

## TALLER “PROGRAMACIÓN ORIENTADA A LA BIOLOGÍA”

(En el marco del III CONCURSO “BIOINFORMÁTICA EN EL AULA”)

### ¡Comencemos! Pero, ¿cómo?

Podemos comenzar con algunas definiciones que nos ayuden a entender un poco más de qué se trata este taller:

### ¿En qué consiste una computadora?

Una computadora está formada por el *Hardware* (que son todas las partes o elementos físicos que la componen) y el *Software* (que son todas las instrucciones para el funcionamiento del Hardware). El sistema operativo es el principal software de la computadora, pues proporciona una interfaz con el usuario y permite al resto de los programas una interacción correcta con el hardware.

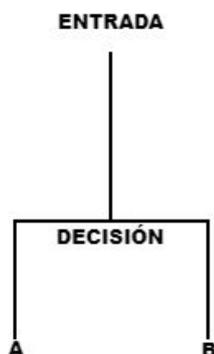
### ¿Qué es programar?

Fundamentalmente, la programación consiste en la resolución de problemas de la vida cotidiana mediante el uso de una computadora. Podríamos enumerar algunos pasos básicos que se requieren para programar:

- Modelar el problema. Es decir, realizar una simplificación práctica del problema con la cual podemos trabajar mejor. (Por ejemplo si una persona camina, decimos que lo hace en línea recta para poder representarlo más fácilmente).
- Escribir la serie de pasos necesarios para que se resuelva el problema.
- Verificar que mediante nuestro programa se logra resolver el problema correctamente. Si no es así, hay que volver al paso anterior y modificar la serie de pasos escritos.

Ahora bien, cuando planteamos los pasos para resolver el problema, podemos optar entre tres enfoques:

- **Programación procedimental.** Se da un paso, comenzando por el primero y cuando éste termina se continúa con el siguiente; así, hasta llegar al final.
- **Tomar decisiones.** Muchas veces se hace necesario que el programa tome un camino u otro, según al estado en que se encuentra. Por ejemplo, imaginemos que queremos construir un programa que calcule si los usuarios son mayores de edad. Como un primer paso pediremos al usuario que nos diga su edad (ENTRADA). Luego deberemos decidir si este valor es mayor o menor que 18. **SI** se cumple la condición de ser mayor de 18, el programa imprimirá en pantalla el cartel “Eres mayor” (OPCIÓN A), pero **SI NO** se cumple, imprimirá “no eres mayor” (OPCIÓN B).



- **Repetición.** Llegados a un punto de nuestra solución, es probable que los pasos propuestos en el programa deban repetirse varias veces. Por ejemplo, si quisiéramos calcular el área de un cuadrado, propondríamos un programa que multiplique por sí mismo el tamaño de un lado del cuadrado. Ahora bien, si tuviéramos siete cuadrados, el programa debería calcular el área de cada uno. Al escribirlo de manera procedimental, el programa quedaría así:

```

calcular_area cuadrado 1
calcular_area cuadrado 2
calcular_area cuadrado 3
calcular_area cuadrado 4
calcular_area cuadrado 5
calcular_area cuadrado 6
calcular_area cuadrado 7

```

Lo cual es ciertamente muy engorroso. Sin embargo existe otro modo en el que podemos escribir el mismo programa:

```

repetir 7 veces:
    calcular_area cuadrado X

```

Así, el paso en el que calculamos el área se repite 7 veces, y el programa es mucho más claro y fácil de escribir.

### ¿Qué hacemos entonces cuando programamos?

Una computadora está constituida, básicamente, por un gran número de circuitos eléctricos que pueden ser activados (1) o desactivados (0). Al establecer diferentes combinaciones de prendido y apagado de los circuitos, los usuarios de computadoras podemos lograr que el equipo realice alguna acción (por ejemplo, que muestre algo en la pantalla).

Pero como escribir en binario es complicado para los humanos, se desarrollaron múltiples lenguajes de programación. Éstos actúan como traductores entre el usuario y el equipo. En lugar de aprender el difícil lenguaje de la máquina, con sus combinaciones de ceros y unos, se puede utilizar un *lenguaje de programación* para dar instrucciones al equipo de un modo que sea más fácil de aprender y entender.

Esto significa que, como programadores de Python (o cualquier otro lenguaje), no necesitamos entender lo que el equipo hace o cómo lo hace, basta con que aprendamos a “hablar y escribir” en el lenguaje de programación.

### ¿Por qué es útil aprender a programar?

Tu smartphone, tu Playstation o Smart TV no serían muy útiles sin programas (aplicaciones, juegos, etc) para hacerlas funcionar. Cada vez que abrimos un documento para hacer un trabajo práctico para la escuela, o usamos el WhatsApp para chatear con nuestros amigos, estamos usando programas que interpretan lo que deseamos realizar, como por ejemplo cambiar un color de fuente, aumentar el tamaño de letra o enviar un mensaje. Estos programas le comunican nuestras órdenes a la PC o teléfono para que las ejecuten. Aprendiendo a programar podrías hacer una gran diversidad de cosas: desde escribir tus propios juegos y aplicaciones para celular, combinar el uso de varios programas en forma secuencial o leer millones de textos sin abrir un solo libro... hasta analizar el



genoma de un organismo o miles de estructuras de proteínas y así sacar conclusiones de relevancia biológica. Si bien existen muchas herramientas hoy en día para hacerlo, el poder construir tus propias herramientas te dará la posibilidad de aplicarlas a la situación particular que se te presente y necesites resolver.

### ¿En qué se relacionan una vaca y una computadora?

La bioinformática es una disciplina científica destinada a la aplicación de métodos computacionales al análisis de datos biológicos, para poder contestar numerosas preguntas. Su objeto de trabajo son entidades biológicas: ADN, proteínas, organismos completos, sus poblaciones, etc. Las tecnologías computacionales permiten, entre otras cosas, el análisis en plazos cortos de gran cantidad de datos (provenientes de experimentos, bases de datos, etc.); así como la predicción de la forma o la función de las distintas moléculas, o la simulación del comportamiento de sistemas biológicos complejos como células y organismos...en este caso... LA VACA!! Así es como echando mano de la tecnología podemos percibir distintos procesos biológicos y poner en imágenes conceptos que pueden resultarnos de otra forma muy abstractos.

### Pensamiento computacional: datos biológicos.

Un paso importante para aplicar el pensamiento computacional (binario), en otras palabras, pasar del cuaderno a la compu, es reconocer qué datos o qué información nos aportan un conocimiento biológico. En principio, se podría decir que toda descripción de un sistema biológico podría ser un “dato biológico”. Por ejemplo, se pueden considerar datos biológicos el número de murciélagos en una región dada, la cantidad de pacientes enfermos con gripe en una población, la cantidad de glóbulos rojos por mililitros de sangre, entre otros. Ahora bien, para comenzar desde lo micro a lo macro podríamos pensar en las células. Dentro de cada una de las células del cuerpo hay información importante almacenada, también hay otras tantas cosas, pero enfocándonos en la información que las hace a cada “célula quienes son” podríamos destacar unas moléculas muy importantes.

**RETO TI: ¿Podrías buscar un ejemplo de macromoléculas que almacenen información sobre la ‘identidad’ de una célula?**

### Entonces: ¿Qué usamos para programar? ¿Qué es Python?

Es un lenguaje de programación con una forma de escritura (sintaxis) sencilla y poderosa. Es lo que se conoce como lenguaje de *scripting*, que puede ser ejecutado por partes y no necesita ser compilado en un paso aparte. Python tiene muchas características, ventajas y usos que vamos a pasar por alto en este taller, pero que puedes leer en las páginas oficiales de [Python](#) y [Python Argentina](#). Para nosotros, la razón más importante para elegir Python es que puedes comenzar a hacer tus propios programas realmente rápido.



