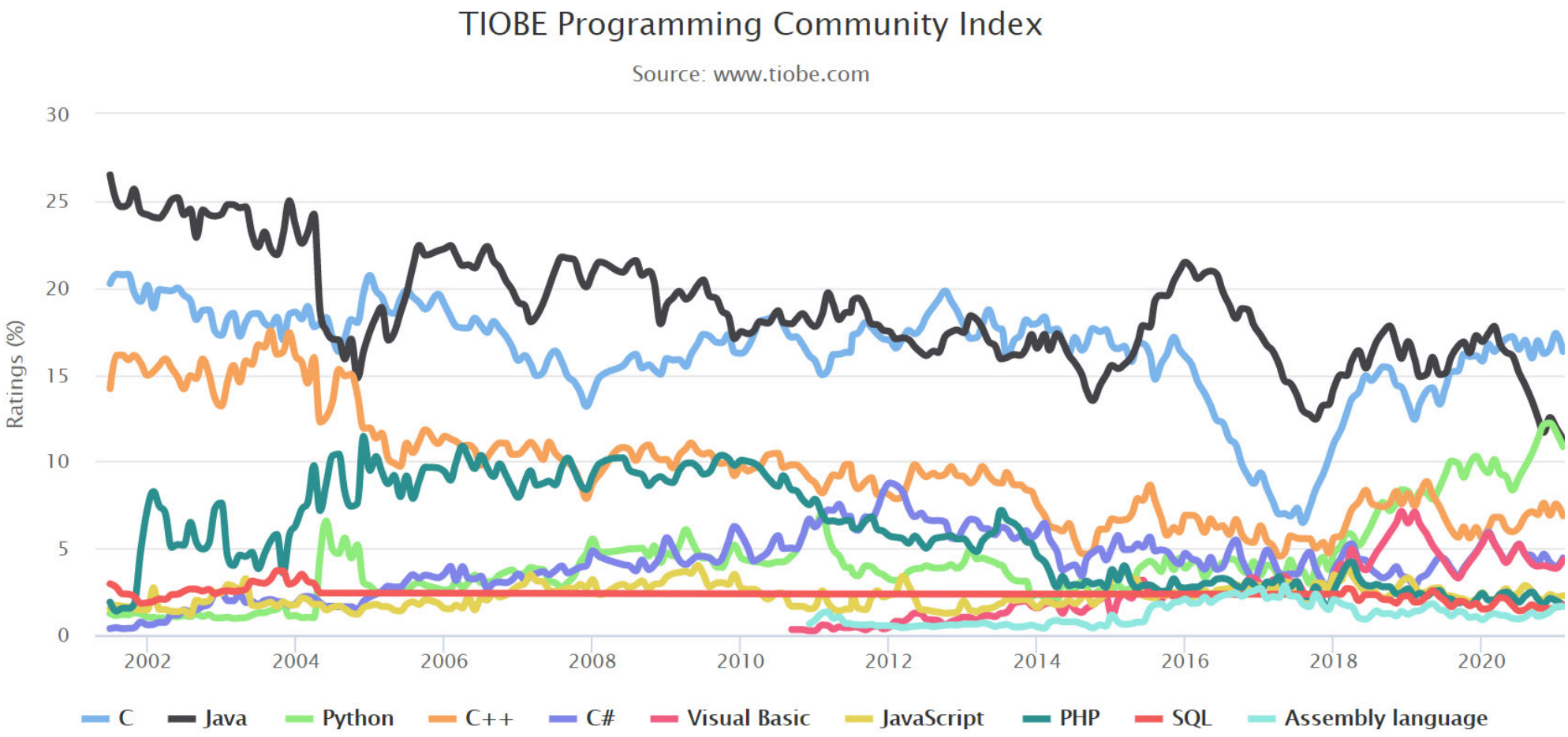
Los lenguajes de scripts suelen tener tipos dinámicos.

Un lenguaje con tipos dinámicos es aquel en el que la comprobación del tipo de dato que se está utilizando se hace durante la ejecución, en lugar de hacerse de manera estática durante la compilación.

