Reportes Científicos de la FACEN

Reportes científicos de la FACEN, Enero-junio 2020, 11(1): 51-62

http://doi.org/10.18004/rcfacen2020110151-62

Tópicos Científicos Actuales

Quiero trabajar en bioinformática ¿Cómo empiezo?

I want to work in bioinformatics. How do I start?

Camilo Franco[1], Rocío Riveros Maidana[1], Abdon Troche Rotela[1,2], Mariana Noto[1], Andrea Arrúa Alvarenga[1,3]* & Danilo Fernández Ríos[1]**

[1]Universidad Nacional de Asunción, Facultad de Ciencias Exactas y Naturales. San Lorenzo, Paraguay.
[2]Universidad Nacional de Asunción, Facultad de Politécnica. San Lorenzo, Paraguay.
[3]Universidad Nacional de Asunción, Dirección General de Investigación Científica y Tecnológica, Centro Multidisciplinario de Investigaciones Tecnológicas. San Lorenzo, Paraguay.
Email de correspondencia: *aaarrua@gmail.com; **dfernandez@facen.una.py

Resumen: A medida que los análisis de grandes volúmenes de datos y el uso de técnicas multi-ómicas se están generalizando, la competencia en el pensamiento computacional es una destreza esencial como parte del conjunto de herramientas de un científico que trabaja con datos biológicos. Todos los estudios de "ómicas" requieren biología computacional: la implementación de análisis requiere habilidades de programación, mientras que el diseño experimental y la interpretación requieren una sólida comprensión del enfoque analítico. Aunque los núcleos académicos, los servicios comerciales y las colaboraciones pueden ayudar en la implementación de los análisis, los conocimientos computacionales necesarios para diseñar e interpretar los estudios ómicos no pueden ser reemplazados o complementados. Sin embargo, muchos profesionales de biociencias están exclusivamente entrenados en técnicas experimentales.

Palabrasclave: *ómicas, lenguajes de programación, bash, Python, R, SAS, MatLab, Perl, Fortran, C/C++.*

Abstract: As the analysis of large volumes of data and the use of multi-omics techniques are becoming more widespread, competence in computational thinking is an essential skill as part of the toolkit of a scientist working with biological data. All studies of "omics" require computational biology: implementing analysis requires programming skills, while experimental design and interpretation require a solid understanding of the analytical approach. Although academic nuclei, commercial services, and collaborations can assist in the implementation of analyses, the computational skills needed to design and interpret omics studies cannot be replaced or supplemented. However, many bioscience professionals are exclusively trained in experimental techniques.

Keywords: *omics, programming languages, bash, Python, R, SAS, MatLab, Perl, Fortran, C/C++*

Introducción

El presente trabajo es una adaptación del artículo "Ten simple rules for biologists learning to program" (Carey & Papin, 2018), publicado bajo la licencia de Atribución 4.0 Internacional (Creative Commons, 2018), el cual ofrece una lista de "reglas" útiles para académicos, investigadores y estudiantes de ciencias que deseen embarcarse en el mundo de la biología computacional.

Se realizó la presente adaptación con la intención de disponibilizar en habla hispana las principales recomendaciones para un investigador en ciencias biológicas entrenado de manera tradicional (¡pipeta!), pero interesados en adquirir un conjunto fundamental de habilidades computacionales.

Si no tienes un nivel avanzado de inglés, no te desanimes

Aunque los lenguajes de programación y su documentación correspondiente se desarrollen en inglés, no es imposible aprender a programar sin saber inglés. Existen numerosos recursos para principiantes en español, algunos de los cuales han sido citados en la Tabla 1 de este artículo. Y aunque es innegable que tener habilidades lingüísticas en inglés (más bien en la lectura, para leer la documentación y los foros; y en la comprensión auditiva, para el aprovechamiento de video-tutoriales) es una ventaja a la hora de programar a nivel profesional, también es cierto que si no se poseen estas habilidades se pueden compensar con el uso de traductores, y que a medida que se avanza en las habilidades de

Recibido: 25/02/2020 Aceptado: 07/05/2020

Tabla 1 (comienzo). Una discusión no inclusiva de los lenguajes de programación. Un *shell* es una interfaz de línea de comandos (es decir, de programación) de un sistema operativo, como por ejemplo sistemas operativos similares al Unix. Los lenguajes de programación de bajo nivel se ocupan del hardware de un ordenador. El proceso de pasar de las instrucciones literales del procesador hacia aplicaciones legibles para humanos se llama "abstracción". Los lenguajes de bajo nivel requieren poca abstracción. Los lenguajes interpretados son más rápidos de probar (por ejemplo, para la ejecución de algunas líneas de código); esto facilita el aprendizaje a través de la prueba y el error. Los lenguajes interpretados tienden a ser más legibles para el ser humano. Los lenguajes compilados son poderosos, porque a menudo son más eficientes y se pueden utilizar para tareas de bajo nivel. Sin embargo, la distinción entre lenguajes interpretados y compilados no es siempre rígida. Todos los lenguajes que se presentan a continuación son gratuitos, a menos que se indique lo contrario. La página de Wikipedia sobre lenguajes de programación proporciona una gran visión general y una comparación de lenguajes.

