María Paula Llano

Departamento de Ciencias de
la Atmósfera y los Océanos

Laboratorio de Procesamiento de Información Meteorológica/Oceanográfica

Buenas Prácticas

Cuando tenemos un “problema” y queremos programar una solución, vamos a tener tantas maneras da hacerlo como personas pensando en dicha solución.

Al momento de diseñar nuestros algoritmos y de traducirlos en un lenguaje de programación específico, existen consejos o reglas que conviene tener en cuenta para:

- ahorrar tiempo y ser eficientes
- minimizar el potencial de errores
- facilitar el intercambio de nuestras soluciones con nuestros pares
- facilitar el crecimiento de nuestro código (desde códigos simples para fines específicos hacia un rango mayor de propósitos)

Seguir un estilo de programación:

Esto se refiere principalmente a cómo presentar el código de un programa para que sea más fácil de interpretar. Algunos aspectos importantes en un estilo de programación son:

- convenciones para nombrar variables, funciones, archivos, etc
- indentación
- manejo de dependencias

Existen variedad de estilos posibles, pero hay algunos aspectos en los que todos concuerdan y estos son los puntos que vamos a mencionar.

Convenciones para nombrar:

Cuando se trata de nombrar archivos, variables y funciones, es importante que los **nombres** sean los más **autodescriptivos** posibles.

Por ejemplo si tenemos que almacenar el valor de la aceleración por gravedad en el vacío en una variable podemos hacer:

my_var_10 <- 9.81

O podemos hacer

Grav <- 9.81

O si vamos a considerar a esta variable como un parámetro que no cambia en nuestro programa

ConstGrav <- 9.81

Estas 3 son válidas, pero las últimas dos nos van a permitir recordar más fácilmente que es lo que está almacenado en la variable en cuestión. También será más fácil para alguien que ve nuestro código entender que es lo que hace.

Podemos seguir reglas para nombrar variables, constantes y funciones y tratar de mantenerlas a lo largo del programa y entre los diferentes programas que escribamos. **Nombres derivados del inglés son siempre preferibles**.

Cuando se trata de parámetros o valores que queremos que permanezcan constantes, algunos lenguajes como Fortran tienen tipos de datos especiales para asegurar que eso pase (es decir que una vez asignado el valor no pueda ser modificado a lo largo del programa).

Indentación:

```
for ( valor in MiLista ) {
  if ( valor != -999 ) {
     ValorMedio <- ValorMedio + valor
     CantidadDatosValidos <- CantidadDatosValidos + 1
  }
}
```

```
for ( valor in MiLista ) {
if ( valor != -999 ) {
ValorMedio <- ValorMedio + valor
CantidadDatosValidos <- CantidadDatosValidos + 1
}
}
```

La **indentación** es agregar espacios o tabs al inicio de cada línea en el código.

Es una manera de facilitar la interpretación de las estructuras como ciclos o condicionales.

Su uso se vuelve muy importante cuando tenemos ciclos y condicionales anidados entre sí.

Indentación y espacios:

```
for ( valor in MiLista ) {
  if ( valor != -999 ) {
    ValorMedio <- ValorMedio + valor
    CantidadDatosValidos <- CantidadDatosValidos + 1
   }
}
```

En algunos lenguajes como python, la indentación es obligatoria. Esto es, el inicio y final de un ciclo o un condicional, están definidos por la indentación (a diferencia de R que usa {}) para controlar esto.

Otro aspecto que hace al estilo del código fuente para facilitar su lectura e interpretación es el uso de espacios.

En general es una buena práctica darle “aire” a nuestro código y separar los nombres de las variables de los operadores por uno o más espacios.

El uso de múltiples espacios se justifica si queremos incrementar la alineación de un mismo operador (ej <-) en múltiples líneas consecutivas.

```
TotalValue      <- mean( ItemValue )
TotalValueTax   <- TotalValue + Tax
TotalValueDisc  <- TotalValueTax - Discount
```

paqeuetes se instalan en la compu y se "traen/ llaman" al script
Para "llamar" se usa library (). Esta bueno ponerla al principio

Manejo de dependencias:

Otro aspecto que es recomendable ubicar al principio de un código, es la **lista de paquetes** que ese código va a utilizar.

Podemos cargar el paquete usando la función **library()**.

Al ubicar esto al principio del código será más fácil para un usuario entender las dependencias y en caso que no estén disponibles el código dará un error al inicio.

En general, salvo que sea absolutamente necesario, no es una buena idea incluir la instalación o desinstalación de paquetes como parte de nuestro código. El mantenimiento de los paquetes disponibles en nuestro sistema es algo que debemos hacer independientemente de nuestro código.

