Traducción.

Traducción de Phython programming for the Humanities.

María Gimena del Rio Riande y Frank Fischer.

Cita:

María Gimena del Rio Riande y Frank Fischer (2015). *Traducción de Phython programming for the Humanities*. Traducción.

Dirección estable: https://www.aacademica.org/gimena.delrio.riande/66

ARK: https://n2t.net/ark:/13683/pdea/Syg

Capítulo 1: Empezando

-- Un curso de Python para las Humanidades, por Folgert Karsdorp y Maarten van Gompel

-- Traducido al castellano por Gimena del Río y Frank Fischer (cap. 1, diciembre, 2014)

In [ ]:

```
print("En sus marcas, listos, ya!")
```

Todos podemos aprender a programar y el mejor modo de aprender es haciéndolo. En este tutorial te pedimos escribir mucha cantidad de código. Hacé clic en cualquier bloque de código, como el de más arriba y presioná CTRL+ENTER a la vez para ejecutarlo. Empecemos, entonces, a escribir un pequeño primer programa.

Quiz!

En la caja de código más abajo, escribí un pequeño programa que calcule cuántos minutos hay en siete semanas. (El signo x (por) se representa con *.) Líneas que empiezan con el signo numeral (#) son notas que no son interpredadas por Python.

In [ ]:

```
# insert your code here
```

Bien! Has escrito tu primer pequeño programa sin ninguna ayuda! Entonces, ¿podemos ir más allá ahora usando nuestro lenguaje de programación como una calculadora? Antes de pedirte que programes un poco más, te explicaremos algo sobre assignment o asignación.

Podemos asignarles valores a las variables usando el operador =. La variable es solo un nombre que le damos a un valor, podemos imaginarla como un casillero donde ponemos un valor y donde escribimos un nombre. El próximo ejercicio contiene dos operaciones. Primero, asignamos el valor de 2 al nombre x. Después de hacer eso x mantendrá el valor de 2. Podemos decir que Python guardó el valor de 2 en x. Finalmente, escribimos el valor usando, como hicimos más arriba, el comando print().

In [ ]:

```
x = 2

print(x)
```

Ahora que hemos asignado y guardado el valor de 2 para x, podemos usar la variable x para hacer cosas como las siguientes:

In [ ]:

```
print(x * x)

print(x == x)

print(x > 6)
```

¿Qué sucede en estos casos?

Pero las variables no son solo números. También pueden ser textos (cadenas de caracteres). Por ejemplo:

In [ ]:

```
book = "The Lord of the Flies"

print(book)
```

Un texto o cadena de caracteres en Python debe ir entre comillas (simples o dobles). Sin esas comillas Python supone que está tratando con variables que fueron definidas antes de comenzar a programar. book es la variable a la que se le asigna la cadena "The Lord of the Flies", pero esa cadena no es una variable sino un valor.

Los nombres de las variables pueden ser elegidos arbitrariamente. Podemos elegir el nombre que gustemos. Es, de todos modos, recomendable que usemos nombres con un cierto sentido, ya que usaremos directamente los nombres de las variables y no los valores que representan.

In [ ]:

```
# not recommended...

banana = "The Lord of the Flies"

print(banana)
```

Podés usar el nombre banana para dar cuenta del título "The Lord of the Flies", pero convengamos en que este no es transparente en su significado.

Las variables pueden variar y podemos actualizarlas. Podemos contar cuántos libros hay en nuestra biblioteca:

In [ ]:

```
number_of_books = 100
```

Así, cuando nos hacemos de un nuevo libro, podemos actualizar su número:

In [ ]:

```
number_of_books = number_of_books + 1

print(number_of_books)
```

Actualizaciones como estas surgen constantemente. Por ello, Python provee atajos y podés escribir lo mismo usando +=:

In [ ]:

```
number_of_books += 5

print(number_of_books)
```

Por el momento, la cuestión más interesante que podemos mencionar sobre las variables es que podemos asignar el valor de una variable a otra variable. Explicaremos mejor esto más tarde, pero es importante que comprendas el mecanismo básico de funcionamiento. Antes del próximo ejercicio, ¿podés predecir que es lo que Python va a escribir?