### ¿Cómo se puede usar Python?



Depende del dispositivo que uses. En las computadoras, suele venir instalado. Si tenés un teléfono inteligente existen varias aplicaciones que instalan todo lo necesario para que Python funcione. Solo debés buscar ‘Python’ en tu tienda y descargarte alguna de las apps disponibles. Recomendamos las siguientes opciones gratuitas:

- Para teléfonos Android: QPython 3 (o Pydroid 3).
- Para teléfonos Windows: Python 3.
- Para iOS: python2IDE.

**PARA TENER EN CUENTA: Estas aplicaciones son gratuitas y funcionan sin conexión a internet. Sin embargo es necesaria la conexión para poder descargarlas.**

## ¿Cómo escribimos código en Python dentro de las apps?

¡Es muy fácil! Sólo tenés que abrir la app y ejecutar la *consola* o terminal de Python. Te aparecerá una pantalla con un pequeño símbolo, usualmente ‘>’, donde podrás ingresar las distintas órdenes (o *comandos*) que le darás a la computadora, siempre en sintaxis de Python. Cada vez que des *Enter* se ejecutará la orden ingresada y podrás escribir un nuevo comando. ¡Ojo! Al salir de la consola se borrarán los comandos, a menos que los guardemos en un archivo o *script* para volver a ejecutarlos más adelante.

## ¿Qué otras formas tenemos para correr Python?

Existen *consolas en línea* que te permiten correr Python en internet como si estuviese instalado en tu PC o teléfono, que son completamente gratis (bueno, ¡siempre que tengas internet!). Te recomendamos dos, pero siempre podés buscar otras:

- repl.it: <https://repl.it/languages/python3>
- Tutorials Point: [https://www.tutorialspoint.com/execute\\_python\\_online.php](https://www.tutorialspoint.com/execute_python_online.php)
- Mumuki: <https://mumuki.io/home/>



**PARA TENER EN CUENTA: Estas aplicaciones son gratuitas pero funcionan con conexión a internet.**

## “Aún el camino más largo siempre comienza con el primer paso” - Lao Tse

El primer paso para poder hacer tu primer programa es abrir la consola de Python, tu app del teléfono o consola en línea, ¡lo que tengas a mano para arrancar!

**RETO TI: Como vimos el objetivo de este taller es darle una vuelta de tuerca a la Biología, si bien existen muchas herramientas disponibles para analizar datos biológicos, poder construir tus propias herramientas te dará la posibilidad de comprender en profundidad los conceptos y aplicar este conocimiento a nuevas problemáticas.**

**Ahora bien, es importante conocer nuestra herramienta de trabajo, en este caso nuestra PC o teléfono inteligente. ¿Podrías identificar qué parte de la computadora estaremos trabajando?**

## El principio de un comienzo

En todo proceso de aprendizaje los ‘errores’ tienen un rol muy importante: nos plantean nuevos interrogantes, nos acercan a nuevas hipótesis y sobre todo nos dan oportunidades para seguir aprendiendo. En la programación los ‘errores’ también importan, ¡y mucho! Son una suerte de comunicación con la máquina, que nos advierte cuando no funciona algo de lo que intentamos hacer. Existen distintos tipos de errores en Python y con cada tipo de error la máquina nos indica qué es lo que puede estar fallando de nuestro código. Por eso te pedimos que anotes todos los errores que puedan ir apareciendo durante tu trabajo en el taller o en casa y que los compartas con nosotros, para charlar entre todos acerca de los conocimientos que se ponen en juego durante la resolución de estos problemas. Así que, no tengas miedo de equivocarte, ¡no se rompe nada! Y como diría la señorita Ricitos en su Autobús Mágico, **a cometer errores, tomar oportunidades y rockear con Python, que donde termina la carretera comienza la aventura!**

## Tu primer programa

Una forma no muy original de aprender a escribir un primer programa en Python es simplemente abrir la consola, escribir lo siguiente y darle *Enter*:

```
print('Hola Bioinformática!')
```

¿Qué pasó? `print` es una función que te permite imprimir o mostrar en la consola todo lo que se encuentre dentro de los paréntesis y entre comillas, como en nuestro ejemplo. Entre otras cosas, esta función nos permite interactuar con nuestro programa o con el futuro usuario de nuestro programa. Felicitaciones, ¡ese fue tu primer programa en Python!

### Una calculadora super-archi-genial

Con Python podemos hacer todo tipo de cálculos matemáticos. Aunque suene medio bodrio, aprender a hacer estos cálculos nos va a ayudar después a trabajar sobre otros tipos de datos. Vamos a probar algunos cálculos. Escribí en tu consola:

```
3*5
```

¿Cuál es el resultado? Si, como ves, el asterisco es el símbolo que se utiliza en Python para multiplicar. Probemos ahora:

```
8/4
```

¿Qué resultado nos da? ¿Para qué se usa la barra hacia adelante?

En Python se puede usar los siguientes símbolos básicos de matemáticas, que en programación se llaman operadores:

+	<b>Suma</b>
-	<b>Resta</b>
*	<b>Multiplicación</b>
/	<b>División</b>

Si por ejemplo tomamos la siguiente operación:

```
5+30*20
```

¿Qué da? ¿Por qué? Y si ahora hacemos:

```
(5+30)*20
```

¿Nos da el mismo resultado? ¿Por qué pensás que ocurre eso?

De las dos operaciones anteriores podemos concluir dos cosas importantes: en este lenguaje, al igual que en la matemática, los operadores no tienen la misma prioridad de lectura o ejecución. La multiplicación y la división tienen mayor orden de prioridad que la suma y la resta. Esto significa que en el ejemplo `5+30*20`, Python primero realiza la operación `30*20` y luego le suma 5 al resultado de la multiplicación. Los paréntesis nos sirven para reordenar las prioridades: al hacer `(5+30)` obligamos la ejecución de esta operación antes que la multiplicación.

¿Qué tal si probamos algo más complejo? Escribamos lo siguiente:

```
((4+5)*2)/5
```

**RETO I: ¿Podés descubrir y anotar el orden en que se ha ejecutado cada operación?**

## ¡Un momento, cerebrito!

Al trabajar en Python, cada elemento es un objeto distinto: el número 18, la palabra "dinosaurio", la frase "Hola Mundo!"... Para saber de qué objeto estamos hablando, podemos usar variables. Podríamos decir entonces que una variable es un nombre que le ponemos a un objeto en particular. Y aunque no sea estrictamente cierto, podemos pensar que al darle un nombre a esos objetos, es decir al declarar una variable, hará que Python recuerde lo que le hayamos asignado a ese nombre.

En Python, una variable se define con la sintaxis:

nombre de la variable = valor de la variable

### ¡No cualquiera está en la tele!

Existen en Python algunas restricciones para los nombres de variables:

Los nombres de variables deben ser UNA sola palabra alfanumérica (con letras y números). Eje.: variable1

No pueden comenzar con números, aunque sí contenerlos.

No deben ser utilizados los nombres de funciones como "print".

Definamos entonces algunas variables:

```
a = 5
b = 3
c = 'hola'
```

Como verás, una variable puede contener números, textos, lista de cosas, etc. En el caso de las letras o palabras, siempre debemos escribirlas entre comillas para que Python las considere como texto.