Resumen de características de los lenguajes de programación



Ranking de lenguajes de programación



Ranking de lenguajes de programación

GitHub Language Rankings, 2018-2020

Language	2020 Ranking	2019 Ranking	2018 Ranking
JavaScript	1	1	1
Python	2	2	3
Java	3	3	2
TypeScript	4	7	4
C#	5	5	6
PhP	6	4	4
C++	7	6	5
C	8	9	8
Shell	9	8	9
Ruby	10	10	10

Ranking de lenguajes de programación

IEEE Spectrum
Top Programming
Languages

Rank	Language	Type	Score
1	Python		100.0
2	Java		95.3
3	C		94.6
4	C++		87.0
5	JavaScript		79.5
6	R		78.6
7	Arduino		73.2
8	Go		73.1
9	Swift		70.5
10	Matlab		68.4
11	Ruby		66.8

Notas:

Fuentes:

http://www.tiobe.com/tiobe-index/

- Fecha: Feb 2021

https://insights.dice.com/2020/12/03/10-most-popular-programming-languages-on-github/

- Fecha: Feb 2021

https://developer-tech.com/news/2020/jul/27/ieee-spectrum-python-top-programming-language-2020/

- Fecha: Feb 2021

Los diferentes rankings usan criterios distintos, y por ello no salen las mismas listas

En todo caso, como podemos ver, el lenguaje Python es uno de los más populares en la actualidad

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

Notas:

El nombre Python procede de la afición que tenía el autor del lenguaje por el grupo de humoristas Monty Python.

Python se creó con unos objetivos generales que en su mayor parte representan ventajas ofrecidas por este lenguaje:

- *Tipos dinámicos*: ya hemos comentado que esto significa que al operar con datos, en lugar de comprobar en tiempo de compilación si estos datos con compatibles, esto se hace en tiempo de ejecución. Esto implica menor eficiencia, pero mayor libertad y expresividad.
- *Gestión automática de memoria*: una de las formas más fáciles de equivocarse al programas es gestionar la memoria mal. Si la gestión de la memoria es automática, el error no es posible.
- *Objetos dinámicos*: se crean en el momento de usarse, lo que añade más flexibilidad.
- *Programación orientada a objetos y estructurada*. Ya hemos hablado de ellas. Mejoran la legibilidad y productividad.
- Mucho *software ya disponible*, lo que hace que sea muy productivo.
- Las distribuciones de *código abierto* tienen numerosas ventajas, en comparación con las distribuciones privativas, que pertenecen a una empresa. Una ventaja es que es gratis. Otra ventaja aún más importante es que cualquiera puede modificar el compilador y mejorarlo. La comunidad de desarrolladores aprovecha el trabajo en equipo de miles de personas para ir mejorando el producto.

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 ha sido discontinuada con la distribución 2.7.18 (Abr 2020)

La versión actual es la 3.9.1 (Dic 2020)

Trabajaremos con la versión 3.8 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)