In [ ]:

```
book = "The Lord of the Flies"

reading = book

print(reading)
```

Quiz!

Ahora que comprendés cómo asignar valores a las variables, llegó el momento de volver a programar. Queremos que escribas algo de código que defina una variable, name, y le asignes una cadena que sea tu nombre. Si tu nombre tiene menos de cinco caracteres, usá tu apellido. Si tu apellido tiene menos de cinco caracteres, usa ambos.

In [ ]:

insert your code here

print(name)

Qué aprendimos

Para terminar esta sección, he aquí un repaso de los conceptos que aprendiste. Revisá la lista y asegurate que comprendés todos los conceptos.

variable

valor

asignarles valor a las variables

diferencia entre variable y valor

cadena de caracteres

variables variables

Manipulación de las cadenas de caracteres

Muchas disciplinas humanísticas trabajan con textos. Por ello, la programación para las Humanidades se enfoca principalmente al trabajo sobre los textos. En el último ejercicio se te pidió que definieras una variable que diera cuenta de una cadena que represente tu nombre. También vimos algo de aritmética en nuestro primer ejercicio. No solo números, pero también cadenas de caracteres pueden ser añadidos o, más precisamente, concatenados:

In [ ]:

book = "The Lord of the Flies"

print(name + " likes " + book + "?")

La cadena consiste en un número de caracteres. Podemos individualizar los caracteres con la ayuda de indexing (indizar). Por ejemplo, para encontrar solo la primera letra de tu nombre podés tipear:

In [ ]:

first_letter = name[0]

print(first_letter)

Tené en cuenta que para identificar la primera letra usamos el número 0. Esto puede parecer extraño pero recordá que las listas/índices/relaciones en Python empiezan en cero.

Quiz!

Ahora, si sabés cuántas letras tiene tu nombre, podés preguntar por la última letra de tu nombre:

In [ ]:

```
last_letter = name[# fill in the last index of your name (tip indexes start at 0)]

print(last_letter)
```

No resulta demasiado práctico tener que contar los caracteres de nuestras cadenas si queremos saber cuál es la última letra de estas. Python tiene un modo simple de acceder a una cadena desde el final:

In [ ]:

```
last_letter = name[-1]

print(last_letter)
```

Hay otra alternativa: la función len(), que devuelve la medida de una cadena:

In [ ]:

```
print(len(name))
```

¿Entendés lo siguiente?

In [ ]:

```
print(name[len(name)-1])
```

Quiz!

Ahora, ¿podés escribir el código que defina la variable but_last_letter y asignárselo a la penúltima letra de tu nombre?

In [ ]:

```
but_last_letter = # insert your code here

print(but_last_letter)
```

Te estás conviertiendo en un/a experto/a en indizar cadenas. Ahora, ¿qué pasaría si quisiésemos averiguar cuáles son las dos últimas letras de tu nombre? En Python podemos usar las llamadas

porciones de listas/índices (slice-indexes). Para encontrar las dos primeras letras de tu nombre escribí:

In [ ]:

```
first_two_letters = name[0:2]

print(first_two_letters)
```

El índice 0 es opcional, por eso podemos tan solo escribir name[:2]. Esto es: 'Tomá todos los caracteres del nombre hasta que llegues al índice 2.' También podemos empezar en el 2 y dejar el índice final sin especificar:

In [ ]:

```
without_first_two_letters = name[2:]
```

Dado que no especificamos el índice final, Python continuará hasta que encuentre el final de nuestra cadena. Si quisiésemos encontrar cuáles son las últimas dos letras de nuestro nombre, deberíamos escribir:

In [ ]:

```
last_two_letters = name[-2:]

print(last_two_letters)
```

Mirá esta imagen. ¿Entendés de qué se trata?

Quiz!

¿Podés definir la variable middle_letters y asignársela a todas las letras de tu nombre exceptuando las primeras dos y las últimas dos?

In [ ]:

```
middle_letters = # insert your code here

print(middle_letters)
```

Si te dan estas dos palabras, ¿podés escribir el código para formar la palabra humanities usando solo porciones de cadena y concatenación? (Es decir, no se permite escribir palabras).