Lenguaje	Características principales	Documentación	Ejemplos de tutoriales	Grupos comunitarios
Bash	*Shell Unix más común. *Práctico para la ejecución de secuencias de comandos escritos en todos los demás lenguajes. *Versátil. *Fácil de eliminar archivos o hacer otros cambios drásticos. *Las debilidades incluyen la ejecución de matemáticas y estructuras de datos limitadas *Por defecto para macOS y la mayoría de las distribuciones de Linux	*http://www.gnu.org/software/bash/manual/	*La guía para principiantes del Proyecto de Documentación de Linux: https://es.wikibooks.org/wiki/El_Manual_de_BASH_Scripting_B%C3%A1sico_para_Principiantes https://thales.cica.es/rd/glinex/practicas-glinex05/manuales/bash/practica.pdf *Documentación de Ubuntu: https://help.ubuntu.com/community/Beginners/BashScripting *Página de GitHub de Azet: https://github.com/azet/community_bash_style_guide	*Página de recursos de la comunidad GitHub: https://github.com/awesome-lists/awesome-bash
Python	*Lenguaje de uso general *Existencia de paquetes para varias tareas *Se considera fácil de aprender debido a su legibilidad *Sintaxis flexible considerada a la vez una fuerza y una debilidad *Lenguaje interpretado	*https://docs.python.org/3/	*Clase Python de Google: https://developers.google.com/edu/python *La guía del autoestopista de Python: https://uniwebsidad.com/libros/python	*Grupo de usuarios de Python: https://wiki.python.org/moin/LocalUserGroups *Grupos de interés especial de Python: https://www.python.org/community/sigs/

Tabla 1 (continuación).

Lenguaje	Características principales	Documentación	Ejemplos de tutoriales	Grupos comunitarios
R	*Alta participación de la comunidad *Existencia de muchos paquetes para varias tareas *Desarrollo centrado en las aplicaciones *Fácil de aprender acoplando programación básica y aplicaciones *Visualización bien desarrollada *Calidad de envase variable *Comunidad de datos ordenada *Lenguaje interpretado	*https://www.rdocumentation.org/ *https://www.r-project.org/ *https://cran.r-project.org/	*R para gatos: https://www.rforcats.net/ *Libros de HadleyWickham: http://hadley.nz/ *Tutorial de introducción a R: https://cran.r-project.org/doc/contrib/rdebuts_es.pdf *Tutorial de Cyclismo R: https://cyclismo.org/tutorial/R/	*R-Ladies: https://rladies.org/ *Grupos de usuarios de R: muchos!
SAS	*Cálculo estadístico *Desarrollo de alta calidad de funciones estadísticas por parte de los desarrolladores comerciales y académicos. *Uso específico del dominio *Gratuito solo para estudiantes *Lenguaje interpretado	*https://support.sas.com/en/support-home.html	*SAS Training for Statistics: http://sct.uab.cat/estadistica/sites/sct.uab.cat.estadistica/files/ManualSAS.PDF	*Grupos de usuarios de SAS: https://www.sas.com/en_us/connect/user-groups.html
MATLAB	*Aplicaciones bien desarrolladas en ingeniería *Mantenido profesionalmente *Lenguaje interpretado *Licencia académica con descuento	*https://www.mathworks.com/help/matlab/	*Tutoriales MATLAB: http://canal.etsin.upm.es/web_cnum/main_matlab.pdf http://webs.ucm.es/centros/cont/descargas/documento11541.pdf	*MATLAB Central: https://www.mathworks.com/matlabcentral/

Tabla 1 (final).

Lenguaje	Características principales	Documentación	Ejemplos de tutoriales	Grupos comunitarios
Perl	*Lenguaje de uso general *Maneja bien el texto *Disminución de la participación de la comunidad *Sintaxis modelada a partir del lenguaje humano *Lenguaje interpretado	*https://www.perl.org/ *https://www.cpan.org/	*Comenzando Perl: http://es.tldp.org/Tutoriales/PERL/tutoperl-print.pdf *Tutorial de Perl maven: https://es.perlmaven.com/perl-tutorial	*Perl Mongers: https://www.pm.org/ *Perl Monks: https://perlmonks.org/
Fortran	*Cálculo numérico *Rápido *A menudo se utiliza para computación de alto rendimiento *Desarrollo limitado *Lenguaje compilado	*http://fortranwiki.org/fortran/show/HomePage	*muchos en la wiki de Fortran: http://pelusa.fis.cinvestav.mx/tmatos/LaSumA/LaSumA2_archivos/Supercomputo/Fortran.pdf https://manual-informatica.com/download-file.html	*Fortran Friends: http://fortran.orpheusweb.co.uk/

programación el lenguaje técnico más utilizado se adquiere por exposición.