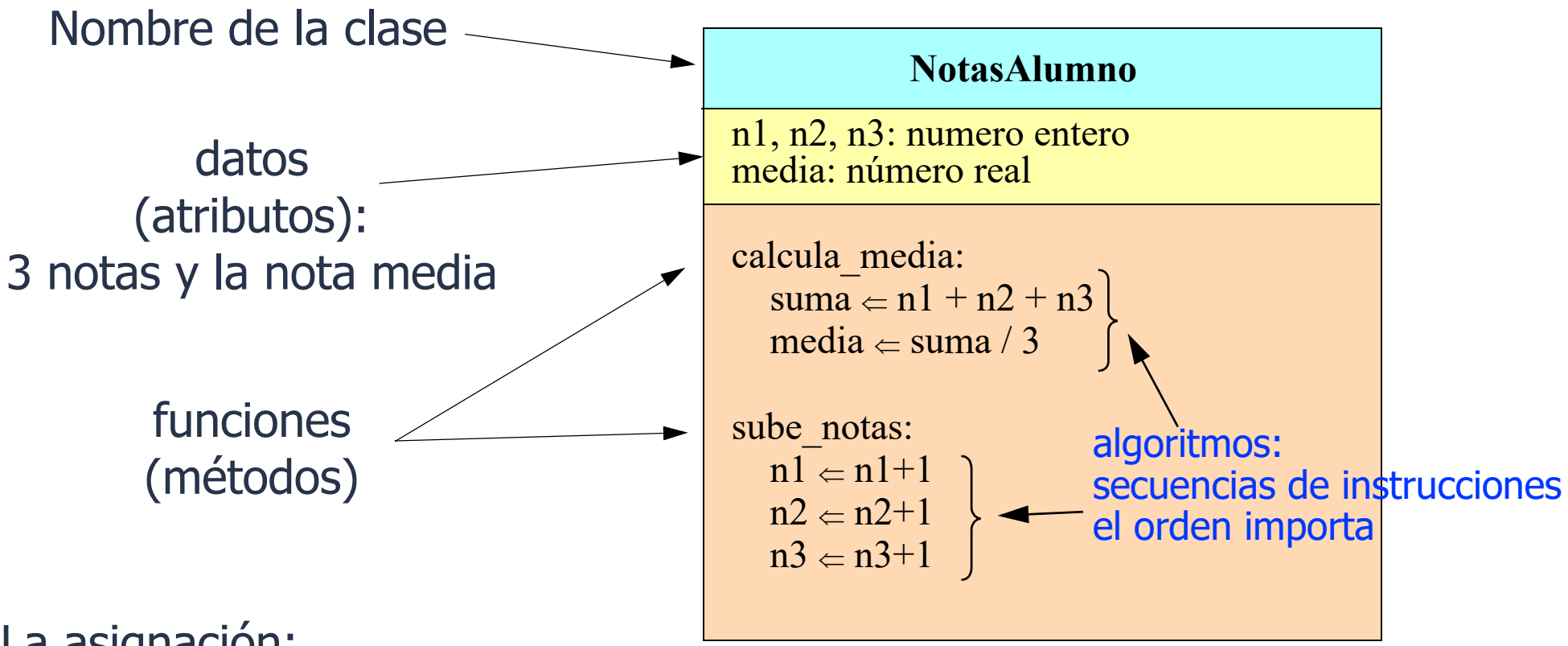
Cada "trozo" o *módulo* de programa contiene en su interior:

- *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 el *dato* de la izquierda
en Python lo representaremos con el signo **=**

Notas:

En el ejemplo se muestra parcialmente la estructura de una clase llamada `NotasAlumno`, cuyo objetivo es

- Almacenar cuatro datos:
  - tres notas de un alumno, llamadas `n1`, `n2` y `n3`, que son números enteros
  - almacenar la nota media del alumno, llamada `media`, como número real
- Contener dos operaciones (también llamadas funciones o métodos):
  - `calcula_media`: para calcular la nota media
  - `sube_notas`: para subir las tres notas en una unidad cada una

Usando esta clase como definición podremos crear múltiples objetos para almacenar las notas de muchos alumnos. Un objeto por cada alumno concreto. Cada objeto tendrá los datos definidos en la clase, pero con valores concretos.

- Por ejemplo, el objeto correspondiente al alumno "Pedro Suárez" contendrá sus tres notas, por ejemplo 7, 8 y 9, así como la nota media.

La clase está incompleta, pues al menos hará falta una operación para dar valor a las tres notas.

En las funciones usamos una instrucción básica en todos los lenguajes de programación: la asignación.

- La asignación permite dar valor a un dato.
- A la derecha del símbolo de la asignación se pone el nombre del dato.
- A la izquierda se pone el valor, que puede ser un cálculo más o menos complicado o un valor simple.

