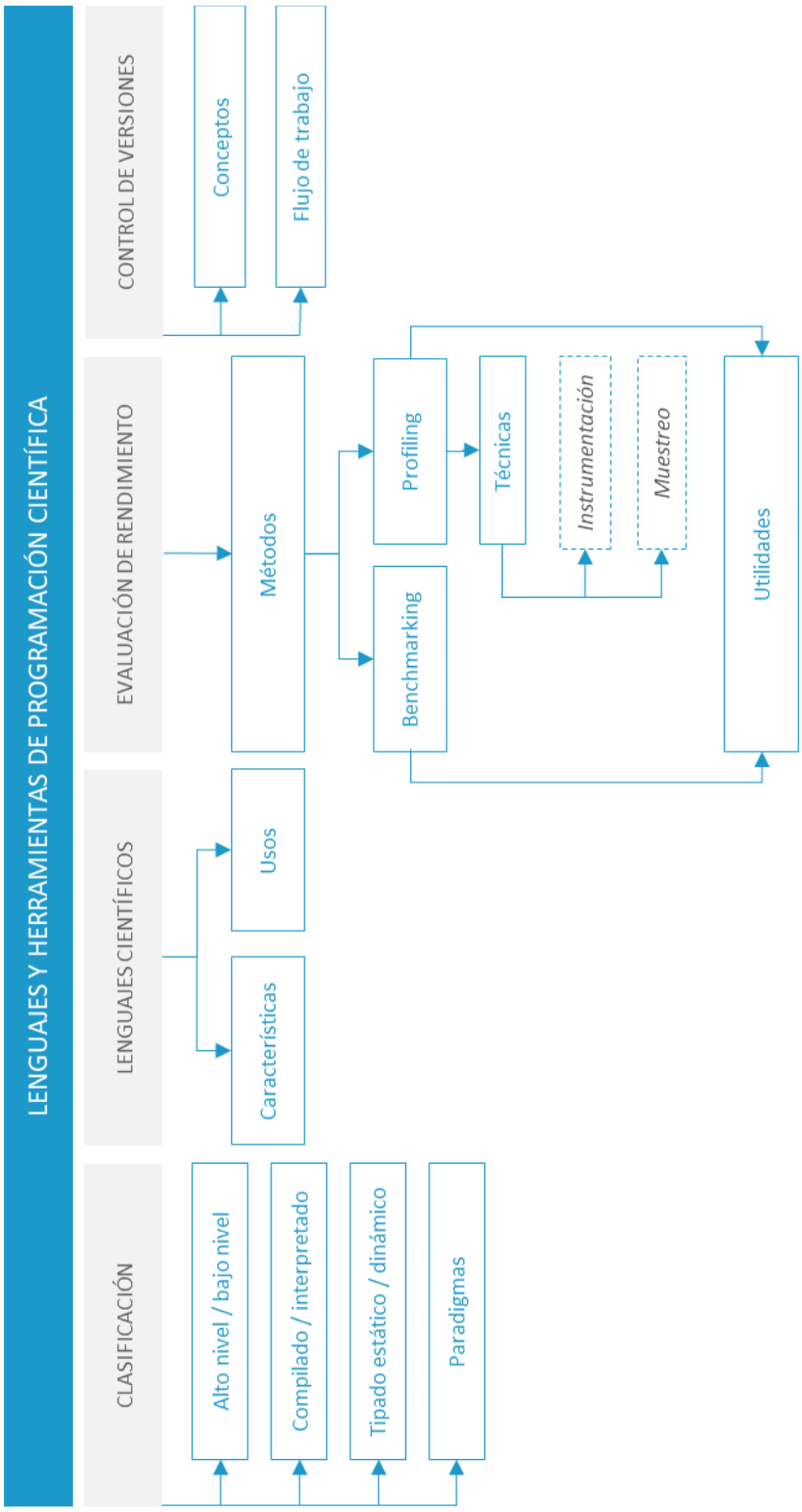
Programación Científica y HPC

Lenguajes y herramientas de programación científica

Índice

Esquema



Ideas clave

1.1. Introducción y objetivos

La irrupción de las computadoras programables, hace ya unos 75 años, y su posterior adopción generalizada entre la comunidad científica, **ha posibilitado el tratamiento y resolución de problemas cada vez más complejos,** hasta hace unos casi imposibles de resolver debido, entre otras cosas, a la cantidad de operaciones de cálculo que se necesitan realizar.

Del mismo modo, **la metodología empleada para programar estas máquinas ha evolucionado con el paso de los años,** especialmente en las últimas dos décadas, debido a la irrupción de Internet y las posibilidades de colaboración que ofrece. Hoy en día, **programar no es un acto aislado,** hay que tener en cuenta cómo sacar el máximo partido del ecosistema tan desarrollado del que nos proveen los principales lenguajes en el ámbito científico. Dicho ecosistema se compone de herramientas y bibliotecas o librerías auxiliares que facilitan diversas tareas, tales como la cooperación entre equipos, y ayudan, por ejemplo, a la toma de decisiones en la búsqueda de un código más legible o eficiente.