Ahora le preguntaremos a Python cuánto vale 'b':

```
print(b)
```

Veremos que efectivamente se ha guardado en memoria que b es igual a 3. En Python podemos modificar el contenido de una variable, es decir asignarle un nuevo contenido, simplemente declarándola de nuevo:

```
b = 500
print(b)
```

Además, podemos asignar a más de una variable el mismo valor:

```
d = 500
print(b)
print(d)
```

**RETO II: Creá una variable llamada *doble*, que sea el doble de la suma entre a y b.**

## ¿Y qué con eso?

Muy interesante esto de las variables, pero ¿para qué sirve? Una utilidad obvia es almacenar en memoria datos que nosotros seguro olvidaremos, pero que podríamos llegar a necesitar más adelante. Una variable también nos podría ayudar a guardar datos que aún no conocemos, pero que querríamos modificar o utilizar una vez adquiridos. Por ejemplo, nos gustaría crear un programa que salude al que lo usa. Como no sabemos *a priori* quién va a usar nuestro programa, podemos consultarle primero su nombre y luego saludarlo. Para ello sería útil guardarnos el dato de quién es la persona:

```
input('Decime tu nombre, por favor!')
```

Esta función `input` le consulta al usuario algo, lo que escribamos entre paréntesis y comillas, y espera la respuesta. Verás que una vez ejecutada la línea no vuelve a aparecer el *prompt* hasta que no ingresemos alguna palabra y le demos *Enter*.

Ahora bien ¿podrías armar un programa que salude a quien lo use?

Pensemos juntos qué pasos debe tener el programa:

- 1) Lo primero que deberíamos hacer es preguntarle su nombre al usuario y almacenarlo en una variable. ¡Hagámoslo!

```
nombre_de_usuario = input('Decime tu nombre, por favor!')
```

- 2) Luego podemos imprimir un mensaje en pantalla que contenga el mensaje deseado:

```
print ('Hola ' + nombre_de_usuario + '. ¡Este es mi programa!')
```

Muy probablemente, si lo hiciste sin mirar la guía, tu programa no sea exactamente igual al mío. ¡Existen muchas formas de lograr un mismo resultado! Así pues, si tu programa saluda a la persona que lo usa lo has hecho bien, aún cuando no sea de lo más estético el saludo. En el ejemplo anterior usamos un “truco” para imprimir el saludo, que puede serte muy útil de ahora en más: en Python podemos unir palabras o textos para generar frases o textos más largos simplemente sumándolas (+).

## Nada es mejor, nada es igual... Bueno igual puede ser!

Existen formas de comparar dos variables, y esto es utilizando lo que se conoce como *operadores relacionales*. Con ellos podemos saber si dos variables son iguales (`==`), o si son distintas (`!=`), o si una es mayor que la otra (`>`). Por ejemplo:

```
edad_lola = 13
edad_ana = 32
print(edad_lola == edad_ana)
```

¿Qué resultado obtenés al comparar dos variables? Sí, los *operadores relacionales* no devuelven valores numéricos, sino que nos afirman (*True*) o rechazan (*False*) la hipótesis que hayamos puesto a prueba. En nuestro ejemplo la hipótesis es que la edad de Ana es igual a la de Lola y por eso Python nos devuelve *False*.

Los operadores relacionales que se pueden usar en Python son:

==	<b>Igual</b>
!=	<b>Distinto</b>
<	<b>Menor</b>
>	<b>Mayor</b>

¿Has probado estos operadores con variables que contengan una palabra? Intentá crear dos variables cuyos valores sean “Ana” y “ana” y verificar si son iguales. ¿Qué da? ¿Por qué?

### Una palabra no dice nada...

En programación al texto se le llama ‘string’. Este tipo de datos no es más que una cadena de caracteres, así como una palabra se puede entender como una cadena de letras. Un string no necesariamente tiene que tener sentido. En Python las cadenas se definen escribiendo los caracteres entre comillas (simples o dobles, indistintamente). A una variable se le puede asignar una cadena de la siguiente forma:

```
cadena = 'este es un ejemplo de cadena'
print(cadena)
```

Podemos operar sumando y multiplicando cadenas del mismo modo en que operamos con números.

```
a = 'Hola '
b = 'mundo!'
print(a+b)
```

¿Qué pasa si a una cadena le sumamos un número? Probá hacer `print(a + 5)` a ver qué pasa.

Las cadenas pueden ser comparadas con los operadores relacionales que vimos antes. Así, podemos saber entonces si dos cadenas son distintas o no lo son (tené en cuenta que Python distingue entre mayúsculas y minúsculas):

```
palabra = 'si'
lo_mismo = 'si'
print(palabra == lo_mismo)
```

¿Qué pasa si el contenido de la variable `lo_mismo` comienza con mayúsculas?

**RETO III: Digamos que el ADN no es más que un mensaje en clave, que debe ser descifrado o interpretado para la síntesis de proteínas. El mensaje está escrito por una secuencia determinada de 4 nucleótidos distintos representados por las letras A, T, G y C. Dentro de la célula, el mensaje es transportado por otra molécula, el ARN, muy similar al ADN pero con U en vez de T. En este mensaje, cada triplete o grupo de tres letras del ARN se denomina codón, y cada aminoácido de las proteínas está codificado por uno o varios codones. Así por ejemplo el codón ‘AUG’ codifica para el aminoácido Metionina, el codón ‘AAA’ para Lisina, el codón ‘CUA’ para Leucina, etc.**

¿Podrías escribir una cadena de ARN que codifique para el péptido (una cadena corta de aminoácidos) ‘Met-Lis-Lis-Lis-Leu-Leu-Met’ combinando las variables `met = ‘AUG’`, `lis = ‘AAA’` y `leu = ‘CUA’` utilizando operadores matemáticos?  
 ¿Compartinos tu código en nuestro grupo de Facebook ‘[Talleres de programación Orientada a la Biología - SBG\\_UNQ](#)’!

### Fetas de texto: dame doscientos!

En Python podemos saber qué caracteres o subpartes conforman una cadena o *string*. Python le asigna a cada caracter de una cadena un número de posición. El primer carácter es la posición cero (¡sí, cero!) y las posiciones aumentan de a una hasta el fin de la cadena.

Por ejemplo en la cadena `a = 'hola'`, la ‘h’ tiene asignada la posición cero y la ‘a’ tiene asignada la posición tres.

h	o	l	a
0	1	2	3

Si quisiéramos saber cuál es el primer caracter de la cadena, hacemos referencia al caracter de la posición cero de la misma escribiendo el nombre de la variable seguida de la posición que nos interesa, escrita entre corchetes:

```
a = 'Hola mundo'
print(a[0])
```

Podríamos tomar solo un segmento de la cadena, indicando entre corchetes desde qué carácter hasta qué carácter, separado por dos puntos.

```
print(a[0:1])
```

En este caso, ¿cuántos caracteres se imprimirán? ¿Cómo hacemos para imprimir la palabra ‘Hola’ completa?

### Chamuyo: el arte de manipular palabras/cadenas.

Existen muchas funciones útiles para manipular cadenas. Como dijimos antes, a Python no le dan lo mismo las mayúsculas que las minúsculas. Existen funciones muy útiles para interconvertir unas en otras:

```
apellido = 'Curie'
apellido.upper()
```

¿Y si queremos minúsculas?

```
nombre = 'MARIE'
nombre.lower()
```

Podemos conocer la longitud de una cadena (es decir de cuántos caracteres está formada) utilizando la función `len()`:

```
apellido = 'Marie'
len(apellido)
```

También podemos conocer la cantidad de veces que aparece un dado carácter en una cadena utilizando la función `count()`:

```
apellido = 'Marie'
apellido.count('e')
```

Otra función muy útil para la manipulación de cadenas es `replace()`. Esta función nos permite reemplazar un tipo de carácter por otro. Por ejemplo:

```
apellido = 'Marie'
apellido.replace('e','a')
```

Cómo podés ver, la forma de reemplazar un carácter por otro es escribir entre los paréntesis de la función `replace('caracter viejo','caracter nuevo')` el carácter que queremos reemplazar, entre comillas y separado por coma del carácter por el cual lo queremos reemplazar, también entre comillas.