1.5. Estructura de un programa

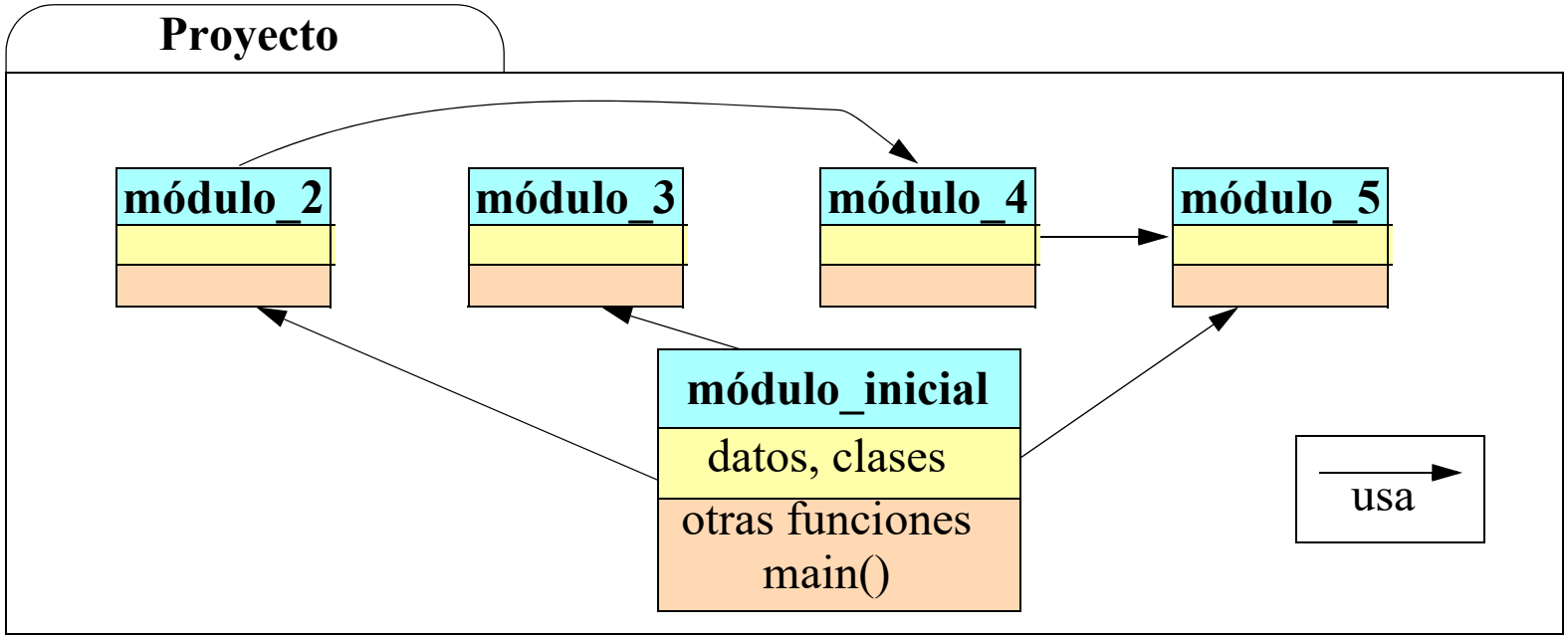
Un programa python es un conjunto de uno o varios módulos con:

- 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()`



Notas:

El programa se suele componer por muchos módulos, cada uno con una o varias funciones.

Surge un problema: Cuando queremos ejecutar el programa ¿Por dónde empezamos a ejecutar sus instrucciones?

La solución es tener un módulo con una función especial que por convenio llamaremos main(). Así, el programa siempre comenzará a ejecutar las instrucciones de esa función.

- Desde esas instrucciones podremos invocar otras funciones del mismo módulo o de otros.

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

Notas:

Este módulo tiene solo una instrucción suelta.

- Dijimos más arriba que no recomendamos poner instrucciones sueltas en los módulos.
- Enseguida veremos cómo escribir este módulo de forma correcta, aunque para hacer una prueba rápida nos vale.

El módulo está en un fichero acabado en `".py"`, para indicar que es un módulo python.

La instrucción única del módulo consiste en invocar la función predefinida `print()`.

- Esta función pone en pantalla el mensaje que se indica entre paréntesis, con un texto entre comillas.
- Es nuestra primera instrucción Python.

Un principio fundamental de la programación

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

Notas:

El programa se lee muchas veces:

- Cuando otras personas del equipo de trabajo intentan seguir trabajando en ese programa.
- Al tratar de entender lo que hace, para usarlo.
- Al intentar corregir fallos que pueda tener.
  - Por desgracia, los fallos en los programas son mucho más frecuentes de lo que desearíamos.
  - Programar bien es muy difícil.
- Al intentar mejorarlo.
  - Cualquier programa útil necesita evolucionar y mejorar.

Por tanto, es *imprescindible* hacer un esfuerzo extra al escribir el programa para que sea *fácil de leer*.

Una de las formas de conseguir esto (pero no la única) es proporcionando *documentación* adecuada.

- La documentación es texto que acompaña al programa y explica lo que hace.

En esta asignatura daremos *mucha importancia* a que los programas sean fáciles de leer

- y que por tanto estén acompañados de una documentación adecuada.