El objetivo de este tema es hacer una **revisión de las prácticas más aceptadas dentro de la comunidad científica en cuanto al desarrollo de software** destinado a computación, sin entrar aún en cuestiones específicas del lenguaje.

1.2. Clasificación de lenguajes de programación

Existen, literalmente, **miles de lenguajes de programación** y, a pesar de que obviamente no todos disfrutan del mismo grado de popularidad, es importante establecer ciertos baremos que permitan delimitar categorías y clasificar estos lenguajes, con base en sus características.

El nexo común que comparten todos los lenguajes de programación, y lo que los define como tales, **es su capacidad para expresar instrucciones** cuyo destino es ser ejecutadas por una máquina. **Sirve de intermediario entre los humanos,** que utilizan los lenguajes naturales para la comunicación, **y las máquinas** que utilizan un sistema binario basado en diferencias de potencial eléctrico. Como se puede intuir, las diferencias entre ambos métodos son inmensas; los lenguajes de programación ayudan a «tender puentes» entre ambas partes y suavizar estas diferencias.

Pueden contemplar un árbol evolutivo de los principales lenguajes de programación en Wikimedia. Examine las relaciones de influencia que se establecen entre ellos.

Los criterios bajo los que es posible clasificar los lenguajes de programación son igualmente variados, pero estos se pueden considerar los cuatro más importantes: nivel del lenguaje **(alto nivel *vs*. bajo nivel),** modo de procesamiento **(compilado *vs.* interpretado),** tipificación **(estática *vs.* dinámica)** y el **paradigma** que siguen.

Alto nivel *vs*. bajo nivel

Es importante no confundir estos conceptos, ya que el hecho de que un lenguaje sea de alto o de bajo nivel no significa que sea más o menos «capaz», «apto» o «desarrollado»; tiene que ver con la mayor o menor **cercanía conceptual** del lenguaje hacia la máquina en vez de hacia el humano o viceversa. L**os elementos léxicos y sintácticos de un lenguaje de bajo nivel son más sencillos** y se corresponden de

manera más directa con las instrucciones implementadas por hardware en el procesador. Por el contrario, **un lenguaje de alto nivel se parece más a la forma en la que se construyen las frases en un lenguaje natural,** siendo por tanto más sencillo de escribir y comprender por un humano. Es natural pensar que la elección en este caso es obvia, pero la principal **ventaja de los lenguajes de bajo nivel radica en su mayor rendimiento.**

En la siguiente tabla se pueden observar claramente las diferencias en cuanto a concisión y sencillez de dos programas que tienen el mismo efecto (mostrar el mensaje «¡Hola mundo!» por pantalla) implementado en dos lenguajes situados en extremos opuestos de la balanza del bajo y alto nivel:

Comparativa de lenguajes de bajo y alto nivel	
Ensamblador x86_64, notación Intel **Bajo nivel**	Python 3 **Alto nivel**

`section .text`
 	
`   global  _start`
 	
`_start:`
`    mov     edx, len`
`    mov     ecx, msg`
`    mov     ebx, 1`
`    mov     eax, 4`
`    int     0x80`
 	
`    mov     ebx, 0`
`    mov     eax, 1`
`    int     0x80`
 	
`section .data`
 	
`msg db`
`mundo!", 0xa`
`len equ     $ - msg`
	`print("¡Hola mundo!")`

Tabla 1. Comparativa de lenguaje de bajo y alto nivel. Fuente: elaboración propia.

Compilado *vs.* interpretado

Independientemente del nivel del lenguaje, siempre se necesitará realizar un proceso de traducción del código fuente en un código ejecutable en un procesador. Esto es llevado a cabo por un programa externo denominado **compilador** o **intérprete,** según el caso. La diferencia fundamental entre ambos radica en el momento en el que se realiza dicha traducción.

Programa compilador

Este trabaja **antes de tiempo,** también conocido como AOT (*Ahead-of-time*), en el sentido que las etapas de traducción y de ejecución están separadas. Esto tiene implicaciones en el rendimiento, un programa escrito en un lenguaje de programación compilado puede realizar comprobaciones y optimizaciones de código que incrementan el tiempo de procesado de forma significativa (mayor cuanto más grande y complejo sea el programa) y partir, al momento de la ejecución, de un **fichero o archivo ejecutable que ya contiene únicamente las instrucciones** que directamente ejecutará el procesador.

Programa interpretado

En este caso, las etapas son inseparables y **la ejecución se realiza a medida que se va realizando la traducción.** Tiene que transformar un código fuente en una secuencia de instrucciones ejecutables por el procesador, aplicando algunas optimizaciones sencillas por el camino y, finalmente, ejecutar el código resultante. Todo ello sin penalizar demasiado la interactividad que espera el programador.