Comenzar con el final en mente

Cuando tengas tu objetivo en mente puedes elegir el lenguaje de programación. ¿Quieres ser programador? ¿Quieres desarrollar herramientas bioinformáticas? ¿Quieres implementar herramientas? ¿Quieres que se analicen ya estos datos? Escoge un enfoque y un lenguaje que se adapte a tus metas a largo y corto plazo. Los lenguajes varían en intención y uso. Cada lenguaje y paquete fue creado para resolver un problema particular, por lo que no hay un lenguaje universal "mejor". Elige la herramienta adecuada para el trabajo seleccionando un lenguaje que se adapte bien a las preguntas biológicas que deseas hacer. Si muchas personas en tu campo usan un lenguaje, es probable que funcione bien para los tipos de problemas que encontrarás. Si la gente en tu campo usa una variedad de lenguajes, tienes opciones. Para evaluar la facilidad de uso, considera cuánto apoyo de la comunidad tiene un lenguaje y la cantidad de recursos generados por la comunidad, tales como la prevalencia del desarrollo del usuario, el soporte de paquetes (documentación y tutoriales) y la "presencia" del lenguaje en las páginas de ayuda. En la práctica, las licencias de los lenguajes varían en costo para uso académico y comercial. Los lenguajes libres son más amigables para el trabajo de código abierto (es decir, compartir tus análisis o paquetes). Ve la Tabla 1 para una breve discusión de varios lenguajes de programación, sus características clave y dónde aprender más.

Los pasos pequeños son pasos

Una vez que hayas comenzado, concéntrate en una tarea a la vez y aplica tu pensamiento crítico y tus habilidades para resolver problemas. Esto requiere desglosar un problema en pasos. El análisis de datos ómicos puede sonar difícil, pero los pasos individuales que componen esta tarea no lo son: por ejemplo, leer los datos, decidir cómo interpretar

los valores que faltan, escalar según sea necesario, identificar las condiciones de comparación, dividir para calcular el valor de cambio de expresión de una condición respecto a su referencia (*fold change*), calcular significancia, corregir para pruebas múltiples. Divide un problema grande en tareas pequeñas e implementa una a la vez. Edita iterativamente para lograr eficiencia, flujo y concisión. Los errores ocurrirán. Eso no es un problema; lo que importa es que los encuentres, los corrijas y aprendas de ellos.

La inmersión es la mejor herramienta de aprendizaje

No armes un análisis "caótico" intercalando entre lenguajes y/o entornos. Mientras aprendes, si un trabajo se puede hacer en un solo lenguaje o entorno, hazlo todo allí. Por ejemplo, la importación de una hoja de cálculo de datos (como la que verías en Excel) no es necesariamente sencilla; Excel determina automáticamente cómo leer el texto, pero el método puede diferir de las convenciones en otros lenguajes de programación. Si el proceso de importación "lee mal" tus datos (por ejemplo, celdas en blanco no se leen en blanco o "NA", los números están entre comillas que indican que se leen como texto, o no se mantienen los nombres de las columnas), puede ser tentador volver a Excel para corregirlos con estrategias de búsqueda y reemplazo. Sin embargo, estos problemas pueden solucionarse leyendo correctamente el archivo y entendiendo las estructuras de datos del lenguaje. Como una lengua hablada(Genesee, 1991, 1994), la inmersión es la mejor herramienta de aprendizaje(Campbell & Bolker, 2002; Guzdial, 2004). Además de ralentizar la curva de aprendizaje, la transferencia a través de los programas induce al error. Ver referencias(Boddy, 2016; Linke, 2009; Zeeberg et al., 2004; Ziemann, Eren, & El-Osta, 2016) para errores adicionales inducidos por Excel o procesamiento de textos.

Eventualmente, podrás identificar tareas que no se adaptan bien al lenguaje que utiliza. En ese momento, puede ser útil aprender otro lenguaje para utilizar la herramienta adecuada para el trabajo. De hecho, comprender un lenguaje facilitará el aprendizaje de otro. Hasta entonces, sin embargo, concéntrate en la inmersión para aprender.

Llama a un amigo

Existen numerosos recursos en línea: tutoriales, documentación y sitios web destinados a la comunidad para preguntas y respuestas (StackOverflow, StackExchange, Biostars, etc.), pero nada reemplaza a un amigo o la ayuda de un colega. Encuentra una comunidad de programadores, desde usuarios principiantes hasta experimentados, para pedir ayuda. Es posible que desees buscar tanto apoyo técnico (es decir, un grupo centrado en un lenguaje) y apoyo en relación con una aplicación científica en particular (por ejemplo, un grupo centrado en análisis ómicos). Muchas universidades tienen grupos de computación científica, alojados en la biblioteca o en el departamento de tecnología de la información (TI); estos grupos pueden ser tu punto de partida. Si tu laboratorio o universidad no tiene una comunidad de programadores, búscalos virtual o localmente. Los cursos de Coursera, por ejemplo, tienen paneles de comentarios para que los estudiantes respondan a las preguntas de los demás y aprendan de sus compañeros. Organizaciones como Software y Data Carpentry o grupos de usuarios de lenguajes tienen listas de correos para conectar a los miembros. Muchas ciudades tienen eventos organizados por grupos de usuarios de lenguajes específicos o grupos de interés que se centran en los grandes datos (*big data*), el aprendizaje automático o la visualización de datos. Estos pueden encontrarse a través de meetup.com, grupos de Google, o a través del sitio web de un grupo de usuarios; algunos de ellos se incluyen en la Tabla 1.

Una vez que encuentres una comunidad, pide ayuda. En las etapas iniciales, la ayuda en persona para deconstruir o interpretar una respuesta en línea es invaluable. Además, pídele un código a un amigo. Tú no escribirías un trabajo sin antes leer un montón de trabajos o comenzarías un nuevo proyecto sin seguir a unos cuantos experimentadores. Primero, lee sus códigos. Implementa e interpreta, tratando de entender cada línea. Regresa para discutir tus preguntas. Una vez que empieces a escribir, pide que lo editen.