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/2020
"""
print("Hola, ¿qué tal?")
```

Notas:

Al escribir módulos y funciones es necesario manifestar para qué sirven, mediante una *documentación adecuada*.

Al crear un módulo, hay una documentación mínima exigida:

- Sistema de codificación
  - ¿Esto qué es? Como hemos indicado anteriormente, los computadores solo almacenan números binarios en su memoria. Sin embargo, los programas son texto.
  - ¿Cómo guardamos texto en la memoria del computador? Asignando a cada letra o símbolo un código numérico.
  - Como en el mundo hay muchos alfabetos, hay distintas maneras de hacer esto. Se llaman sistemas de codificación. Están estandarizados internacionalmente.
  - Nosotros usaremos el `utf-8`, que permite guardar en códigos de 8 bits (8 ceros y unos)) letras de muchos alfabetos, incluyendo el castellano.
  - El formato es un poco extraño, pero hay que copiarlo tal como se indica arriba en la primera línea del módulo.
- Entre triples comillas pondremos:
  - breve descripción
  - autor y fecha, con el formato indicado arriba.

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*

Notas:

Ya hemos dicho anteriormente que no se recomienda poner instrucciones sueltas en un módulo.

El motivo es que esas instrucciones no podrán ser invocadas desde otros módulos, y pierden utilidad.

La solución es meter las instrucciones dentro de una función.

- Siempre lo haremos así.

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

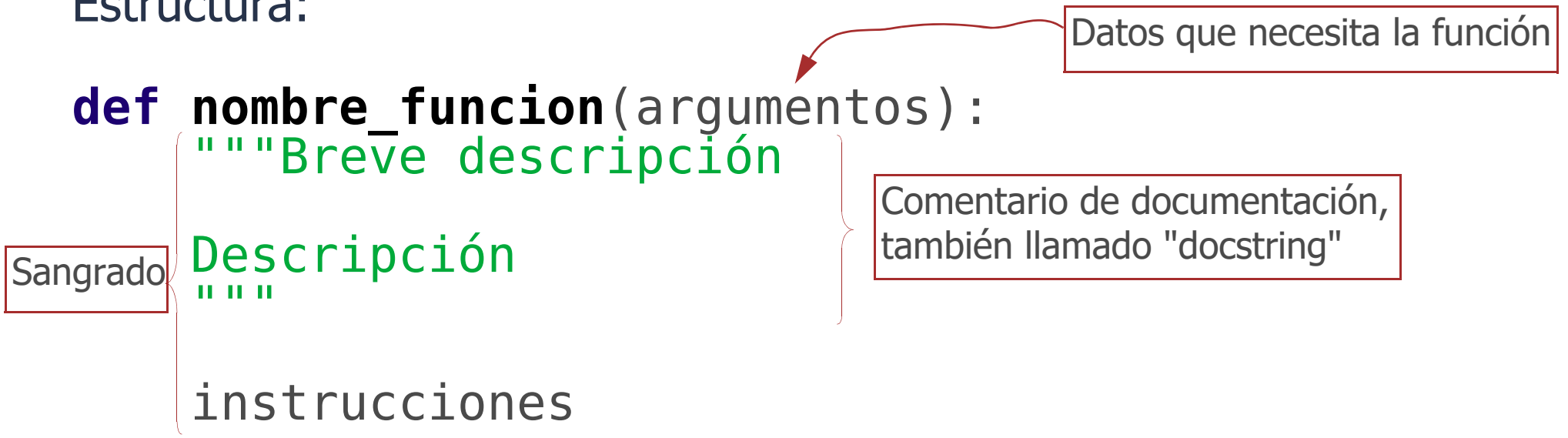
Estructura de una función

Estructura:

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

    Descripción
    """

    instrucciones
```

Estructura de una función

Estructura:



Invocar (ejecutar) una función

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

```
nombre_funcion(datos_concretos)
```

Ejemplo:

```
print("Hola, ¿qué tal?")
```

Notas:

La función comienza con la instrucción `def`

- `def` viene de `def`inir función.

A continuación se pone el nombre que queramos, aunque:

- no podremos usar palabras reservadas, que representan instrucciones Python: https://docs.python.org/3/reference/lexical_analysis.html#keywords
- ni debemos usar los nombres de las funciones predefinidas en Python, como print; si tenemos curiosidad los podremos encontrar aquí: https://docs.python.org/3/library/functions.html

Tras el nombre pondremos `( )` obligatoriamente.

- Opcionalmente, dentro del () podremos poner *argumentos*, que son datos que se pasan a la función desde el exterior, para que ésta los use; los veremos más adelante.