Esto quiere decir que los lenguajes compilados presentan ventajas, sobre todo en la velocidad de ejecución, pero los lenguajes interpretados son, por norma general, **más sencillos y requieren de menos esfuerzo y líneas de código** para conseguir programas funcionalmente equivalentes a sus contrapartidas compiladas.

Una de las características más útiles que los lenguajes interpretados hacen posible, son los bucles de lectura-evaluación-impresión (REPL, por sus siglas en inglés), especialmente cuando se quieren hacer pruebas sencillas o cálculos rápidos. Puedes consultar su descripción en el artículo de Wikipedia.

Tipificado estático *vs.* dinámico

El tipificado hace referencia al tipo de una variable, argumento de función y valor de retorno, entre otros. A su vez, este tipo **no es más que una categorización para determinar qué clase de operaciones están definidas sobre ellos.** Por ejemplo, multiplicar dos números enteros es algo bien definido y que tiene sentido semántico y matemático, mientras que multiplicar dos cadenas de texto probablemente no lo tiene.

Un lenguaje de programación con **tipificado estático delega en el programador la tarea de especificar el tipo de todas las variables presentes en el programa** (o de proporcionar una manera de derivarlos sin ambigüedad), mientras que en los lenguajes de **tipificado dinámico es tarea del intérprete deducir estos tipos en tiempo de ejecución basándose en cómo son usadas estas variables.** Esto también significa que es posible que se produzcan errores de tipos incompatibles durante la ejecución de un programa interpretado, lo que no ocurre en la ejecución de un programa que proviene de un código fuente escrito en un lenguaje compilado, ya que estos errores habrían sido identificados durante la fase de compilación. Por último, cuando se trabaja con bases de código moderadamente grandes, la presencia explícita de tipos favorece los procesos de **refactorización.**

Paradigmas de programación

Un paradigma de programación está constituido por una serie de lenguajes, herramientas y técnicas, que **conforman un enfoque con características específicas para la implementación de algoritmos.**

Existen multitud de paradigmas de programación, y no son excluyentes entre ellos, es decir, un lenguaje de programación puede incluir elementos propios de varios paradigmas, no sólo uno. Pueden tener relación con características tales como la organización del código, la sintaxis del lenguaje y el modelo de ejecución. Es importante conocer las características que conforman los lenguajes de programación **imperativos, procedurales, funcionales** y **orientados a objetos.** Python es un lenguaje de **alto nivel, interpretado,** de **tipificado dinámico** y que soporta múltiples **paradigmas,** entre ellos, programación **imperativa, procedural, funcional** y **orientada a objetos.**

1.3. Lenguajes científicos

Al comenzar cualquier proyecto de programación, **una de las primeras decisiones que hay que tomar es el lenguaje principal que se va a utilizar.** Dentro de la comunidad científica se valoran aquellos lenguajes que favorezcan ciertas características, tales como extraer el máximo rendimiento posible de la máquina, la rapidez para experimentar, el soporte de librerías externas y contar con una comunidad con buena presencia en Internet que pueda proporcionar ayuda cuando sea necesario.

Hoy en día, los lenguajes que más se utilizan en el contexto de la programación científica son: **C/C++, Fortran, Python, R, Java, Scala** y **MATLAB/Octave**. En la Tabla 2 se presenta un resumen de algunas de las características más notables de los lenguajes mencionados.

Logo	Lenguaje	Nivel	Tipificado	Traducción	Usos principales
	C/C++	Bajo (C) Alto (C++)	Estático	Compilado	Computación de alto rendimiento, librerías matemáticas, computación heterogénea
	Fotran	Alto	Estático	Compilado	Computación de alto rendimiento, librerías matemáticas
	Python	Alto	Dinámico	Interpretado	Ciencia de datos, *big data*, aprendizaje automático
	R	Alto	Dinámico	Interpretado	Estadística
	Java	Alto	Estático	Compilado	*Big data*, computación distribuida, herramientas gráficas
	Scala	Alto	Estático	Compilado	*Big data*, computación distribuida
	MATLAB/ Octave	Alto	Dinámico	Interpretado	Computación numérica, simulación

Tabla 2. Características de lenguajes usados en programación científica. Fuente: elaboración propia.

Tanto Java como Scala utilizan un modelo mixto de compilación/interpretación, ya que compilan su código, pero no directamente a código máquina de la arquitectura en cuestión, sino a un *bytecode,* que representa a una máquina abstracta; posteriormente es labor de una máquina virtual denominada JVM (Java Virtual Machine) interpretar ese *bytecode* y ejecutarlo realmente. La principal ventaja de este enfoque es la portabilidad del código entre sistemas operativos y arquitecturas.