Aprende A Hacer Preguntas

Hay una respuesta para casi cualquier cosa en línea, pero hay que saber qué pedir para obtener ayuda. Para saber qué preguntar, tienes que entender el problema. Comienza por interpretar un mensaje de error. Debes estar atento a los errores genéricos y aprender de ellos. Identifica qué componente del mensaje de error indica cuál es el problema y qué componente indica dónde está el problema (Figura 1 a Figura 4). Comprender el problema es esencial; este proceso se llama "debugging". Sin comprender realmente el problema, cualquier "solución" propagará e incrementará el error, haciendo más difícil la interpretación de los errores en el futuro. Una vez que entiendas el problema, busca respuestas. La búsqueda de respuestas requiere un googleo efectivo. Aprender el vocabulario (y metavocabulario) del lenguaje y de sus usuarios. Una vez que entiendas el problema y tengas identificado que no existe una solución obvia (y públicamente disponible), pide respuestas en las comunidades de programación (Tabla 1). Al preguntar, parafrasea el problema fundamental. Incluye mensajes de error e información suficiente para reproducir el problema (incluye paquetes, versiones, datos o datos de muestra, código, etc.). Presenta un breve resumen de lo que se hizo, lo que se pretendía, cómo interpretas el problema, qué medidas de resolución de problemas ya se han tomado, y si has buscado la respuesta en otras publicaciones. Véase el siguiente sitio web para sugerencias: http://codereview.stackexchange.com/help/howto-ask y (Collado-Torres, 2017). Termina con un "gracias" y espera a que llegue la ayuda. Es importante revisar los blogs de las páginas web oficiales de los lenguajes de programación que usamos ya que normalmente hay una sección de discusiones o blogs, y si un paquete utilizado está en un repositorio como por ejemplo GitHub, debemos siempre recurrir al *issue tracker* del repositorio.

No reinventes la rueda

Usa todos los recursos disponibles, incluyendo tutoriales en línea, ejemplos en la documentación del lenguaje, códigos publicados, fragmentos de códigos interesantes que tu compañero de laboratorio compartió, y, sí, tu propio trabajo. Lee ampliamente para identificar estos recursos. Copiar-y-pegar(ctrl+x, ctrl+v) es tu amigo. Proporciona

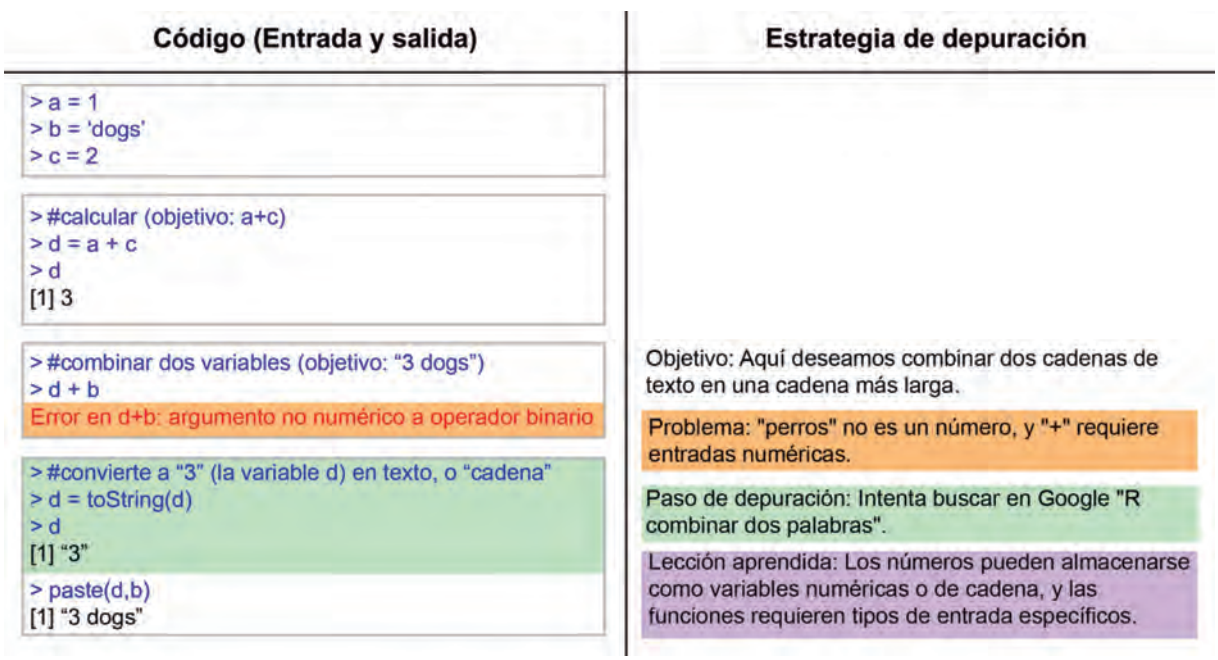


Figura 1. Anatomía de un mensaje de error, Parte 1 (o: Cómo escribir más de una línea de código). Aquí mostramos un ejemplo del proceso de depuración en R utilizando el entorno RStudio, con el objetivo de concatenar dos palabras.