**RETO IV: ¿Cadenas?¿letras? Si hablamos de cadenas y letras en Biología, lo primero que se nos viene a la cabeza son las macromoléculas. Como bien sabemos, el ADN es un mensaje en clave que guía la síntesis de proteínas. Este mensaje está escrito por una secuencia determinada de 4 nucleótidos distintos representados por las letras A, T, G y C. El contenido de C y G (es decir el porcentaje de CG) presente en el ADN de un organismo es una característica distintiva: por ejemplo las *Actinobacterias* tienen un contenido característicamente más alto de CG que otros organismos. Ahora, contar la cantidad de C y G en una cadena de ADN larguísima a mano puede ser un verdadero tedio ¿Podrías crear un programa que calcule el porcentaje de C y G de una cadena dada de ADN?**

**¡Compartinos tu código en nuestro grupo de Facebook [‘Talleres de programación Orientada a la Biología - SBG UNQ’!](#)**

**Secuencia ejemplo:**

```
TGATAAGAGTACCCAGAATAAAATGAATAACTTTTTAAAGACAAAATC
CTCTGTTATAATATTGCTAAAATTATTCAGAGTAATATTGTGGATTAA
GCCACAATAAGATTTATAATCTTAAATGATGGGACTACCATCCTTACT
CTCTCCATTTCAAGGCTGACGATAAGGAGACCTGCTTTGCCGAGGAG
GTACTACAGTTCTCTTACAAAACAATTGTCTTACAAAATGAATAAAAC
AGCACTTTGTTTTATCTCCTGCTTTTAATATGTCCAGTATTCATTTTT
GCATGTTTGGTTAGGCTAGGGCTTAGGGATTTATATATCAAAGGAGGC
TTTGTACATGTGGGACAGGGATCTTATTTTAGATTTATATATCAAAGG
AGGCTTTGTACATGTGGGACAGGGATCTTATTTTACAAACAATTGTCT
TACAAAATGAATAAAACAGCACTTTGTTTTATCTCCTGCTCTATTGTG
CCATACTGTTGAATGTTTATAATGCATGTTCTGTTTCCAAATTTTCATGA
AATCAAACATTAATTTATTTAAACATTTACTTGAAATGTTACAAACA
ATTGTCTTACAAAATGAATAAAACAGCACTTTGTTTTATCTCCTGCTT
TTAATATGTCCAGTATTCATTTTTGCATGTTTGGTTAGGCTAGGGCTT
AGGGATTTATATATCAAAGGAGGCTTTGTACATGTGGGACAGGG
ATCTTATTTTACAAACAATTGTCTTACAAAATGAATAAAACAGCACTTT
```

GTTTTATCTCCTGCTCTATTGTGCCATACTGTTGAATGTTTATAATGC  
 ATGTTCTGTTTCCAAATTCATGAAATCAAAACATTAATTTATTTAAAC  
 ATTTACTTGAAATGTGGTGGTTTGTGATTTAGTTGATTTTATAGGCTA  
 GTGGGAGAATTTACATTCAAATGTCTAAATCACTTAAAATTTCCCTTA  
 TGGCCTGACAGTAACTTTTTTTTATTCATTTGGGGACAACATATGTCCG  
 TGAGCTTCCATCCAGAGATTATAGTAGTAAATTGTAATTAAGGATAT  
 GATGCACGTGAAATCACTTTGCAATCAT

### Lo esencial es invisible a los ojos

Para Python, buscar una aguja en un pajar ¡es un juego de niños! Lo que metafóricamente llamaríamos ‘leer entre líneas’, en programación podría hacerse sin problemas. Incluso para cadenas con millones de caracteres podríamos encontrar patrones determinados en cuestión de segundos, y saber si un *substring* o fragmento de cadena está presente o no. Los operadores de pertenencia `in` y `not in` nos permiten este tipo de operaciones. Simplemente mediante la afirmación `in` podemos consultar si la subcadena está en la cadena. Python nos responderá afirmativamente (`True`) si esta afirmación es cierta, o negativamente (`False`) si es falsa:

```
cadena = 'canis lupus'
'can' in cadena
```

### El lenguaje de la naturaleza

Como seguramente has notado, en el tratamiento de datos biológicos es muy importante la manipulación del tipo de datos *string* (cadenas). Muchos de los datos que podemos obtener de un organismo pueden ser pensados como una secuencia de caracteres: el ADN, el ARN, las secuencias proteicas, etc. ¡Estamos un paso más cerca de convertirnos en verdaderos bioinformáticos!

**RETO V: La Asombrosa Maravillosa es nuestra valiente superheroína. Sus poderes son producto de mutaciones en un gen muy común, cuya secuencia en la mayoría de las personas es 'ATGGAACTTGCAATCGAAGTTGGC'. A diferencia de nosotros, el gen mutado de la Asombrosa Maravillosa incluye la secuencia 'GTTTGTGGTTG' en su interior.**

**La Asombrosa Maravillosa adquirió sus poderes al beber Jugo Vencido. El primer sorbo de esta poción prohibida causa el cambio de todas las citosinas (C) por timinas (T). El siguiente sorbo cambia todas las adeninas (A) por guaninas (G). El tercer sorbo cambia las citosinas (C) por adeninas (A). El cuarto sorbo... puede ser mortal.**

**¿Podés escribir un programa que nos diga cuántos sorbos de Jugo Vencido debe beber un portador del gen normal, para ganar los poderes de la Asombrosa Maravillosa?**

**¡Compartinos tu código en nuestro grupo de Facebook ‘[Talleres de programación Orientada a la Biología - SBG UNQ](#)’! Buscanos en Twitter!**



Python nos permite pensar en grande. Para eso, nos da un tipo de datos que se llaman listas, que nos permiten manipular un gran número de datos en forma sencilla. Las listas son conjuntos de datos en

cantidad ilimitada: podemos tener desde listas vacías, hasta listas con tantos elementos como se nos ocurra.

Por ejemplo, si conocemos los nombres de algunos de los dinosaurios más grandes que han vivido en el territorio de la Argentina, podemos guardar una lista con sus nombres en una variable:

```
dinos = ['Patagotitan', 'Titanosaurus', 'Argentinosaurus']
```

Las listas se escriben separando a sus elementos con comas, y agrupando a todos entre corchetes. Los elementos pueden ser cadenas, números, ¡incluso otras listas!

Al igual que se puede acceder a los distintos caracteres de una cadena, podemos acceder a los elementos de una lista indicando entre corchetes el número de elemento que queremos obtener. Recordá que Python comienza a contar desde cero. Para obtener el segundo elemento, podemos escribir:

```
dinos[1]
```

Podemos seleccionar solo algunos elementos de la lista:

```
dinos[0:2]
```

Podemos conocer, como en las cadenas, el número de elementos de la lista con la función `len()`:

```
len(dinos)
```

### **Pasito a pasito...**

Un recurso muy utilizado en el desarrollo de programas es crear variables vacías que luego vamos llenando. Imaginemos que somos los primeros cazadores de dinosaurios del país y queremos comenzar nuestra lista de descubrimientos asombrosos. Esta lista comenzará vacía pero a medida que nos volvamos famosos y millonarios con nuestros descubrimientos, la lista irá creciendo. ¿Cómo armamos nuestra lista vacía? Simplemente declarando una variable con el formato de una lista, pero sin nada adentro:

```
dinos = []
```

Una vez declarada la variable podemos ir agregando otros dinosaurios gigantes de la Argentina utilizando la función `append()`, de la siguiente forma:

```
dinos.append('Puertasaurus')
```

Si descubrimos que alguno de los dinosaurios no debería pertenecer a la lista (por ejemplo y pese a su nombre, el Titanosaurus no era tan grande como los demás), podríamos eliminar elementos de nuestra lista de la siguiente manera:

```
dinos.remove('Titanosaurus')
```

Y si quisiéramos llevar un registro prolijo de nuestros dinosaurios, podríamos ordenar la lista alfabéticamente utilizando la función `sort()`:

```
colonias.sort()
```

### **RETO VI: ¿Se te ocurre qué operadores podrías usar para las listas?**

**La historia sin fin... ¡O casi!**

Existen más tipos de datos en Python, entre ellos, lo que se denominan tuplas y diccionarios. Ambos tipos de datos son muy útiles a la hora de programar. ¡Pero esta vez vamos a dejarte investigar un poco a vos!