In [ ]:

```
word1 = "human"

word2 = "opportunities"

# insert your code here
```

Qué aprendimos

Para finalizar esta sección, he aquí un repaso de lo que aprendimos. Revisá la lista y asegurate de que entendés todos los conceptos:

concatenación (por ej., suma de cadenas)

indizar

dividir (en porciones)

len()

Listas

Prestá atención a la frase:

In [ ]:

```
sentence = "Python's name is derived from the television series Monty Python's Flying Circus."
```

Las palabras están hechas de caracteres, y también las cadenas de objetos en Python. Como veremos, es siempre preferible representar nuestros datos de la forma más natural posible. En cuanto a la oración de más arriba, parece más natural describirla en términos de palabras que de caracteres. Digamos que queremos acceder a la primera palabra de nuestra oración. A ver qué pasa si escribimos eso:

In [ ]:

```
first_word = sentence[0]

print(first_word)
```

Python solo devuelve la primera letra de la oración. Podemos transformar la oración en una lista (list) de palabras (representada por cadenas) usando la función split() del siguiente modo:

In [ ]:

words = sentence.split()

print(words)

Al añadir la función split (dividir) en nuestra oración, Python divide la oración en espacios y devuelve una lista de palabras. En muchos casos la lista funciona como una cadena. Podemos acceder a todos sus componentes usando índices y podemos usar porciones de índices para acceder a partes de la lista. Probemos:

Quiz!

Escribí un pequeño programa que defina la variable first_word (primera palabra) y asignale la primera palabra de nuestra lista de palabras.

In [ ]:

first_word = # insert your code here

print(first_word)

Una lista (list) actúa como un contenedor donde podemos almacenar todo tipo de información. Podemos acceder a una lista usando índices y porciones. Podemos también añadir nuevos elementos a la lista. Para eso usamos el método append (añadir). Veamos cómo funciona. Si quisiésemos, por ejemplo, hacer una lista con nuestras lecturas favoritas empezaríamos con una lista vacía y le añadiríamos nuestros libros favoritos:

In [ ]:

#start with an empty list

good_reads = []

good_reads.append("The Hunger games")

good_reads.append("A Clockwork Orange")

print(good_reads)

Si, por alguna razón, no nos gustase más un libro, podríamos cambiar la lista del siguiente modo:

In [ ]:

good_reads[0] = "Pride and Prejudice"

print(good_reads)

Quiz!

Tratá de cambiar el título del segundo libro de nuestra colección.

In [ ]:

```
# insert your code here

print(good_reads)
```

Cambiamos un elemento en la lista. Notá que si hacés lo mismo para una cadena obtendrías un error:

In [ ]:

```
name = "Pythen"

name[4] = "o"
```

Esto es porque las cadenas (strings) y otros tipos son inmutables. Es decir, no pueden ser modificados, en oposición a las lists (listas) que son mutables. Exploremos otras formas en las que podemos manipular listas.

remove()

Supongamos que nuestra colección de libros favoritos creció demasiado y quisiésemos quitar libros de nuestra lista. Python provee del método remove (remover), que trabaja sobre una lista y quita de ella los elementos que ya no queremos.

In [ ]:

```
good_reads = ["The Hunger games", "A Clockwork Orange",
        "Pride and Prejudice", "Water for Elephants",
        "The Shadow of the Wind", "Bel Canto"]


good_reads.remove("Water for Elephants")


print(good_reads)
```

Si tratamos que quitar un libro que no está en la lista, Python da error.

In [ ]:

good_reads.remove("White Oleander")

Quiz!

Definí la variable good_reads como una lista vacía. Ahora añadí algunos de tus libros favoritos a ella (al menos tres) y hacé que Python escriba los últimos dos libros de tu lista.