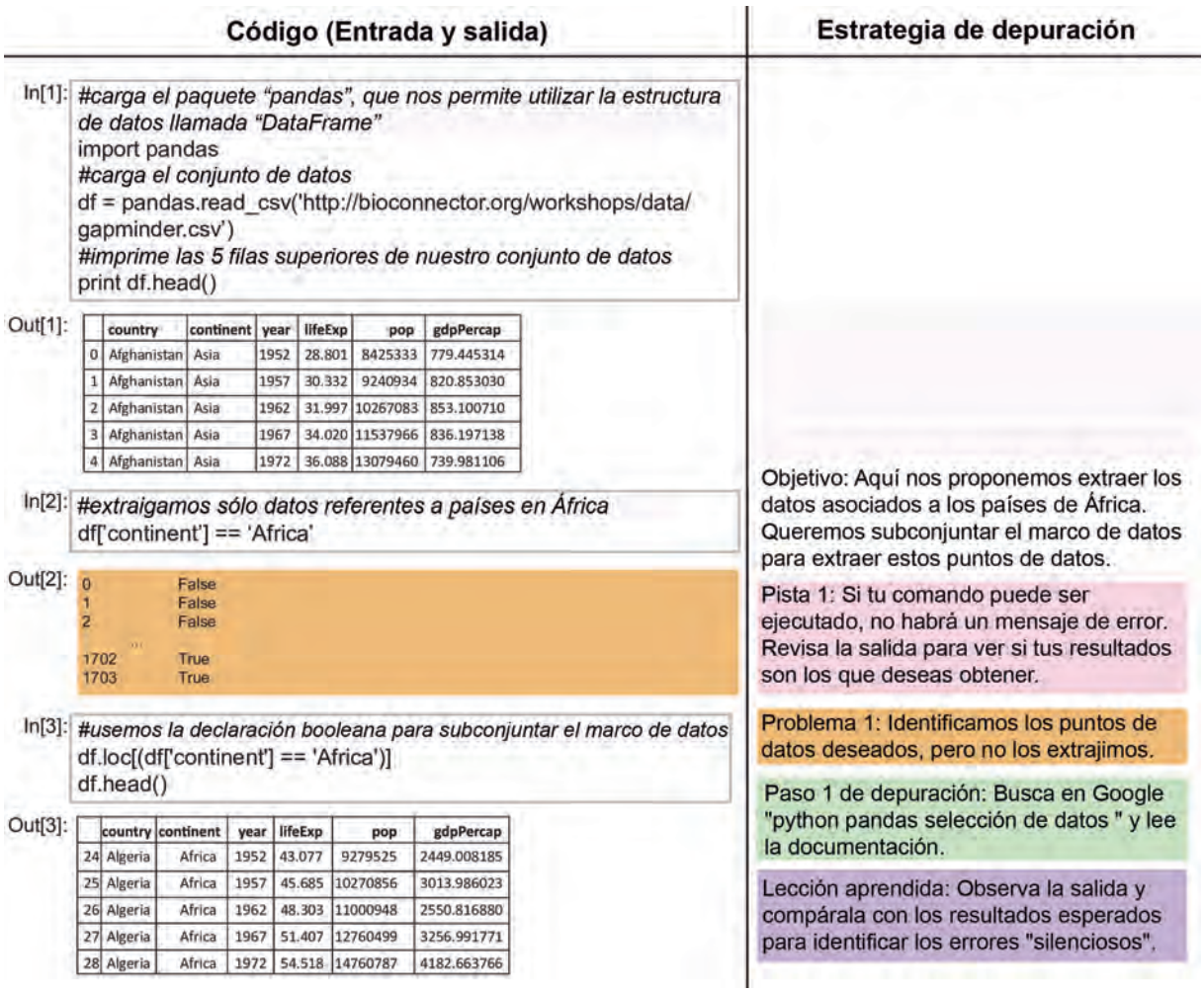


Figura 2. Anatomía de un mensaje de error, Parte 2 (o: Sólo porque funcione, no significa que sea correcto). Aquí le ofrecemos más ejemplos del proceso de depuración. Los ejemplos mostrados se llevan a cabo en Python usando un cuaderno Jupyter. Los entornos como RStudio y los cuadernos Jupyter son dos ejemplos de entornos de desarrollo integrados; estos entornos ofrecen soporte adicional, incluyendo herramientas de depuración incorporadas. Primero, mostramos un error que no induce un mensaje de error, pero que el usuario debe depurar.

crédito si es apropiado (por ejemplo, comentario "adaptado de la secuencia de comandos de...") o necesario (por ejemplo, leer los detalles de las licencias del software). Documenta tus *secuencias de comandos* comentando (usa #) en notas para ti mismo para que puedas usar el código antiguo como plantilla para el trabajo futuro. Estos comentarios te ayudarán a recordar lo que cada línea de código intenta hacer, acelerando tu habilidad para encontrar errores. Los lenguajes incluyen sus métodos de comentar el código que haces, es importante que aprendas esa herramienta antes de que empieces a escribir programas extensos.