Pese a que prácticamente la totalidad de estos lenguajes son considerados de propósito general —es decir, pueden ser usados para tratar un amplio espectro de problemas—, lo habitual es que, con el paso de los años, se especialicen en ciertos nichos. Algunos de **los lenguajes de más reciente creación**, sobre todo, ya **han sido ideados con ciertas características para orientarlos a su uso en contextos determinados.**

1.4. Evaluación de rendimiento: *profiling*

Independientemente del lenguaje de programación que se haya decidido utilizar, un concepto universal aplicable a todos ellos es la capacidad para evaluar la eficiencia de ejecución de los programas implementados. Es importante tener en cuenta que esta cuestión se puede abordar desde diferentes puntos de vista. Se puede analizar la eficiencia de un programa o algoritmo de manera teórica; sin embargo, cuando se habla de ***profiling o benchmarking*** se hace referencia a un **enfoque práctico en el que se analiza el programa en el momento de ser ejecutado por la máquina.**

Probablemente, la manera más sencilla de **cuantificar cómo de eficiente es una determinada sección de código** (o programa al completo) es simplemente **medir el tiempo que tarda en ser ejecutada.** Esto puede ser realizado insertando, en el propio código del programa, marcas de tiempo justo antes y después de la región de interés. Por ejemplo, se puede usar `System.currentTimeMillis()` en Java o con `time.time()` en Python, como se muestra en el siguiente código:

```
import time

start = time.time()
# Aquí el código de interés
stop = time.time()
print(f"Tiempo empleado: {stop - start}
segundos")
```

Figura 1. Uso de `time.time()` en Python. Fuente: elaboración propia.

Esto proporciona una manera rápida y sencilla de comparar, por ejemplo, dos implementaciones diferentes para resolver un mismo problema y puede ayudar a tomar la decisión de usar la implementación más rápida. Sin embargo, es importante tener en cuenta que este método tiene ciertas limitaciones, como las que se enumeran a continuación:

- **Poca robustez:** este método es muy dependiente de todo lo demás que ocurra en la máquina mientras se está ejecutando el programa, y arrojará medidas

significativamente diferentes si la máquina está en reposo o, por el contrario, está ejecutando otros programas al mismo tiempo.

- **Resolución limitada:** en general, cuanto menor sea el tiempo que tarde la sección de interés, menos posibilidades hay de obtener mediciones fiables. En el caso de Python, deben transcurrir al menos 0,1 segundos entre las llamadas a `time.time()` o, de lo contrario, registrará siempre 0 segundos de tiempo transcurrido.

Dentro de esta misma categoría también existe la posibilidad de cronometrar el tiempo que tarda el programa en ejecutarse completo. Dos ejemplos serían las aplicaciones auxiliares `time` e `hyperfine`. En la Figura 2 se muestra un ejemplo de ambas en acción, midiendo el tiempo que tarda un pequeño programa en Python en computar la función de Ackermann con parámetros `m=3` y `n=6`:

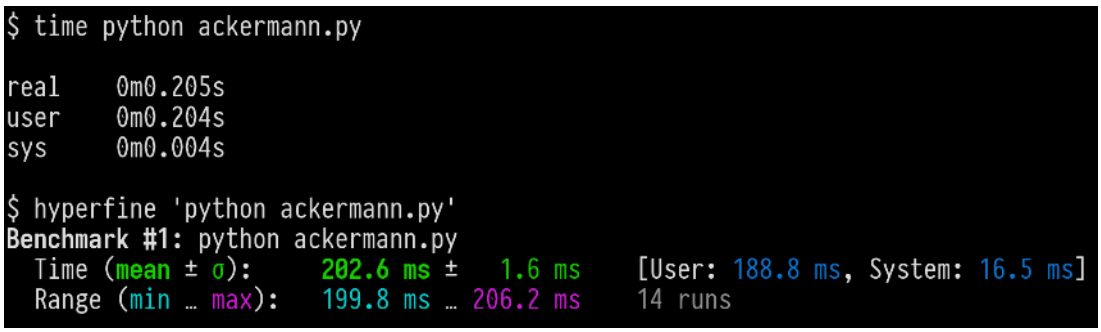


Figura 2. Ejemplo de benchmarking con `time` e `hyperfine`. Fuente: elaboración propia.

En la imagen previa se ha ejecutado el mismo programa dos veces, una anteponiendo el comando `time` antes de la llamada al intérprete de Python y otra anteponiendo el comando `hyperfine`. Como se puede observar, el tiempo total de ejecución que devuelve time es de 0,205 segundos, que prácticamente coincide con el tiempo devuelto por `hyperfine` (0,203 segundos). Sin embargo, salta a la vista que `hyperfine` es más **detallado y cuidadoso con la metodología empleada para obtener estas mediciones**, ya que ejecuta el comando objetivo (`python ackermann.py`) varias veces y obtiene una media de los tiempos de ejecución, calculando también la desviación estándar y los valores mínimos y máximos obtenidos. **Esto permite juzgar con mayor**

precisión el rendimiento real del programa, y son un ejemplo de medidas que pueden tomarse para incrementar la **robustez** y disminuir la **sensibilidad al ruido**.