Documentación

La documentación de un código es fundamental para nosotros mismos y para terceros que puedan utilizar este código. No obstante es bueno tener en cuenta algunos consejos generales:

- Siempre comentar el propósito del código y el por que de determinadas acciones (más que describir qué es lo que está haciendo).
- Balance entre cantidad y calidad. No es necesario comentar cada línea de código. Más bien identificar secciones de código y describir su propósito en términos generales.
- Siempre documentar las funciones. Esto es probablemente uno de los aspectos más críticos de la documentación.

- Mantener la documentación, tanto como mantenemos nuestro código. Muchas veces hacemos modificaciones al código para mejorarlo o corregir errores, pero olvidamos modificar de manera consistente la documentación que acompaña a dichas modificaciones.
- Trabajar sobre la “interpretabilidad” de un código para que necesite menos documentación. Resolver 10 acciones en una sola línea de código puede ser un desafío interesante, pero muchas veces puede oscurecer el código. Buscar un balance entre “interpretabilidad” y eficiencia a la hora de escribir un código.

Algunas cosas que no deberían faltar:

- **Un comentario inicial** en el programa principal indicando el propósito general del programa, los tipos de datos sobre los que opera (si lee de un archivo, si se ingresa por pantalla), las dependencias y un breve versionado del código (lista de modificaciones más importantes y su propósito).
- Documentación de las funciones que definamos, indicando cantidad y tipo de parámetros de entrada y cantidad y tipo de parámetros de salida junto con el propósito de la función. En caso que aplique se puede incluir una cita bibliográfica documentando el algoritmo que se implementa en dicha función.
- Explicación de decisiones particulares y poco triviales que debieron ser tomadas para solucionar alguna propiedad específica de los datos con los que se trabaja (por ejemplo el cálculo de la media con datos faltantes).

Dependiendo el tipo de aplicación podemos pensar en diferentes tipos de documentación:

Documentación orientada a usuarios: ¿Qué debe saber alguien que quiere usar mi código? (cómo preparar los datos de entrada, como "configurar" el código, en qué forma se generarán los resultados). Hay requisitos previos para poder usar el código (ej, debo instalar algún paquete adicional para que funcione el código?).

Documentación orientada a otros desarrolladores: ¿Qué debe saber alguien que quiere continuar el desarrollo de mi código o modificarlo para sus propósitos? De qué manera está estructurado el código. Cuales son las convenciones de nombres que se usaron. La documentación de las funciones individuales que componen el código, etc.

No escribir código que no sea necesario:

Esto se refiere a encontrar un balance entre potenciales generalizaciones y expansiones de nuestro código y el cumplimiento de los requerimientos actuales.

Ejemplo: Necesito un código que calcule la media de un vector numérico. Se que actualmente todos serán números.

- Puedo querer tener en cuenta que: el vector puede tener diferentes largos, puede haber datos faltantes (con su código respectivo). Estas son cosas que suceden típicamente y por ende son consideraciones razonables a tener en cuenta al diseñar el código.
- Puedo además querer que: el código calcule la media cuando el input es un número complejo, el código calcule diferentes tipos de media (aritmética, geométrica, pesada, armónica, etc), el código realice un análisis estadístico completo, incluyendo media, varianza, asimetría, curtosis, ajuste a una normal, etc.

Si comparamos el objetivo original con el segundo grupo, veremos que estamos sobredimensionando la complejidad del problema. Esto implica que invertiremos tiempo en solucionar problemas que no existen (al menos por el momento).

Esto nos puede llevar a un código innecesariamente complicado (e incrementar la probabilidad de errores en detrimento del objetivo original).

Si es una buena práctica contemplar posibles expansiones futuras y diseñar nuestros algoritmos con esto en mente, aún cuando no implementemos estas mejoras.

Utilizar al máximo las herramientas disponibles: funciones intrínsecas, librerías, etc.

Cuando tengamos que enfrentar el desafío de programar un algoritmo complejo, lo primero es verificar que no esté disponible como parte de una librería en nuestro lenguaje o en otro lenguaje con el cual podamos interactuar.

En caso que no esté disponible podemos verificar que no exista alguna función o conjunto de funciones que nos facilite la tarea. Es decir que puedan resolver algunas funcionalidades que necesitamos para lograr tener funcionando nuestro algoritmo.

Esto no solo ahorra tiempo de diseño y programación, también ahorra mucho tiempo en optimización y testeo. Las librerías disponibles usualmente (aunque no siempre) son sometidas a un testeo y a un proceso de optimización exigente.

DRY (Don’t repeat yourself):

A medida que vayamos escribiendo algoritmos más complejos vamos a notar que escribimos códigos para resolver el mismo problema muchas veces.