Desarrolla buenos hábitos desde el principio

La investigación computacional es investigación, así que utiliza tus mejores prácticas. Esto incluye el mantenimiento de un cuaderno de laboratorio computacional y la documentación de tu código. Un cuaderno de laboratorio computacional es por definición un cuaderno de laboratorio: tu cuaderno de laboratorio incluye protocolos, por lo que tu cua-

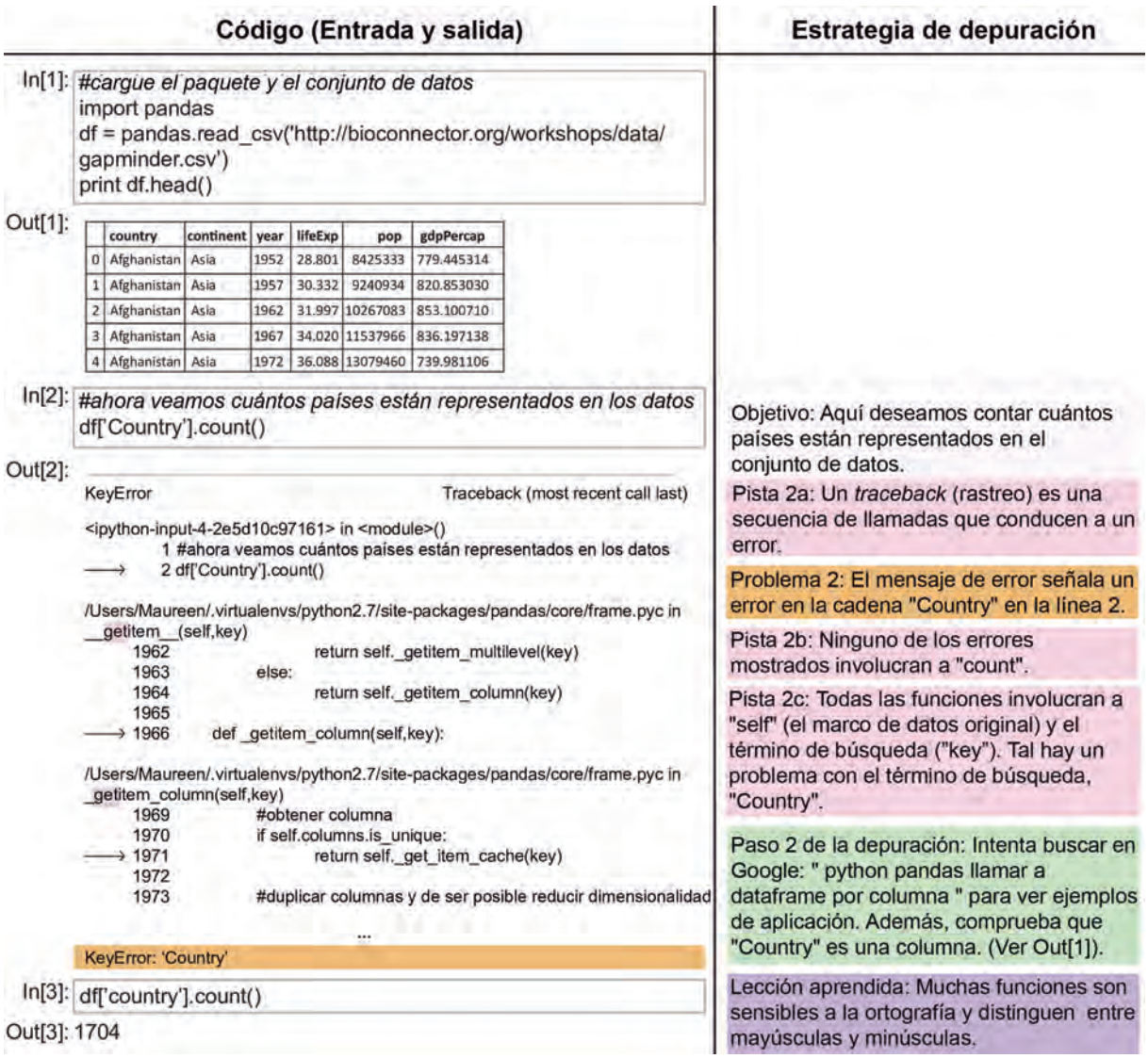


Figura 3. Anatomía de un mensaje de error, Parte 3 (o: Rastree el camino hasta el problema). Aquí mostramos un mensaje de error explícito.

derno de laboratorio computacional también debe incluir protocolos. Los protocolos computacionales son secuencias de comandos, y estos deben incluir el código en sí y cómo acceder a todo lo necesario para implementar el código. Debes incluir también la entrada (datos brutos) y la salida (resultados). Las figuras y la interpretación pueden incluirse si así es cómo organizas tu cuaderno de laboratorio. Desarrolla "buenas prácticas digitales" (estrategias para guardar archivos). Es más fácil organizar un cajón que organizar todo un laboratorio, así que comienza tan pronto como empieces a aprender a programar. Si puedes encontrar ese experimento que hiciste el 12 de junio de 2011 -tu protocolo y resultados- en menos de cinco minutos, deberías ser capaz de encontrar esa figura que se generó para la reunión de laboratorio hace tres semanas, completa con código y datos, en menos de cinco minutos también. Esto requiere un buen control de versiones o documentación de tu trabajo. Como con los protocolos, cada vez que ejecutes una secuencia de comandos, debes tener en cuenta cualquier

Código (Entrada y salida)

In[1]:
```
#carga el paquete y el conjunto de datos
import pandas
df = pandas.read_csv('http://bioconnector.org/workshops/data/
gapminder.csv')
```