Las mediciones de tiempo transcurrido (*wall-clock time*, como se conoce este concepto en inglés) sólo pueden dar una estimación *grosso modo* de lo que tarda el programa en finalizar sus cálculos.

Supóngase la siguiente situación: se tiene un programa relativamente complejo (unos pocos miles de líneas de código) que tarda diez minutos en realizar una serie de cálculos y devolver el resultado. Si se quisiese optimizar este aspecto y reducir esos tiempos en un orden de magnitud, para que cada ejecución tome alrededor de un minuto, ¿por dónde se debe empezar?, ¿cómo se puede saber dónde están las mejores oportunidades de optimización? **Es importante conocer en qué partes del código se concentra la mayor parte del tiempo empleado para concentrar ahí los esfuerzos.**

El principio de Pareto señala que el 80 % del tiempo de ejecución de un programa se emplea en tan sólo un 20 % de líneas de código. Esto no es más que una observación empírica, pero recuerda que el tiempo total empleado por un programa en ejecutarse no está distribuido homogéneamente entre las líneas de código que lo componen.

Es aquí donde entra el *profiling* en escena. Su nombre viene de obtener un **perfil de rendimiento** de un programa, que puede indicar el porcentaje del tiempo total empleado en cada línea de código, tiempos acumulados, número de llamadas a cada función, tiempo medio empleado por cada llamada y otras medidas. Sin duda **es capaz de proporcionar información mucho más completa y detallada** sobre el empleo del tiemplo en el programa analizado y, de acuerdo con esta información**, los desarrolladores podrán tomar decisiones** mucho más fundamentadas **acerca de cómo acortar los tiempos de cómputo.**

Hay que tener en cuenta, eso sí, que esta precisión extra no es gratis. Los *profilers* son capaces de obtener toda esa información porque **utilizan métodos más agresivos** para el seguimiento del código, y **la consecuencia es que el propio rendimiento del código se ve afectado** cuando está bajo *profiling*. A continuación, se listan las dos principales técnicas de *profiling*, junto a sus ventajas e inconvenientes:

- **Muestreo:** consiste en tomar una muestra de la pila de llamadas cada cierto tiempo, a intervalos regulares. La pila de llamadas indica qué función está siendo ejecutada actualmente y a través de qué otras llamadas a funciones han llegado hasta ahí. Los intervalos de muestreo son configurables, aunque es habitual escoger valores como 50 o 99 *Hz* (muestras por segundo). **A mayor frecuencia, más fino será el detalle** de cómo se desgrana la ejecución del programa, pero a expensas de influir cada vez más negativamente en el tiempo de ejecución. **A frecuencias demasiado bajas, se pueden perder detalles importantes** de funciones cortas que son llamadas muchas veces.

- **Instrumentación:** la instrumentación «acompaña» al programa en toda su ejecución y va tomando nota de todas y cada una de las llamadas a funciones que se producen, así como el tiempo y las instrucciones que emplean cada una de ellas. La ventaja es, por supuesto, que **se obtendrá el perfil más detallado posible del programa analizado, pero** el inconveniente es que la ejecución del programa **puede ralentizarse enormemente** en comparación con una ejecución normal. Factores de ralentización de más de 100x no son raros de ver en estos casos.

En Python, profile y cProfile son los *profilers* más utilizados. En la Figura 3 y la Figura , se pueden ver algunas representaciones gráficas de los informes que generan:

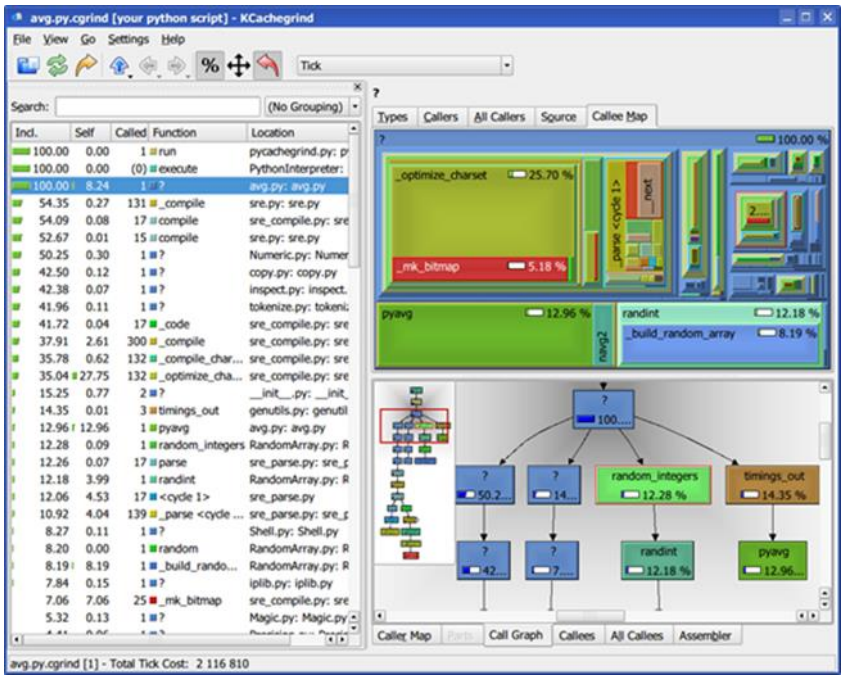


Figura 3. KCacheGrind como interfaz para cProfile. Fuente: http://www.fperez.org/py4science/profiling/index.html