In [ ]:

insert your code here

Tal como en las cadenas, podemos unir dos listas. He aquí un ejemplo:

In [ ]:

#first we specify two lists of strings:

good_reads = ["The Hunger games", "A Clockwork Orange",

"Pride and Prejudice", "Water for Elephants",

"The Shadow of the Wind", "Bel Canto"]

bad_reads = ["Fifty Shades of Grey", "Twilight"]

all_reads = good_reads + bad_reads

print(all_reads)

sort()

Siempre es conveniente organizar la biblioteca. Podemos ordenar nuestros libros con la siguiente expresión:

In [ ]:

good_reads.sort()

print(good_reads)

nested lists

Hasta aquí nuestras listas solo estuvieron constituidas por cadenas. De todos modos, una lista puede contener diferentes tipos de datos, como números enteros y también listas. ¿Entendés lo que pasa en este ejemplo?

In [ ]:

```
nested_list = [[1, 2, 3, 4], [5, 6, 7, 8]]

print(nested_list[0])

print(nested_list[0][0])
```

Podemos usar esto para mejorar nuestra lista de libros favoritos otorgando un número (a modo de calificación o nota numérica) a cada uno de ellos. Una entrada en nuestra colección consistirá de una nota entre 1 y 10 y el título de nuestro libro. El primer elemento es el título, el segundo es la nota: [title, score]. Empezamos con una lista vacía:

In [ ]:

```
good_reads = []
```

Y le agregamos dos libros:

In [ ]:

```
good_reads.append(["Pride and Prejudice", 8])

good_reads.append(["A Clockwork Orange", 9])
```

Quiz!

Actualizá la lista de libros favoritos (good_reads) con algunos de tus libros y otorgales una nota. ¿Podés escribir la nota que le diste al primer libro de tu lista?

In [ ]:

```
# insert your code here
```

Qué aprendimos

Para finalizar esta sección, he aquí una revisión de los nuevos conceptos y funciones que aprendiste:

lista

mutable versus inmutable

.split() (dividir)

.append() (añadir)

nested lists (listas anidadas)

.remove() (quitar)

.sort() (ordenar)

Diccionarios

Nuestra pequeña lista de libros favoritos ha empezado a crecer y podemos manipularla de muchas formas. Ahora, imaginemos que nuestra lista es extensa y que queremos buscar la nota que le dimos a un libro. ¿Cómo hacemos para encontrarlo? Para ello, Python provee de una estructura de datos denominada diccionario (dictionary). Un diccionario es similar a los diccionarios que tenemos en casa. Este consiste de entradas, o claves, que tienen un valor. Definamos una:

In [ ]:

```
my_dict = {"book": "physical objects consisting of a number of pages bound together",
        "sword": "a cutting or thrusting weapon that has a long metal blade",
        "pie": "dish baked in pastry-lined pan often with a pastry top"}
```

Revisá esta nueva sintaxis. Prestá atención a los paréntesis curvos y dos puntos. Las claves se encuentran a la izquierda de los dos puntos; los valores al lado derecho. Para encontrar el valor de una clave, indizamos el diccionario usando esa clave:

In [ ]:

```
description = my_dict["sword"]
print(description)
```

Decimos que indizamos porque usamos la misma sintaxis con corchetes cuando indizamos listas o cadenas. La diferencia estriba en que no usamos un número para indizar un diccionario sino una clave. Al igual que las listas, los diccionarios son mutales, lo que significa que podemos añadir o quitar entradas de él. Definamos un diccionario vacío y añadámosle algunos libros. Los títulos serán nuestras entradas y las calificaciones sus valores. Esta es la sintaxis de una entrada:

In [ ]:

```
good_reads = {}
good_reads["Pride and Prejudice"] = 8
good_reads["A Clockwork Orange"] = 9
```

Esto es, de algún modo, similar a lo que vimos antes cuando modificamos nuestra lista de libros. Allí indizamos la lista usando un número entero para individualizar un libro en particular. Aquí directamente usamos el título del libro. ¿Por qué te parece que esto es útil?

Quiz!

Actualizá la estructura de tu lista con tus propios libros. Tratá de escribir la calificación que le diste a uno de los libros.