In[2]:
```
#veamos si la esperanza de vida media (lifeExp) cambia por año
new_df = df.groupby('year',as_index = False)['lifeExp'].mean()
new_df.head()
```

Out[2]:

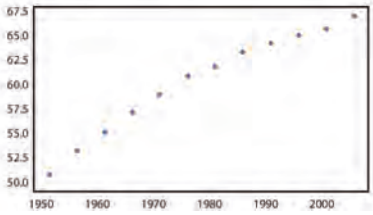
	year	lifeExp
0	1952	49.057620
1	1957	51.507401
2	1962	53.609249
3	1967	55.678290
4	1972	57.647386

In[3]:
```
#Encontramos una solución, pero qué significa? Descompongámosla.
#df.groupby: la función groupby se aplica al dataframe df
#groupby requiere una variable de agrupación ('year')
#Ahora tomamos la media de la columna "lifeExp" de este df agrupado
#pero entonces qué hace "as_index = False"?
#intentemos cambiar "False" a "True" para ver cómo cambian los resultados
new_df = df.groupby('year',as_index = True)['lifeExp'].mean()
new_df.head()
```

Out[3]:
```
year
1952  49.057620
1957  51.507401
1962  53.609249
1967  55.678290
1972  57.647386
Name: lifeExp, dtype: float64
```

In[4]:
```
#si deseamos ambas variables como columnas, usamos "as_index = False"
plot_df = df.groupby('year',as_index = False)['lifeExp'].mean()
#para hacer un gráfico de estos datos, carga un paquete de gráficos.
Especificamos la abreviatura "plt" para acceder a funciones más fácilmente
import matplotlib.pyplot as plt
#construye el gráfico eligiendo el tipo (dispersión) y las variables x e y
plt.scatter(x = plot_df['year'], y = plot_df['lifeExp'])
plt.show()
```

Out[4]:



Estrategia de depuración

Objetivo: Aquí deseamos encontrar la esperanza de vida media cada año. Encontramos el código buscando en Google "python pandas hallar media de grupos", pero no entendemos la línea de código que da este resultado. Entonces, depuraremos la línea para entenderla.

Paso 3 de la depuración: Descompone cada componente del código en conocidos y desconocidos. Usa la documentación para entender lo desconocido.

Pista 3b: El índice del marco de datos es una etiqueta de eje. Intenta googlear "python pandas índice" para saber más. La nueva salida está bien, a menos que quieras ambas variables (año y esperanza de vida media) como columnas. Queremos cada variable en una columna para facilitar el trazado.

Lección aprendida: Depura para entender la solución, no sólo el problema.

Figura 4. Anatomía de un mensaje de error, Parte 4 (o: Depuración de una solución). Por último, mostramos cómo depurar una solución para entender una línea de código que se encuentra en Internet.

modificación que se realice. Documenta todos los cambios en los protocolos experimentales y computacionales. Estos hábitos te harán más eficiente al mejorar la reproducibilidad de tu trabajo. Para consejos específicos, véanse Perez-Riverol et al., 2016; Sandve, Nekrutenko, Taylor, & Hovig, 2013; Schnell, 2015.

La práctica hace al maestro

Utiliza conjuntos de datos ficticios para practicar un

problema o análisis. Los datos biológicos rápidamente se vuelven grandes. Es difícil de encontrar la aguja en un pajar computacional, así que prepárate para tener éxito practicando en un ambiente controlado con ejemplos más sencillos. Genera pequeños conjuntos de datos ficticios que utilizan la misma estructura que sus datos. Haz que los datos sean lo suficientemente simples como para predecir cómo los números, texto, etc., deberían reaccionar en su análisis. Haz pruebas para asegurarte de que reaccionan como se espera. Esto te ayudará a comprender lo que se está haciendo en cada paso y solucionar los errores, preparándolo para ampliarlo a grandes e impredecibles conjuntos de datos. Utiliza estos conjuntos de datos para probar su enfoque, su implementación, y su interpretación. Los conjuntos de datos ficticios son su control negativo, lo que le permite diferenciar entre los resultados negativos y el fracaso de la simulación.

Sé autodidacta

¿Cómo te enseñarías si fueras otra persona? Te enseñarías con un poco más de paciencia y un poco más de empatía de lo que estás practicando ahora. No estás solo en tu frustración ocasional (Figura 5). El aprendizaje lleva tiempo, así que planifica adecuadamente. Los cursos introductorios son útiles para aprender lo básico porque lo básico es fácil de descuidar en el auto-estudio. Expresa expectativas claras para ti mismo y referencias para el éxito. Aplica parte de la estructura (plazos, tareas, etc.) que proporcionarías a un estudiante para ayudar a motivar y evaluar tu progreso. Si algo no está funcionando, ajústalo; no todos aprenden mejor con un solo enfoque. Explora tutoriales, clases en línea, talleres, libros como *Practical Computing for Biologists* (Haddock & Dunn, 2011), reuniones de programación local, etc., para encontrar tu enfoque preferido.