### Quedándote o yéndote...

Como en todo, a la hora de escribir un programa debemos tomar decisiones y las decisiones que tomamos siempre dependen de los condicionantes que se presentan. En Python existe una sentencia llamada `if` (del inglés: si condicional) que le permite al programa hacer una cosa u otra, dependiendo de las condiciones que fijemos. **Si** cierta condición se cumple, **entonces** el programa hace algo. En el caso de que la condición no sea cierta podemos pedirle que haga algo más usando la sentencia `else`. La estructura de esta sentencia es la siguiente:

*if condición:*

*aquí van las órdenes que se ejecutan si la condición es cierta*

*else:*

*aquí van las órdenes que se ejecutan si la condición es falsa*

Como verás una sentencia **if** se compone de un '**if**', que significa "si", seguido de una '**condición**' y terminando con dos puntos (:). Una condición es un cálculo de programación cuyo resultado es verdadero (*True*) o falso (*False*), y se crea utilizando los operadores relacionales que ya conocés (`==`, `>`, `<`, `!=`). La línea siguiente son las órdenes a ejecutar si la condición es cierta, y siempre comienza con un *tab*. La sentencia **else** consiste de un '**else**', que significa "de lo contrario", y dos puntos (:). En la línea de abajo se escriben las órdenes a ejecutar en caso de que la condición `if` no sea cierta, también comenzando con un *tab*. Es muy importante dejar esta tabulación o sangría (un espacio en blanco, insertado con la tecla *tab*; o cuatro espacios con el ingresados espaciador) en el comienzo de la línea de abajo; ya que le permite al intérprete de Python saber en qué orden debe ejecutar cada cosa. De olvidar esta sangría Python te hará saber que no le gustó, con el cartel: **IndentationError: expected an indented block.**

Veamos un ejemplo para poder comprender mejor las sentencias **if/else**:

Imaginemos que formamos parte del departamento de Bioinformática del Centro Nacional de Frikiadas Atómicas (CNFA), que se encuentra enfocado en la poco creíble investigación sobre 'invisibilidad'. Sabemos como investigadores eruditos/as que existen resultados poco confiables que apuntan a que la presencia de la secuencia de nucleótidos 'ACATAAA' podría estar asociada a la invisibilidad en la rana marciana de pelo enulado. Queremos hacer un aporte a la ciencia creando un programa que detecte este marcador de invisibilidad en una secuencia dada, por ejemplo: 'ACAGAAAGTTAATGGAAGGACATAAAAGTTATATCA'. Por lo que nos proponemos escribir un programa que efectúe los siguientes pasos:

1) definimos nuestra variable `sec`

```
sec = "ACAGAAAGTTAATGGAAGGACATAAAAGTTATATCA"
```

2) revisamos el string:

- a) si el marcador está contenido en el mismo, imprimimos '¡Wow! ¡Es invisible!'
- b) de lo contrario imprimimos: '¿Qué flasheas? ¡Es normal!'

```
if 'ACATAAA' in sec:
```

```

print(";Wow! ;Es invisible!")
else:
    print(";Qué flasheas? ;Es normal!")

```

**PARA TENER EN CUENTA: Cualquier error en la sintaxis, ya sean la ausencia de los dos puntos (:) o la falta de la indentación requerida, generan un error al ser ejecutados de tipo 'syntax error' o 'Indentation error' respectivamente. Es importante hacer énfasis en la sintaxis en estos ejemplos y la coherencia en la escritura dentro del código.**

¡Y voilà! Ahora podemos diferenciar ranas marcianas con el gen de invisibilidad, de las que no lo tienen. ¿Qué sucede ahora si cambiamos nuestra secuencia por la de otra muestra que nos llega de marte?

```

sec2 = "ACAGAAAGTTAATGGAAGGACATCCAAGTTATATCA"

if 'ACATAAA' in sec2:
    print(";Wow! ;Es invisible!")
else:
    print(";Qué flasheas? ;Es normal!")

```

¿Qué resultado obtienes? ¿Podrías describir qué camino siguió el programa?

### Sana, sana, pata de rana ...

Algunas ranas marcianas han sido introducidas en el ecosistema terrestre como prueba piloto de la convivencia entre ambos mundos. Lamentablemente, las cosas no están yendo bien, y la especie se propaga a gran velocidad, por lo que ha sido necesarios profundizar en el estudio de estos animales tan particulares.

Luego de investigaciones exhaustivas y una profunda inspección ocular, se ha podido descubrir que algunas de ellas tienen 5 o 6 dedos en sus patas delanteras, en lugar de 4!! Esta diferencia responde a la presencia de dos variantes en el gen PATA: si este contiene la secuencia GGATGG la rana tiene 5 dedos, mientras que si tiene la secuencia GTTCTA la rana poseerá 6 dedos. En el CNFA nos encontramos con el desafío de desarrollar un programa que nos permita diferenciar las ranas marcianas de las terrestres.

Lamentablemente la opción rápida: cortar y pegar el código del trabajo anterior, pareciera no ser viable. Esta vez debemos diferenciar entre 3 situaciones posibles: que la rana tenga 4, 5 o 6 dedos; por lo que la estructura del programa anterior, en la que solo dos opciones son posibles, no nos sería útil. Afortunadamente, google nos cuenta que para estos casos en donde se deben tomar más de dos decisiones, existe una sentencia llamada **elif** y se ubica después del **if** inicial. Esta sentencia es una combinación de las palabras **else** e **if**, y significa justamente eso. **Si no** ocurre la condición de arriba, pero **si** ocurre la siguiente condición, se ejecutan las siguientes órdenes. La estructura general sería la siguiente:

```

if condición1:
    aquí van las órdenes que se ejecutan si la condición1 es
    cierta
elif condición2:

```

aquí van las órdenes que se ejecutan si la condición1 es falsa pero la condición2 es cierta

else:

aquí van las órdenes que se ejecutan si la condición1 y la condición2 son falsas

Ahora sí podemos podíamos escribir un programa que diferencie entre las ranas según la cantidad de dedos en sus patas:

```
sec = "ACAGAAAGTTAATGGAAGGACATAAAAGTTATATCA"

if "GGATGG" in sec:
    print("¡La rana tiene 5 dedos! ¡Qué cosa rara! Debe ser marciana")
elif "GTTCTA" in sec:
    print("¡La rana tiene 6 dedos! Debe ser marciana")
else:
    print("La rana es normal y tiene 4 dedos")
```

¡¡Qué suerte! Hemos salvado la ecología del mundo desechando las ranas marcianas!

### ¡Alguien quiere pensar en las ranas!

Descubrimos que las ranas que contienen la secuencia de ACATCGA y la secuencia GTACTA, tienen además una pigmentación anómala, generando como consecuencia que este tipo de ranas tengan lunares blancos en sus patas.