Figura 4. *Flame graph* de llamadas. Fuente: https://www.nylas.com/blog/performance/

1.5. Control de versiones: *git*

Actualmente es difícil concebir el desarrollo de cualquier tipo de software, especialmente en equipo, sin algún tipo de **herramienta de control de versiones.** Estas herramientas **siguen la evolución del software** controlando qué cambios se realizan en los ficheros de código y proporcionan la capacidad de volver a una versión anterior en caso de problemas. Además, permite la colaboración sencilla entre integrantes de un mismo equipo de trabajo, entre otras muchas características. Existen variadas herramientas de este estilo, como **CVS, subversion, mercurial, perforce** y **bazaar**. Sin embargo, la herramienta de control de versiones *de facto* hoy en día es *git*.

La importancia de familiarizarse con su uso viene en gran parte dada porque **una enorme cantidad de proyectos,** tanto de índole científica, como de otras muchas áreas, **son distribuidos en forma de repositorios** de *git*. Estos contienen, no solo los ficheros que componen el proyecto, sino todo su historial de cambios hasta llegar al momento de la consulta o descarga. Asimismo, se considera una buena práctica mantener un apropiado control de versiones desde el comienzo del desarrollo del proyecto. Técnicamente, es posible introducir *git* en un desarrollo ya en marcha, pero las ventajas disminuyen al no tener un historial de los cambios anteriores al momento en que se introduce.

A continuación, se presentan algunos conceptos de *git*:

- **Repositorio:** un conjunto de ficheros bajo el seguimiento de *git*. En la práctica se hace referencia a un **directorio y todos los ficheros y subdirectorios que aloja.** Los repositorios pueden ser **locales o remotos**, es decir, si se refiere a una copia presente en un sistema de ficheros local o si está alojado en otra máquina accesible por red, respectivamente. Los ficheros bajo seguimiento pueden también **sincronizarse** entre repositorios locales y remotos.

- ***Commit:*** representa un «punto de control» en la historia del proyecto. Un *commit* **trae consigo una serie de cambios** que deben registrarse y un mensaje que describe esos cambios. Muchas de las operaciones soportadas por *git* trabajan sobre *commits.*

- **Área de trabajo (*working area*):** por defecto, **todos los cambios que se hagan en los ficheros están en el área de trabajo,** desde donde explícitamente se indicará a *git* que se quieren registrar los cambios de un determinado fichero. Cuando eso ocurre, este fichero pasa del área de trabajo al área de preparación.

- **Área de preparación (*staging area*):** esta área se puede entender como una «previsualización» del *commit* que se está preparando. Aquí se pueden hacer **ajustes finos de los cambios** que se quieren registrar exactamente. Cuando finalmente se ejecuta el *commit*, los cambios pasan a reflejarse en el repositorio local correspondiente, y el área de preparación queda limpia y lista para el siguiente *commit*.

- **Ramas (*branches*):** a diferencia de otros sistemas de control de versiones, *git* favorece enormemente el trabajo con ramas. Con ellas **se puede trabajar independientemente en características nuevas,** sin afectar a la base de código principal, y cambiar sin esfuerzo entre una y otra, o fusionarlas.

Existe bibliografía que expone las capacidades de *git* con el objeto de que puedan ser aprovechadas al máximo, y que ayudan a comprender las diferentes maneras de organizar un flujo de trabajo de acuerdo con sus idiosincrasias.

Accedan al vídeo de la lección magistral, **Gestionando proyectos de código con *git,*** que muestra cómo usarlo para la gestión de proyectos sin necesidad de usar el terminal.

Accede al vídeo

En su versión más sencilla, el flujo de trabajo con *git* suele tener este aspecto:

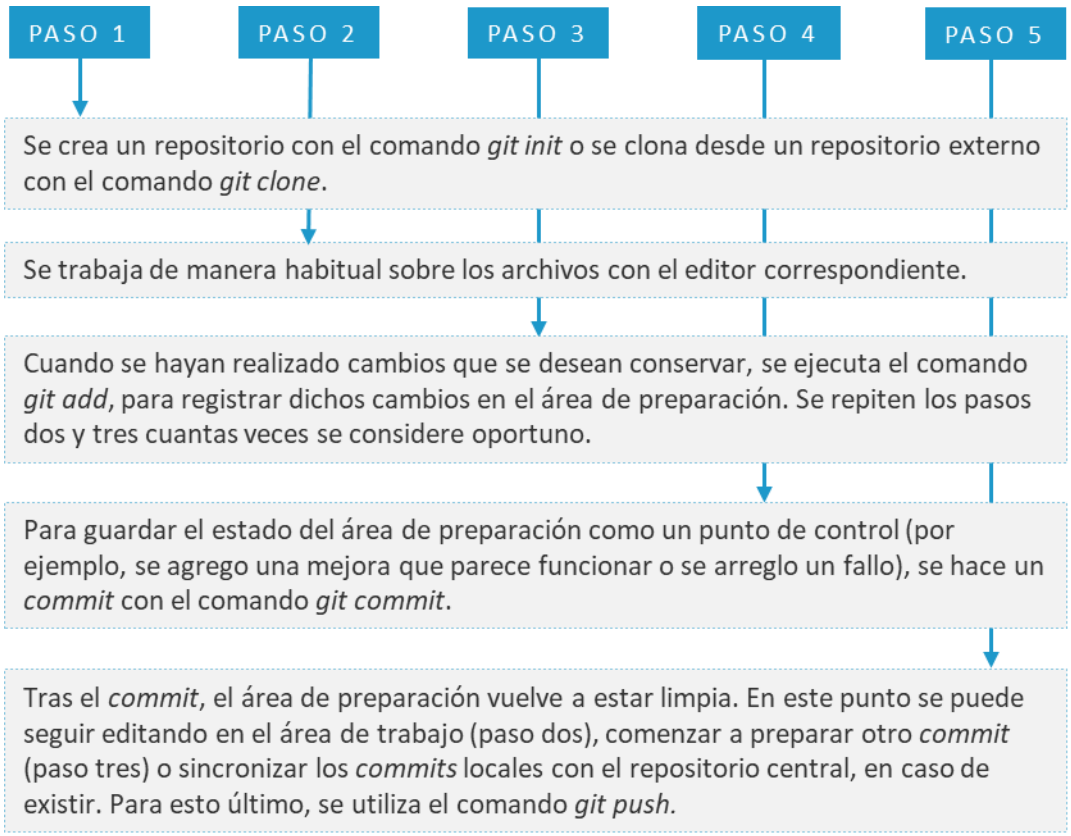


Figura 5. Flujo de trabajo con *git*. Fuente: elaboración propia

Si se quisiese volver a una versión anterior. Se pueden **descartar los cambios** que aparecen en el área de trabajo con el comando `git restore`, y los que se tienen en el área de preparación (después de haber ejecutado `git add`) con el comando `git restore--staged`. Si se quiere **restaurar el estado de un fichero** al que tenía en algún commit anterior, se puede usar el comando `git reset`.

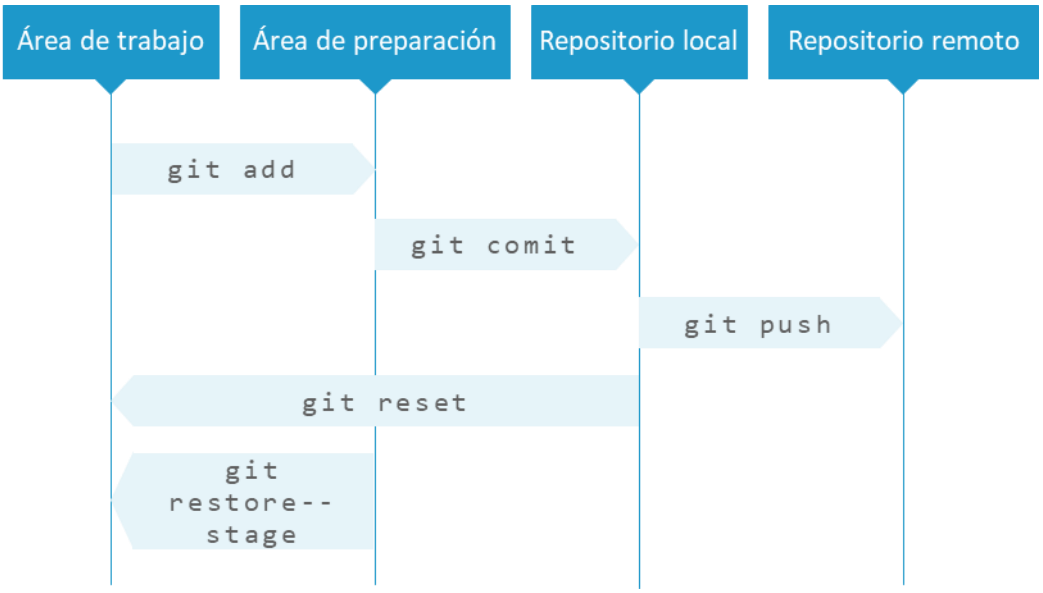


Figura 6. Flujo de trabajo sencillo en *git* Fuente: elaboración propia.