In [ ]:

insert your code here

keys(), values()

Para recuperar una lista de todos los libros de nuestra colección podemos pedirle al diccionario que nos devuelva todas las claves en la lista:

In [ ]:

good_reads.keys()

De un modo similar también podemos pedirle los valores:

In [ ]:

good_reads.values()

Qué aprendimos

Para finalizar esta sección, he aquí un repaso de los nuevos conceptos y funciones que aprendiste:

diccionario

indizar o individualizar entradas de diccionarios

añadir entradas a un diccionario

.keys() (claves)

.values() (valores)

Condicionales

Sentencias condicionales simples

Gran parte de la actividad de programar se relaciona con la ejecución de ciertos trozos de código si una condición particular cumple. Ya hemos visto dos de estas condicionales al principio del capítulo. Aquí te ofreceremos una sucinta introducción. ¿Podés descubrir lo qué hacen los siguientes condicionales?

In [ ]:

```
print("2 < 5 =", 2 < 5)

print("3 > 7 =", 3 >= 7)

print("3 == 4 =", 3 == 4)

print("school == homework =", "school" == "homework")

print("Python != perl =", "Python" != "perl")
```

if, elif y else

El diccionario es una mejor estructura para nuestra colección de libros favoritos. De todos modos, hasta a través de diccionarios podemos olvidarnos de cuáles son los libros que hemos añadido a la colección. ¿Qué pasaría si tratamos de obtener la calificación de un libro que no está en nuestra colección?

In [ ]:

```
good_reads["El silenciero"]
```

Obtenemos un error. Un KeyError, que básicamente significa que "la clave por la que me has preguntado no figura en el diccionario". Vamos a aprender más sobre como manejar errores más tarde, pero por ahora, nos gustaría advertir a nuestro programa para que no lo haga en primer lugar. Escribamos un pequeño programa que dé como resultado "X está en la colección" si un libro en particular está en la colección y "X no está en la colección" si no lo está.

In [ ]:

```
book = "A Clockwork Orange"

if book in good_reads:
    print(book + " is in the collection")
else:
    print(book + " is NOT in the collection")
```

Hasta aquí mucha nueva sintaxis. Vayamos paso a paso. Primero tenemos que preguntar si el valor que le hemos asignado a libro (book) está en nuestra colección. El segmento posterior a if evalúa si es verdadero o falso (True o False). Escribamos esto:

In [ ]:

```
"El silenciero" in good_reads
```

Como este libro no está en la colección, Python nos devuelve False. Hagamos lo mismo para un libro que sabemos que está en nuestra colección:

In [ ]:

```
"A Clockwork Orange" in good_reads
```

Si la expresión después de if evalúa algo como verdadero (True), nuestro programa irá a la próxima línea y escribirá book + " is in the collection". Veamos:

In [ ]:

```
if "A Clockwork Orange" in good_reads:
    print(book + " is in the collection")
```

In [ ]:

```
if "El silenciero" in good_reads:
    print(book + " is in the collection")
```

Tené en cuenta que el contenido del último bloque de código no fue ejecutado. Esto es así porque el valor que le asignamos a book no está en nuestra colección y, consecuentemente, el segmento después de if no fue considerado como True. En nuestro pequeño programa de más arriba usamos otro comando al lado de if, else. No debería ser demasiado difícil de comprender qué está sucendiendo aquí. El segmento después de else será ejecutado si if es evaluado como False. Esto es: Si el libro no está en la colección, escribí que no está.

Indentación!

Antes de continuar, debemos explicarte que la disposición de nuestro código no es opcional. De un modo diferente que en otros lenguajes, Python no utiliza llaves para marcar el comienzo y el final de las expresiones. La única delimitación son los dos puntos (:) y el espaciado del código. Este debe ser usado consistentemente en la escritura del código. Por convención se usan cuatro espacios. Esto quiere decir que cada vez que usas dos puntos (por ejemplo, cuando usamos if), la próxima línea debe ser sangrada con cuatro espacios más que en la línea previa.

A veces tenemos varios condicionales que deben ser evaluados de modo diferente. Para ello, Python nos provee del comando elif, que puede usarse de modo similar a if y else. Notá que, de todos modos, ¡solo podés usar elif después de if! Más arriba te preguntamos si un libro estaba en la colección. Podemos hacer lo mismo para partes de las cadenas de nuestros elementos en la lista. Por ejemplo, podemos evaluar si la letra a está en la palabra banana:

In [ ]:

```
"a" in "banana"
```

Así, esto sería evaluado como False:

In [ ]:

```
"z" in "banana"
```

Usemos esto en una combinación de if-elif-else:

In [ ]:

```
word = "rocket science"

if "a" in word:
    print(word + " contains the letter a")
elif "s" in word:
    print(word + " contains the letter s")
else:
    print("What a weird word!")
```

Quiz!