Ahora bien, para los casos en donde se debe analizar más de una condición a la vez y todas deben ser verdaderas, existe un operador llamado **and**. Este operador se coloca entre las condiciones a analizar. "And" en español se traduce como "y", eso quiere decir que la condición1 **Y** la condición2 se deben encontrar o deben ser verdaderas para que se ejecuten las siguientes órdenes. La sintaxis general es la siguiente:

```
if condición1 and condición2:
    aquí van las órdenes que se ejecutan si la condición1 y la
    condición2 son ciertas
```

Para comprobar que las ranas tengan lunares podemos escribir el siguiente programa:

```
sec = "ACAGACATCGAATGGAAGGACGTACTATATATCA"

if "GTACTA" and "ACATCGA" in sec:
    print("¡La rana pecosa!")
```

### Programadores vs zombies

Se ha descubierto un virus llamado H1Z1 que ataca a las ranas y las convierte en zombies. Solo las ranas con los motivos CGCCTA o GGTACTT en su gen PATA son sensibles al virus; mientras que

las ranas de 6 dedos son resistentes al virus. El ministerio de seguridad zombie ha decidido poner en cuarentena a estas últimas, de forma de detener la proliferación zombie y necesitan nuestra ayuda una vez más.

Si bien hemos aprendido muchos trucos, este caso nos plantea un nuevo desafío; ya habría que analizar si entre dos o más opciones o condiciones, al menos una de ellas es verdadera. Para ello, afortunadamente, existe un operador llamado **or**. Este se coloca entre las condiciones a analizar. "or" en español se traduce como "o", eso quiere decir que la condición1 **O** la condición2 se deben encontrar o deben ser verdaderas para que se ejecuten las siguientes órdenes. La estructura general podríamos escribirla de la siguiente manera:

```
if condición1 or condición2:
    aquí van las órdenes que se ejecutan si la condición1 o la
    condición2 son ciertas
```

Ahora bien, para comprobar cuáles de las ranas hay que poner en cuarentena podríamos entonces escribir el siguiente programa:

```
Rana_lunares = "GATGGACATCGACCGATATGCCGTGTACTAA"
Rana_4dedos = "ACAGAAAGTTAATGGAAGGACATAAAAATCA"
Rana_5dedos = "GAGGCTCGGATGGGTGGTACTTGCAATGCGC"

if "CGCCTA" or "GGTACTT" in Rana_lunares:
    print(";La rana albina debe estar en cuarentena!")
else:
    print(";La rana albina no debe estar en cuarentena!")
if "CGCCTA" or "GGTACTT" in Rana_4dedos:
    print(";La rana de 4 dedos debe estar en cuarentena!")
else:
    print(";La rana de 4 dedos no debe estar en cuarentena!")
if "CGCCTA" or "GGTACTT" in Rana_5dedos:
    print(";La rana de 5 dedos debe estar en cuarentena!")
else:
    print(";La rana de 5 dedos no debe estar en cuarentena!")
```

**RETO VII:** Ya que encontramos el espécimen de rana con pelo en marte, nos gustaría contrastar sus características con las ranas terrestres. Sabiendo que el gen de la proteína diminuta es 'ATGGAAGTTGGAATCCAAGTTGGA' y el gen de una proteína similar de rana terrestre es 'ATGGAAGTTAATGGAAGTTGGAGGAGA' ¿podés crear un programa que compare la longitud de ambos genes y según cuál sea más grande nos imprima un mensaje informándonos el resultado?

¡Compartinos tu código en nuestro grupo de Facebook '[Talleres de programación Orientada a la Biología - SBG\\_UNQ](#)'!

Ella tiene un loop, tiene un loop!!



Hay gente que es reiterativa, hay circunstancias que son reiterativas y hay tareas que son reiterativas. Así que, si no tenemos el pajarito bebedor de Homero Simpson, ¿saber programar en Python nos puede hacer la vida más sencilla! Python, al igual que los demás lenguajes de programación, tienen lo que se llama un **for-loop** o **bucle for**, que nos ahorra bastante trabajo.

Por ejemplo imaginemos que unos ingenieros fanáticos de la Bioinformática crearon unos robots porristas, los más nerds del mundo, para alentar el uso de la Bioinformática en las escuelas. Como ya somos expertos programadores nos encargaron hacer el ‘programa de aliento’. Para ello utilizaremos la vieja táctica de ir pidiendo letras: ‘Dame una B, te doy la B’. Podríamos hacerlo de la forma más tediosa, imprimiendo un cartel por cada letra, tipeando a mano cada línea:

```
print('¡Dame una B! ¡Te doy la B!')
print('¡Dame una I! ¡Te doy la I!')
print('¡Dame una O! ¡Te doy la O!')
print('¡Dame una I! ¡Te doy la I!')
print('¡Dame una N! ¡Te doy la N!')
print('¡Dame una F! ¡Te doy la F!')
print('¡Dame una O! ¡Te doy la O!')
```

Es un embole, ¿no? Otra opción es utilizar lo que se conoce en programación como *loop*. Probemos entonces hacer un bucle, en el que le pediremos a nuestro programa que para cada letra de la palabra ‘BIOINFORMÁTICA’, imprima el cartel ‘Dame una \_\_\_, te doy la \_\_\_!’.

Por suerte, eso se escribe en Python de una forma muy intuitiva:

- 1) Definimos un iterable o variable que podremos ir recorriendo:

```
palabra = 'BIOINFO'
```

- 2) Recorremos cada caracter de la cadena 'BIOINFO', para imprimir la frase 'Dame una \_\_\_ ! Te doy la \_\_\_ !'. Hacemos que para cada caracter (**for i**) en (**in**) el iterable (**palabra**), se imprima el caracter:

```
for i in palabra:
    print( '¡Dame una ' + i + '! ¡Te doy la ' + i + '!')
```

- 3) Ya fuera del bucle (sin la sangría), para que no se repite en cada vuelta, imprimimos la frase entera “Que se formó? BIOINFORMATICA!”:

```
print('¡Qué se formó? ' + palabra)
```

En resumen:

```
palabra = 'BIOINFORMATICA'
for i in palabra:
    print( '¡Dame una ' + i + '! ¡Te doy la ' + i + '!')
print('¡Qué se formó? ' + palabra)
```

El programa que acabamos de crear tiene un **bucle for**. Como hemos visto un **bucle for** consiste en un **for**, una variable (en nuestro caso **i**) que cambia en cada repetición del ciclo, separada por un **in** o “en” y un rango de números o cualquier variable ‘iterable’ (que pueda dividirse en partes), cuyos elementos van a ser los valores que tomará nuestra variable **i**. Nuestro bucle puede ser leído,

entonces, como: para todas las letras `i` de la palabra 'BIOINFO', imprimí el mensaje: 'Dame una '+' + `i` +', te doy la '+' + `i` + '!'.

Para construir el mensaje de salida, concatenamos las distintas partes de nuestro mensaje con un '+'.  
¡Ojo! Para que nuestro mensaje pueda ir cambiando en cada vuelta, utilizamos la variable `i` dentro del mensaje, notando que `i` esté sin comillas ya que es una variable que va tomando distintos valores cada vez. Recordá que es muy importante dejar una tabulación o sangría (un espacio en blanco, insertado con la tecla `tab`; o cuatro espacios) en el comienzo de la línea de abajo; de lo contrario Python te hará saber que no le gustó, con el cartel: **IndentationError: expected an indented block.**

### Divide y loopearás!...

Como dijimos antes, podemos usar cualquier tipo de dato divisible en partes para construir un bucle o *loop for*, ¡inclusive listas!