1.6. Referencias bibliográficas

Perez, F. (s. f.). *KCacheGrind como interfaz para cProfile* [Imagen]. http://www.fperez.org/py4science/profiling/index.html

Morikawa, E. (2016). *Flame graph de llamadas* [Imagen]. https://www.nylas.com/blog/performance/

A fondo

Paradigmas de programación

Programming paradigm (20 de julio de 2021) En *Wikipedia.* https://en.wikipedia.org/wiki/Programming_paradigm

Si quieren indagar más acerca de los diferentes paradigmas de programación existentes, le sorprenderá saber la inmensa cantidad de ellos que existen. El artículo correspondiente en Wikipedia tiene una lista bastante exhaustiva con características generales de todos ellos.

***Zine* de *profiling* y trazado de programas**

Evans, J. (2018). *Profiling & Tracing with perf* [zine]. https://jvns.ca/perf-zine.pdf

Julia Evans es una ingeniera de software canadiense muy conocida por hacer temas complejos accesibles, a través del formato *zine* (pequeñas revistas de veinte o treinta páginas en formato cómic). La que se enlaza aquí está dedicada al comando `perf` y en general a explorar más cómo funciona realmente el *profiling* por dentro.

Tutorial de treinta minutos en vídeo para empezar a usar *git*

Traversy Media (5 de febrero de 2017). *Git & GitHub Crash Course For Beginners* [Archivo de vídeo]. Youtube. https://www.youtube.com/watch?v=SWYqp7iY_Tc

Este vídeo los ayudara si nunca han manejado *git*. También cubre las instrucciones para el proceso de instalación.

Practicar conceptos avanzados de *git* en el navegador

Learn Git Branching (https://learngitbranching.js.org)

Learn Git Branching permite repasar lo básico de *git* y practicar conceptos más avanzados, como crear y fusionar ramas, revertir cambios, incorporar cambios de terceros, gestionar conflictos, etc. Todo ello en una plataforma web sin necesidad de instalar nada.

Referencia rápida de *git* en español

Training GitHub (https://training.github.com/downloads/es_ES/github-git-cheat-sheet/)

Una pequeña guía rápida con los comandos más comunes, con descripciones en español. Se recomienda guardarla y extenderla con los nuevos comandos que se vayan aprendiendo.

Test

1. ¿Cuál de las siguientes características no es cierta de Python?

A. Es un lenguaje con tipificado estático.

B. Es un lenguaje de alto nivel.

C. Es un lenguaje interpretado.

D. Es un lenguaje multiparadigma.

2. Un lenguaje [...] tiene una correspondencia muy marcada entre sus elementos sintácticos y gramáticos, y las instrucciones que ejecutará el procesador.

A. Estático.

B. Imperativo.

C. Compilado.

D. De bajo nivel.

3. ¿Cuál de los siguientes lenguajes de programación no es comúnmente utilizado en la comunidad científica?

A. Scala.

B. JavaScript.

C. Python.

D. MATLAB.

4. Acerca de *profiling* por muestreo e instrumentación, escoge la opción correcta:

A. El *profiling* por muestreo tiene un impacto mayor que por instrumentación en el rendimiento del programa.

B. El *profiling* por muestreo tiene un impacto menor que por instrumentación en el rendimiento del programa.

C. La frecuencia de instrumentación determina cuán precisos serán los perfiles obtenidos.

D. La instrumentación no provee un perfil detallado de ejecución, sólo una aproximación.

5. ¿Cuál de las siguientes no es una ventaja del *benchmarking* de tiempo transcurrido?

A. No se ve afectado por factores externos, tales como el porcentaje de uso actual del procesador.

B. Es un método sencillo de obtener una cuantificación aproximada del rendimiento.

C. Apenas tiene influencia en el rendimiento del propio programa comparado con una ejecución sin benchmarking.

D. Casi siempre, el propio lenguaje tiene soporte nativo en sus librerías estándar para establecer y manipular marcas de tiempo.

6. ¿Cuál de las siguientes opciones representa uno de los *profilers* más ampliamente utilizados en Python?

A. KcacheGrind.

B. *Flame graph*.

C. cProfiler.

D. `Hyperfine`.

7. ¿Cuál de los siguientes conceptos no es propio de *git*?

A. Repositorio.

B. *Commit*.

C. Área de clonado.

D. Rama.

8. El comando para sincronizar los *commits* con un repositorio remoto es:

A. `git commit.`

B. `git add.`

C. `git clone.`

D. `git push.`

9. ¿Cuál de los siguientes comandos de *git* no está relacionado con descartar cambios que no nos interesan?

A. `git init`.

B. `git restore`.

C. `git reset`.

D. `git restore--staged`.

10. La función principal del área de preparación de *git* es:

A. Proporcionar un espacio donde crear un proyecto vacío con `git init`.

B. Alojar un repositorio remoto.

C. Editar los ficheros normalmente.

D. Previsualizar y realizar ajustes en el siguiente *commit* que se llevará a cabo.