Pongamos en práctica nuestros nuevos conocimientos. Escribe un pequeño programa que defina la variable weight (peso). Si peso es > 50 libras (22 kilos aprox.) ejecuta "Hay un recargo de 25$ por equipaje de ese peso". Si no es mayor, ejecuta: "Gracias". Cambia el valor de peso en ambos ejemplos (usa los comandos < o >).

In [ ]:

```
# insert your code here
```

and, or, not

Hasta ahora nuestros condicionales consistieron de expresiones simples. De todos modos, a menudo necesitamos de condicionales complejos para ejecutar cierto código. Python provee de un número de maneras para hacer eso. La primera es con el comando and. Nos permite yuxtaponer dos expresiones que necesitan ser verdaderas para hacer que la totalidad de lo expresado será verdadero (True). Veamos cómo funciona:

In [ ]:

```
word = "banana"

if "a" in word and "b" in word:
    print("Both a and b are in " + word)
```

Si una de las expresiones es entendida como falsa, nada se ejecutará:

In [ ]:

```
if "a" in word and "z" in word:
    print("Both a and z are in " + word)
```

Quiz!

Reemplazá and por or en esta expresión con if. ¿Qué sucede?

In [ ]:

```
word = "banana"

if "a" in word and "z" in word:
    print("Both a and b are in " + word)
```

En el bloque de código de abajo, ¿podés añadir una expresión con else que ejecute que ninguna de las letras fue encontrada?

In [ ]:

```
if "a" in word and "z" in word:
    print("Both a and z are in " + word)

# insert your code here
```

Finalmente, podemos usar not para probar condicionales que no son verdaderos.

In [ ]:

```
if "z" not in word:
    print("z is not in " + word)
```

Objetos como cadenas o listas son verdaderos (True) porque existen. Cadenas, listas, diccionarios vacíos etc. son, por otra parte, falsos (False) porque, de algún modo, no existen. Podemos usar este principio para, por ejemplo, ejecutar únicamente un segmento de código si una determinada lista contiene un valor:

In [ ]:

```
numbers = [1, 2, 3, 4]
if numbers:
    print("I found some numbers!")
```

Ahora, si la lista está vacía, Python no devolverá nada:

In [ ]:

```
numbers = []
if numbers:
    print("I found some numbers!")
```

Quiz!

¿Podés escribir un código que ejecute "Esta es una lista vacía", si la lista provista no contiene ningún valor?

In [ ]:

```
numbers = []
# insert your code here
```

¿Podés hacer lo mismo pero usando el comando len() (ingl. 'length', esp. 'extensión', 'longitud')?

In [ ]:

```
numbers = []
```

insert your code here

Qué aprendimos

Para finalizar esta sección, he aquí un repaso de las nuevas funciones, expresiones y conceptos que hemos aprendido. Revisalos y asegurate que comprendés su sentido y cómo se usan.

condicionales

espaciar

if (si)

elif

else

True (verdadero)

False (falso)

objetos vacíos son falsos

not (no)

in (en)

and (y)

or (o)

condicionales múltiples

==

<

>

!=

KeyError

Bucles

La programación es más útil si podemos ejecutar una cierta acción en un número extenso de elementos. Por ejemplo, si nos dan una determinada lista de palabras, podemos querer saber la longitud de todas las palabras, no solo de una. Ahora podrías hacer esto yendo por vez a través de todos los índices de nuestras listas de palabras, tomando tantas líneas de código como índices. De todos modos, esto es algo incómodo.