Imaginemos que tenemos una bacteria que se divide por fisión binaria, entonces la célula hija es lo que se denomina un clon: un organismo con genes exactamente iguales. Ahora, si quisiéramos decirle a cada célula nueva que no es más que una copia genéticamente idéntica de la bacteria parental, podríamos imprimir en cada vuelta de reproducción el cartel 'Habemus clon!'. Entonces, suponiendo que nuestra célula se reproduce 20 veces, podemos construir el siguiente código:

```
for i in range(0,20):
    print('¡Habemus clon!')
```

El programa que acabamos de crear tiene un bucle 'for', que consiste en un `for`, una variable cambiante (en nuestro caso `i`), un `in` y una lista de cosas (un rango de números) que van a ser los valores que tomará nuestra variable `i`. La función `range()` permite crear de forma rápida y sencilla una lista de números que comienza en el primero y finaliza antes del último. Nuestro bucle puede ser leído, entonces, como: para la variable `i`, tomando valores entre 0 y 49, imprimí cada vez en la pantalla el mensaje 'Habemus clon!'.

**RETO VIII:** Si nos ponemos un poco más estrictos, y siguiendo con el tema de los clones de bacterias, el programa que creamos antes tiene algunas fallas 'numéricas': en cada vuelta de división celular binaria se generarán dos clones, no uno. ¿Podrías escribir un programa que imprima '¡Somos 2 clones nuevos!' en cada una de 20 vueltas?

¡Compartinos tu código en nuestro grupo de Facebook '[Talleres de programación Orientada a la Biología - SBG UNQ](#)'!

**RETO IX:** Si ahora queremos hacer nuestro programa un poco más estricto, por cada vuelta deberíamos sumar el total de células que tenemos e imprimir ese número en el mensaje. Entonces, por ejemplo, como en la primer vuelta tenemos dos células, imprimiríamos como mensaje '¡Somos 2 clones!', pero en la segunda vuelta serán en total 4 células y el mensaje a imprimir debería ser '¡Somos 4 clones!'. ¿Podrías escribir esta modificación del programa?

¡Compartinos tu código en nuestro grupo de Facebook '[Talleres de programación Orientada a la Biología - SBG UNQ](#)'! Buscanos en [Twitter](#)! (@BioinformaticaA)

**PISTA I:** Para la nueva variante del programa tené en cuenta lo que vimos antes para imprimir strings:

```
el_resultado = 'El resultado de la cuenta es %s'
print(el_resultado %5)
```

**PISTA II: Puede ser útil generar una variable antes del loop que se llame ‘células’, y que vaya tomando distintos valores en cada vuelta del loop.**

### Al infinito y más allá...

Como ya vimos los bucles **for** también se pueden escribir usando listas. En tal caso los valores que tome nuestra variable **i** serán los elementos de nuestra lista. Ahora bien, en general la construcción de programas para su uso científico implican creaciones más complejas. Nuestros programas deben no solo repetir ciertos pasos (un **bucle for**), sino que también tomar decisiones (**condiciones if**); y todo eso en una secuencia ordenada de pasos que le permita ir ejecutando ciertas órdenes condicionadas a lo que se encuentre en el camino. Esto es lo que se conoce como ‘**anidado**’: escribir dentro de un bucle **for** una condición **if**, tal que nuestro programa pueda tomar distintas decisiones según la situación o elemento con que se encuentra.

Veámos cómo usar el ‘anidado’ en un ejemplo concreto y sencillo. Algo que actualmente le quita el sueño a muchos científicos es la presencia de vida en otros planetas, que se estima está limitada a la presencia de agua líquida en dicho planeta. Imaginemos entonces, que dada la lista de planetas de nuestro sistema solar quisiéramos diferenciar los que no tienen ni rastro de agua de los que sí. Dada nuestra lista de planetas, si se conoce que uno de esos planetas presenta agua líquida en su superficie, le pediremos a nuestro programa que nos lo comunique. Recientemente se descubrió la presencia de agua líquida en Marte, por lo que nuestro programa debería resaltar tanto la Tierra como Marte como planetas con posibilidad de vida. Entonces nuestro programa debería tener los siguientes pasos:

- 1) Definir la lista de planetas (en cualquier orden, o por qué no, por su cercanía al sol).
- 2) Recorrer la lista de planetas:
  - a) si el planeta es Marte o la Tierra, imprimir ‘Planeta apto para la vida como la conocemos’
  - b) si, por el contrario, el planeta no es Marte o la Tierra debe imprimir: ‘No apto para la vida como la conocemos... ¿una vida distinta? quién sabe!’

Veamos entonces cómo quedaría el código:

```
planetas = ['Mercurio', 'Venus', 'Tierra', 'Marte',
            'Jupiter', 'Saturno', 'Urano', 'Neptuno']

for planeta in planetas:
    if planeta == 'Marte':
        print(planeta + ' es un planeta apto para la
        vida como la conocemos')
    if planeta == 'Tierra':
        print(planeta + ' es un planeta apto para la
        vida como la conocemos')
    else:
        print(planeta + ' es un planeta no apto para
        la vida como la conocemos... ¿Una vida distinta? ¡Quién
        sabe!')
```

Repasemos un poco lo que dice nuestro programa: para cada planeta de la lista de planetas (**for planeta in planetas**), si el planeta es Marte (**if planeta == 'Marte'**) imprimo el cartel distintivo de una planeta habitable (**print(i + 'es un planeta apto para la vida como la conocemos')**). Hago lo mismo si el planeta es la Tierra. En caso contrario, imprimimos un cartel esperanzador que nos recuerde que no hay agua en ese planeta, pero que siempre es posible hallar una biología con otras lógicas de funcionamiento (**print( i + ' es un planeta no apto para la vida como la conocemos... ¿Una vida distinta? ¿Quién sabe!')**). Prestá atención a los espacios luego de los dos puntos (:) del bucle **for** y de la sentencia **if**; recordá que estos espacios son importantes para que Python siga en un orden correcto las órdenes que le damos. En caso de no respetar estos espacios, Python nos hará saber que está disconforme y nos arrojará un error: **IndentationError: expected an indented block**.

**RETO X: ¿Te animás a pensar tus propios programas? ¿Estás listo para un desafío? ¡Anotate en el Concurso de Bioinformática para escuelas secundarias! Además, no te olvides de realizar la encuesta así sabemos cómo mejorar nuestros talleres.**

**¡Compartinos tu código en nuestro grupo de Facebook 'Talleres de programación Orientada a la Biología - SBG\_UNO'! Buscanos en Twitter! (@BioinformaticaA)**

### ¿Qué más podemos hacer con Python?

**Python** es un lenguaje muy versátil que nos permite hacer desde simulaciones y programas con cálculos complejos, hasta aplicaciones web y programas de celular, pasando por minería de datos, etc. Todo es de libre acceso, tanto librerías como tutoriales (en la página de [Python Argentina](#), entre otras, podés encontrar libros, tutoriales, scripts, ¡lo que necesites!). Siempre pueden consultarnos ante cualquier duda. ¡Recordá que el truco del buen programador es saber buscar en Google! Todos los errores que te aparezcan al ejecutar tus programas podés buscarlos en la web o en páginas como [Stack Overflow](#), una página donde programadores de todos los lenguajes hacen consultas y se puede encontrar información.

### Un último secreto!

¿Es posible pasar del “línea a línea” a un programa o script? Claro que si!! El primer paso para ello es unir todos los comandos y órdenes en un archivo de texto. Por lo que será necesario identificar cómo abrir el editor de texto, como guardar el script que creamos y cómo ejecutar nuestro script. De esta manera además no necesitamos volver a escribir nuestro programa cada vez que lo necesitemos!