Sólo hazlo

Empieza a programar. No puedes editar una página en blanco. Aprender a programar puede ser intimidante. El poder y la libertad que proporciona la

Figura 5. "¿Cómo salir del editor de vim?" (o: ¡Todos nos quedamos atascados en algún momento!). Véase: http://stackoverflow.com/questions/11828270/how-to-exit-the-vim-editor.

realización de tus propios análisis computacionales aportan muchos puntos de decisión, y cada decisión aporta más espacio para los errores. Además, la evaluación de su trabajo es menos en blanco y negro que en algunos experimentos. Sin embargo, la programación tiene la ventaja de que el fallo está libre de riesgos. No se desperdician recursos, ni dinero, ni tiempo (¡el trabajo de un estudiante es aprender!), ni una reputación científica. En síntesis, el campo de juego es nivelado por el trabajo duro y el esmero. Así que, mientras que la programación puede ser intimidante, el paso más intimidante es empezar.

Conclusiones

Markowetz escribió recientemente, "Los biólogos computacionales son sólo biólogos que usan una herramienta diferente"(Markowetz, 2017). Si eres un biólogo "tradicionalmente entrenado", pretendemos que estas recomendaciones simples sirvan como instrucción (y charla motivacional) para aprender una herramienta nueva, poderosa y emocionante. La curva de aprendizaje puede ser empinada; sin embargo, el esfuerzo dará sus frutos. La experiencia computacional te hará más valorizado en el mercado como científico(Bergman, 2017). La investigación computacional tiene menos gastos generales y reduce la barrera de entrada en los campos en transición(Kwok, 2013), abriendo puertas de la carrera a los investigadores interesados. Tal vez lo más importante es que las habilidades de programación te permitirán implementar e interpretar mejor tus propios análisis y comprender y respetar los sesgos analíticos, lo que también le permitirá ser un mejor experimentalista. Por lo tanto, el tiempo que pasas en tu computadora es valioso. Adquirir experiencia en programación te convertirá en un mejor profesional de biociencias.

Literatura citada

Bergman, C. (2017). An Assembly of Fragments. Retrieved December 30, 2019, from https://caseybergman.wordpress.com/2012/07/31/top-n-reasons-to-do-a-ph-d-or-post-doc-in-bioinformaticscomputational-biology/

Boddy, J. (2016). One in five genetics papers contains errors thanks to Microsoft Excel. Retrieved February 14, 2020, from Science website: https://www.sciencemag.org/news/2016/08/one-five-genetics-papers-contains-errors-thanks-microsoft-excel?utm_source=newsfromscience&utm_medium=facebook-text&utm_campaign=excel-6929

Campbell, W., & Bolker, E. (2002). *Teaching programming by immersion, reading and writing* (W. Campbell & E. Bolker, Eds.). IEEE.

Carey, M. A., & Papin, J. A. (2018). Ten simple rules for biologists learning to program. *PLOS Computational Biology*, *14*(1), e1005871. https://doi.org/10.1371/journal.pcbi.1005871

Collado-Torres, L. (2017). Recent Posts. Retrieved December 30, 2019, from http://lcolladotor.github.io/

Creative Commons. (2018). Atribución 4.0 Internacional (CC BY 4.0). Retrieved February 14, 2020, from Creative Commons website: https://creativecommons.org/licenses/by/4.0/deed.es

Genesee, F. (1991). *Second language learning in school settings: Lessons from immersion* (F. Genesee, Ed.). Lawrence Erlbaum Associates.

Genesee, F. (1994). *Integrating language and content: Lessons from immersion*. Center for Research on Education, Diversity & Excellence.

Guzdial, M. (2004). Programming environments for novices. *Computer Science Education Research*, *2004*, 127–154.

Haddock, S. H. D., & Dunn, C. W. (2011). *Practical computing for biologists*. Sunderland, MA: Sinauer Associates.

Kwok, R. (2013). Computing: Out of the hood. *Nature*, *504*(7479), 319–321. https://doi.org/10.1038/nj7479-319a

Linke, D. (2009). Commentary: Never trust your word processor. *Biochemistry and Molecular Biology Education*, *37*(6), 377–377. https://

doi.org/10.1002/bmb.20340

Markowetz, F. (2017). All biology is computational biology. *PLOS Biology*, *15*(3), e2002050. https://doi.org/10.1371/journal.pbio.2002050

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. da V., … Vizcaíno, J. A. (2016). Ten Simple Rules for Taking Advantage of Git and GitHub. *PLOS Computational Biology*, *12*(7), e1004947. https://doi.org/10.1371/journal.pcbi.1004947

Sandve, G. K., Nekrutenko, A., Taylor, J., & Hovig, E. (2013). Ten Simple Rules for Reproducible Computational Research. *PLoS Computational Biology*, *9*(10), e1003285. https://doi.org/10.1371/journal.pcbi.1003285

Schnell, S. (2015). Ten Simple Rules for a Computational Biologist's Laboratory Notebook. *PLOS Computational Biology*, *11*(9), e1004385. https://doi.org/10.1371/journal.pcbi.1004385

Zeeberg, B. B. R., Riss, J., Kane, D. D. W., Bussey, K. J. K., Uchio, E., Linehan, W. M., … Weinstein, J. N. (2004). Mistaken Identifiers: Gene name errors can be introduced inadvertently when using Excel in bioinformatics. *BMC Bioinformatics*, *5*(1), 80. https://doi.org/10.1186/1471-2105-5-80

Ziemann, M., Eren, Y., & El-Osta, A. (2016). Gene name errors are widespread in the scientific literature. *Genome Biology*, *17*(1), 177. https://doi.org/10.1186/s13059-016-1044-7