Finalmente, pondremos el contenido de la función, precedido de un *sangrado*

- El sangrado es un margen que se pone a la izquierda del texto, normalmente de 4 espacios en blanco.
- Marca el contenido de la función; cuando el sangrado se acaba, la función se acaba.

El contenido de la función incluye:

- comentario de documentación, entre triples comillas
- instrucciones de la función.

Hola Mundo "bien estructurado"

Función escrita en el fichero hola_mundo.py:


```
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 pone en el intérprete:

```
main()
```

Observaciones

Sobre el ejemplo anterior

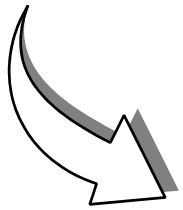
- sangrado
  - marca la estructura del programa y es obligatorio
- función sin argumentos (o parámetros)
- comentario de documentación, escrito entre `""" ..."""`

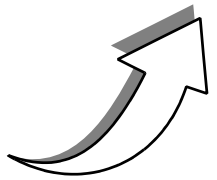
Resumen del programa "Hola Mundo"

```
print("Hola, ¿qué tal?")
```

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

@author: Michael Gonzále
@date  : 18/ene/2020
"""

print("Hola, ¿qué tal?")
```

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

@author: Michael González
@date  : 18/ene/2020
"""

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

Notas:

Daremos importancia al estilo de codificación.

- Influirá en la evaluación.

Muchas de las normas de estilo que se muestran aquí son comprobadas por el analizador de estilos de la herramienta `Spyder` que usaremos.

- Se invoca pulsando F8.

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

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 del encabezamiento

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

Notas:

Disponemos de dos tipos de comentarios

- *De documentación*: se ponen entre triples comillas y explican lo que hace un módulo, una función o una clase.
  - Están pensados para el usuario del módulo, función o clase.
- *Internos*: se ponen tras el símbolo # y son efectivos hasta el final de la línea.
  - Están pensados para los programadores que necesiten entender un algoritmo.
  - Criterio orientativo: usar uno cada *cinco o diez* líneas de código.

Hay que acordarse de usar ambos tipos de comentario:

- Hacerlos *breves*.
- Que sean *relevantes*.

Nunca poner comentarios para personas sin experiencia con Python. Se supone que los que leen nuestro programa son *expertos*.

- Por ejemplo, nunca poner "La instrucción print pone un mensaje en pantalla" antes de un `print()`.

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

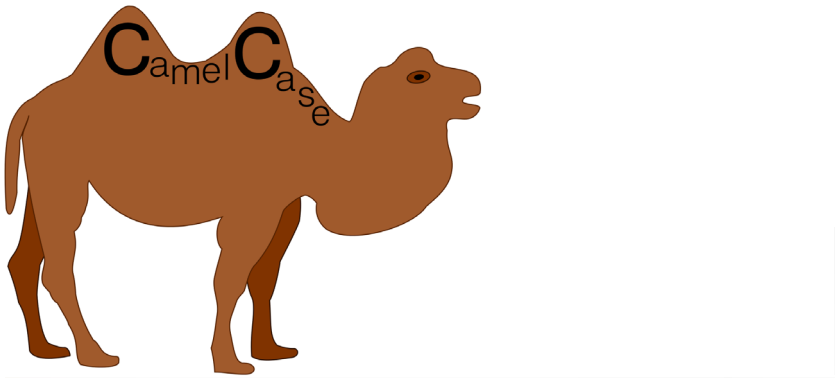
```
come(jamon[1], {chorizo:2})          #si
come ( jamon[ 1 ], { chorizo: 2 } ) #no
```

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

Notas:

Para remarcar la presencia de varias palabras en el mismo nombre no podemos usar espacios en blanco.

Los nombres que siguen el convenio *CamelCase* tienen:

- Comienzan por mayúscula.
- Luego siguen con letras minúsculas o cifras numéricas
- Para remarcar la presencia de varias palabras en el mismo nombre ponemos con mayúscula la primera letra de cada palabra.

Los nombres que siguen el convenio *snake-case* van lo más bajos posible (por el suelo):

- Todas las letras minúsculas.
- Pueden tener cifras numéricas, pero no comenzar por ellas.
- Separamos palabras dentro del mismo nombre con el carácter barra baja: _

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, 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