**PARA TENER EN CUENTA: En el caso de iOS la aplicación corre a través de la ejecución scripts, por lo que esta sección es innecesaria.**

Veamos un ejemplo de cómo hacerlo con QPython3:

1. Abrimos el editor de la aplicación desde la pantalla principal.
2. Se abrirá un editor de texto donde podremos escribir el script que queremos ejecutar.

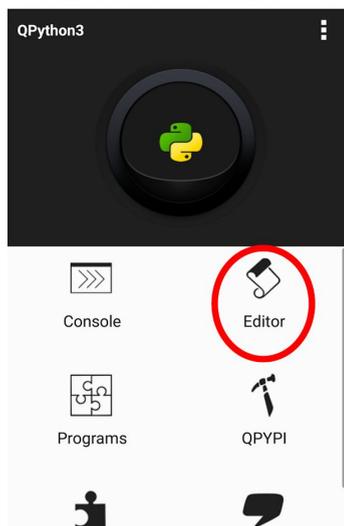
Escribe por ejemplo:

```
a = 2
c = 3
print(a+c)
```

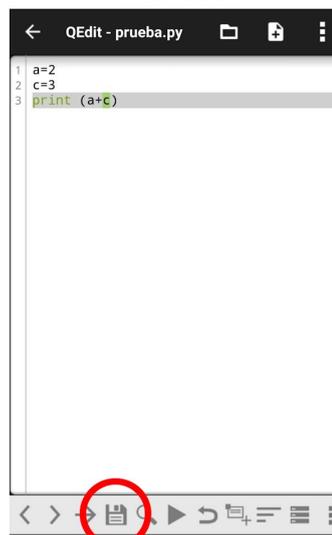
3. Es importante que recordemos guardarlo en una carpeta para poder usarlo en otro momento.

Para guardarlo en una carpeta se utiliza el botón con forma de diskette. 

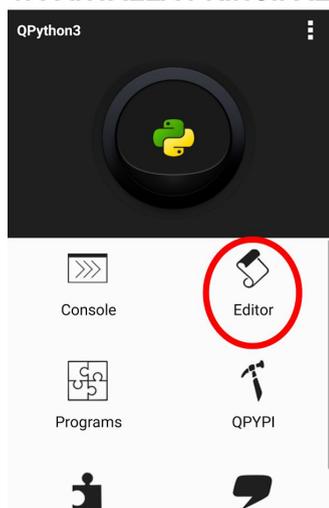
### PANTALLA PRINCIPAL



### EDITOR



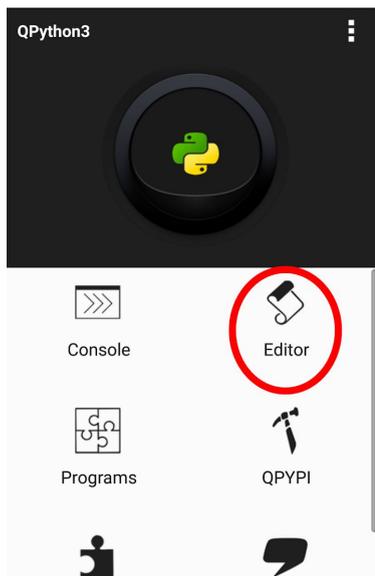
### 1. PANTALLA PRINCIPAL



### 2. EDITOR



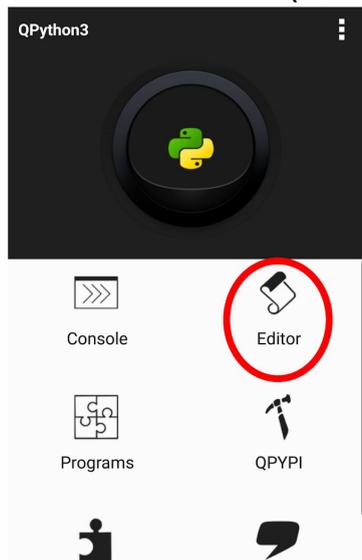
### PANTALLA PRINCIPAL



### EDITOR



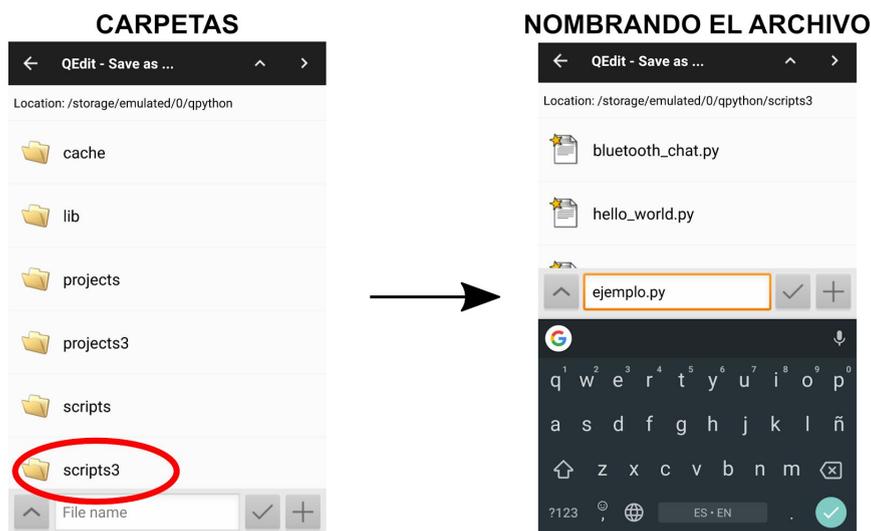
### PANTALLA PRINCIPAL (PASO 1)



### EDITOR (PASO 2 y 3)



- Guarda tu script en una carpeta, por ejemplo, "scripts\_3". . **¡Atención!** Siempre al final del nombre que demos a nuestro script debemos usar la extensión ".py", como en el ejemplo de la foto: "ejemplo.py"



5. Por último, si queremos utilizar nuestro script debemos ejecutarlo y eso se hace desde la página principal; para volver a la página principal utiliza la flecha que se encuentra en la esquina superior izquierda. Cuando estemos en la página principal, apretamos el botón “Programas” y navegamos en los archivos hasta la carpeta donde guardamos nuestro script. Al seleccionarlo aparecerán las opciones que se muestran en la imagen, de las cuales debemos elegir y pulsar “Run”. ¡Tachan! ¡El script se ejecuta!



### Nos veremos otra vez...

Bueno, ahora si!!! Hasta acá llegamos con el taller, les agradecemos su interés y participación. Agradecemos a las escuelas que nos abrieron sus puertas y a los/las profes que nos hicieron un lugar en sus aulas, quienes hicieron de este modo que el proyecto se pueda llevar a cabo. Te invitamos a acercarse a nuestro Grupo y visitar nuestra página siempre que deseen. ¡Esperamos su participación en el próximo Concurso de Bioinformática para Escuelas Secundarias!

**Gracias ¡totales!**

Agradecimientos de los autores a Lao Tse, Antoine de Saint-Exupéry, Gilda, Tan Biónica, La Portuaria, Homero Simpson, Carlos Varela, Luis Alberto Spinetta, Maquiavelo , Charly García y Gustavo Cerati por las frases que inspiraron nuestros títulos.

Ante cualquier duda podés contactar por correo electrónico a Ana Julia Velez Rueda ([anavelezrueda@gmail.com](mailto:anavelezrueda@gmail.com)), coordinadora del proyecto, y a cualquiera de los docentes de nuestros talleres. Encontranos también a través de la página de nuestro grupo, el SBG o Grupo de Bioinformática Estructural de la Universidad Nacional de Quilmes: <http://ufg.unq.edu.ar/sbg/education/index.html> .