Python provee de los llamados comandos for que nos permiten iterar cualquier objeto que se pueda repetir y ejecutar acciones. El formato básico para un comando for:

for X in iterable:

El comando se expresa casi del mismo modo que en inglés. Podemos ejecutar todas las letras de la palabra banana de este modo:

In [ ]:

```
for letter in "banana":
    print(letter)
```

El código en el bucle es ejecutado tantas veces como hay ahí letras, con un valor diferente para la variable letter (letra) en cada iteración.

Del mismo modo, podemos ejecutar todos los elementos en una lista:

In [ ]:

```
colors = ["yellow", "red", "green", "blue", "purple"]
for whatever in colors:
    print("This is color " + whatever)
```

Dado que los diccionarios son asimismo objetos iterables, podemos iterarlos también en nuestra colección de libros favoritos. Esto iterará sobre todas las claves de nuestro diccionario:

In [ ]:

```
for book in good_reads:
    print(book)
```

Podemos asimismo iterar las claves y los valores de un diccionario, como en este ejemplo:

In [ ]:

```
good_reads.items()
```

In [ ]:

```
for x, y in good_reads.items():
    print(x + " has score " + str(y))
```

Usar items() (elementos) puede, en cada iteración, devolver un par de claves y sus valores. En el ejemplo más arriba la variable book hará un bucle sobre las claves del diccionario, y la variable score hará un bucle sobre sus respectivos valores.

Este es el modo más elegante de hacer bucles sobre los diccionarios, pero vamos a ver si comprendés también estas otras alternativas:

In [ ]:

```
for book in good_reads:
    print(book, "has score", good_reads[book])
```

Quiz!

La función len() devuelve la longitud de un elemento iterable:

In [ ]:

```
len("banana")
```

Podemos usar esta función para ejecutar la extensión de cada palabra en una lista de colores. Escribí el código en el ejemplo de abajo:

In [ ]:

```
colors = ["yellow", "red", "green", "blue", "purple"]

# insert your code here
```

Ahora escribí un pequeño programa que itere a través de la lista colors y añada (append) todos los colores que contengan la letra r en la lista colors_with_r. (Tip: usá colors_with_r.append)

In [ ]:

```
colors = ["yellow", "red", "green", "blue", "purple"]

colors_with_r = []

# insert you code here
```

Qué aprendimos

He aquí un repaso de todos los nuevos conceptos, expresiones y funciones que aprendimos en esta sección. Una vez más, revisá la lista y asegurate que lo entendés.

bucle

comando for

objetos iterables

asignación de una variable en un bucle con for

Final Quiz!

Es tiempo de hacer un repaso final. El próximo ejercicio puede ser difícil pero estaríamos muy contentos si te sale bien!

Lo que queremos es que escribas un código que cuente la cantidad de veces que se repite la palabra a en este pequeño corpus. Necesitás hacer esto basándote en la frecuencia de distribución de las palabras que están en el diccionario. En este diccionario de frequency_distribution (frecuencia de distribución) las claves son palabras y los valores son las frecuencias. Asignale un valor a la variable number_of_as (cantidad de 'a').

In [ ]:

```
frequency_distribution = {"Beg": 1, "Goddard's": 1, "I": 3, "them": 2, "absent": 1, "already": 1,
                          "alteration": 1, "amazement": 2, "appeared": 1, "apprehensively": 1,
                          "associations": 1, 'clever': 1, 'clock': 1, 'composedly': 1,
                          'deeply': 7, 'do': 7, 'encouragement': 1, 'entrapped': 1,
                          'expressed': 1, 'flatterers': 1, 'following': 12, 'gone': 9,
                          'happening': 4, 'hero': 2, 'housekeeper': 1, 'ingratitude': 1,
                          'like': 1, 'marriage': 15, 'not': 25, 'opportunities': 1,
                          'outgrown': 1, 'playfully': 2, 'remain': 1, 'required': 2,
                          'ripening': 1, 'slippery': 1, 'touch': 1, 'twenty-five': 1,
                          'ungracious': 2, 'unwell': 1, 'verses': 1, 'yards': 5}
number_of_as = 0
# insert your code here
```

Llegaste al fin de este capítulo. Ignorá el código de abajo, lo pusimos aquí solo para embellecer la página:

In [ ]:

```
from IPython.core.display import HTML

def css_styling():
    styles = open("styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```