A veces dentro de un mismo algoritmo necesitamos realizar un mismo proceso dos o más veces. Ej: si quiero calcular la diferencia entre las medias de dos conjuntos de datos, tengo que calcular la media para el conjunto 1 y luego para el conjunto 2.

No es una buena práctica repetir código haciendo “copy - paste”. Cuando una tarea se ha vuelto repetitiva dentro de un código necesita ser “abstraida” y convertida en una función que me permita reutilizarla una y otra vez en diferentes partes del código o incluso en diferentes códigos.

Por ejemplo en el caso anterior, podemos usar la función intrínseca mean() para calcular la media siempre que sea necesario (y si no existe como función intrínseca, programarla como una función propia).

Con el tiempo vamos a crear nuestras propias librerías, es decir colecciones de funciones que usamos frecuentemente y que nos van a permitir avanzar mucho más rápido a la hora de solucionar problemas nuevos.

Evitar el “hardcoding”

Las constantes o parámetros son valores que queremos que mantengan su valor a lo largo de todo el programa.

Siempre es recomendable usar este tipo de valores para no incluir “números mágicos a lo largo del código”.

```
for ( valor in MiLista ) {
    ValorMedio <- ValorMedio + ( valor / 10 )
}
```

En este segmento de código queríamos calcular la media de un vector de dimensión 10. 10 es un número “mágico” ya que depende de nuestro problema en particular. Así como está el código decimos que el valor 10 esta “harcodeado”.

En un código complejo con cientos o miles de líneas, tener valores "harcodeados" es un problema potencial, ya que si queremos modificar dichos valores vamos a tener que rastrearlos a lo largo del código. Lo más recomendable en estos casos es almacenar el valor en una variable que se defina al principio del código.

```
Const.VectorLen = 10

....
for ( valor in MiLista ) {
    ValorMedio <- ValorMedio + ( valor / Const.VectorLen )
}
```

En este ejemplo no es muy evidente la ganancia, pero si el cálculo de la media estuviera inmerso en un código con miles de líneas, sería mucho más fácil modificar la longitud asumida para los vectores. Otra ventaja, es que escribiendo o "declarando" al inicio del código nuestras constantes, estamos dejando en claro cuales son las hipótesis sobre las cuales trabaja este algoritmo (en este caso que el largo del vector es 10).

Tipos de constantes que aparecen típicamente en nuestras aplicaciones:

Nombres de constantes físicas relevantes para los cálculos que realizamos.

Ubicaciones de archivos / nombres de archivos donde el programa leerá la información que necesita para funcionar o donde escribirá los resultados (ya sea una imagen o un archivo con datos).

Constantes o parámetros derivados del algoritmo que estamos implementando.

Por ejemplo luego de la documentación y luego de cargar los paquetes o dependencias que nuestro código tiene, podemos incluir algo como lo siguiente:

```
1
2
3
4  NOut = 14                          #Longitud de la simulacion en [ horas ]
5  UseCmat = True                     #Usar una parametrizacion de la matriz de contactos?
6  CMatPar = 2                        #Parametrizacion matriz contactos (1-contante,2-poblacion mezclada
7  ObsFile = 'syntetic_obs0.csv'      #Archivo con los datos iniciales (observaciones)
8  RequestedObsType = ['TotalCases']  #Tipo de observaciones con el que vamos a trabajar
9  IniTime = 89                       #Tiempo en el cual se inician las simulaciones
10
```

Donde se definen los valores de los parámetros.

Si los parámetros son muchos o necesitamos cambiarlos muy frecuentemente (ej. con cada ejecución del programa) podemos considerar otras opciones:

- archivos de configuración
- parámetros ingresados como argumentos de entrada al momento de la ejecución

Utilizar control de versiones

El versionado o control de versiones consiste en sistemas que nos ayudan a almacenar la forma en la que fue evolucionando un código, permitiendo identificar fácilmente los cambios que ocurrieron en cada actualización. El control de versiones permite:

- Identificar rápidamente cambios que puedan haber afectado el funcionamiento de un código (para bien o para mal).
- Facilitan el trabajo colaborativo, ya que varias personas pueden trabajar simultáneamente en el desarrollo y mantenimiento de un código.
- Facilitan volver a un punto determinado en el pasado en el cual el código funcionaba correctamente.
- Facilitan mantener diferentes versiones del código que pueden servir para probar alternativas (ej. diferentes algoritmos).

Actualmente existe software de código abierto como GIT, CVS o SVN que permiten implementar el control de versiones.

También existen servidores remotos de acceso gratuito como *github* y *gitlab* que ofrecen espacio en la nube para almacenar código y que permiten llevar el versionado del mismo usando git.