prog_bio_3

July 18, 2018

1 Programación para la Bioinformática

1.1 Unidad 3: Conceptos avanzados de Python

1.1.1 Instrucciones de uso

En el módulo anterior introducimos conceptos básicos sobre variables y su uso en Python. En este módulo estudiaremos conceptos más avanzados como son las instrucciones de flujo de ejecución (for, while, if), cómo definir y utilizar funciones, etc.

1.2 Iteración y operaciones lógicas

En la gran mayoría de ocasiones, tendremos que manipular nuestros datos y para ello utilizaremos los conceptos de iteración y las operaciones lógicas. Las operaciones lógicas nos permiten comparar valores entre variables (mayor, menor, igualdad) y la iteración **ir visitando uno a uno los elementos de una lista, tupla, diccionario o cualquier estructura de datos que sea susceptible de secuenciar**.

Vamos a empezar por visitar las operaciones lógicas:

```
In [1]: # Las operaciones lógicas tendrán como resultado un valor
        # cierto (True) o falso (False):

        a = 5
        b = 1

        # £Es el valor de la variable 'a' mayor que el de la
        # variable 'b'?
        print(a > b)
```

```
True
```

Estos valores de cierto (True) o falso (False) reciben el nombre de *booleanos* en la jerga técnica.
Veamos otros ejemplos de comparaciones:

```
In [2]: # £Es el valor de la variable 'a' menor que el
        # de la variable 'b'?
        print(a < b)
```

```
False
```

```
In [3]: # Cambiamos el valor de la variable ahora a 5:
        b = 5

        # £Es el valor de la variable 'b' igual al de
        # la variable 'a'?
        print(b == a)
```

```
True
```

```
In [4]: # Otros operadores lógicos disponibles son menor o
        # igual '<=', mayor o igual '>=' o la negación 'not':
        print(a <= b)
        print(a >= b)

        a = False
        print(not a)
```

```
True
True
True
```

1.2.1 Estructuras de control

También podemos alterar el flujo de ejecución de nuestro programa utilizando las estructuras *if...else* o *if...elif...else*. Estas estructuras las utilizaremos para comprovar si se cumple una condición determinada y reaccionar en el caso. Veamos unos cuantos ejemplos:

```
In [5]: a = 5
        b = 6
        if a > b:
            print('a es mayor que b')
        else:
            print('a es menor o igual que b')
```

```
a es menor o igual que b
```

En el ejemplo anterior, primero de todo asignamos el valor entero 5 a la variable *a* y el valor entero 6 a la variable *b*. A continuación, *if a* > *b* comprueba si el valor que contiene la variable *a* es mayor que el valor de la variable *b*. Si fuera así, se mostraría por pantalla *a es mayor que b*. Como no es el caso (*a* > *b* es **Falso**), se ejecutará la parte incluída en el bloque *else*.

Vamos a ver ahora un ejemplo en el que se comprueban varias condiciones de forma secuencial:

```
In [6]: a = 5
        b = 5
        if a > b:
            print('a es mayor que b')
        elif a < b:
            print('a es menor que b')
        else:
            print('a es igual a b')
```

```
a es igual a b
```

En este ejemplo, las variables *a* y *b* contienen el mismo valor, 5. Primero se comprueba la condición de si *a es mayor que b*. Como no es el caso, se ejecuta la siguiente condición, $a < b$. Tampoco es el caso, así que se ejecuta la última directiva que está incluída en el bloque *else*.

1.2.2 Iteraciones en Python

En Python existen solo dos formas de iterar una secuencia: mediante **for** o mediante **while**. La primera de las opciones, *for*, iterará uno por uno los elementos contenidos en una lista. En el caso de *while*, iteraremos mientras la condición de permanencia en el bucle se cumpla. Veamos unos ejemplos para entender estos dos nuevos conceptos:

Bucle for

```
In [7]: # Declaramos una lista que contiene cuatro palabras:
        monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

        # Primer método iterando mediante for:
        for monster in monsters:
            print(monster)
```

```
Kraken
Leviathan
Uroborus
Hydra
```

Este primer ejemplo de bucle *for* nos muestra lo fácil que es interar en Python uno por uno todos los elementos de una lista. Fijaos que primero hemos declarado una lista de nombre *monsters* que contiene cuatro palabras. A continuación, la línea *for monster in monsters* nos indica que por cada uno de los elementos que encontremos en la lista *monsters*, los asignaremos uno a uno a la variable *monster*. Por cada uno de los valores que vamos guardando en la variable *monster* (primero 'Kraken', después 'Leviathan' y así hasta acabar la lista), los monstraremos por pantalla utilizando el comando *print monster* porque está indentado un nivel por debajo de la línea *"for monster in monsters:*".

Veamos un segundo ejemplo del uso del bucle *for*:

```
In [8]: # Declaramos una lista que contiene cuatro palabras:
        monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

        # Segundo método. La función especial 'enumerate' nos
        # devuelve una tupla en el que el primer elemento es un
        # índice que empieza en 0 y aumenta de 1 en 1 y el
        # segundo elemento el valor de la posición en la lista:
        for i, monster in enumerate(monsters):
            print(i, monster)
```

```
0 Kraken
1 Leviathan
2 Uroborus
3 Hydra
```

Este segundo ejemplo de bucle *for* es especialmente interesante al hacer uso de la función *enumerate()*. Es muy probable que en ocasiones queramos saber el contador de un determinado elemento en una lista. La función *enumerate()* nos sirve especialmente para eso y no tenemos que encargarnos de mantener un variable que nos sirva de contador a cada vuelta del bucle.

Bucle while El bucle *while* es una segunda forma de iterar en Python y podemos construir expresiones idénticas al bucle *for*, pero con una sintaxis ligeramente diferente:

```
In [9]: # Declaramos una lista que contiene cuatro palabras:
        monsters = ['Kraken', 'Leviathan', 'Uroborus', 'Hydra']

        # Este ejemplo tiene un comportamiento idéntico al
        # anterior, pero necesitamos escribir más operaciones para
        # un mismo resultado. Normalmente preferiremos utilizar
        # for y no while:

        # Declaramos la variable contador 'i' que guardará a cada
        # paso del bucle el número de vueltas
        # del bucle:
        i = 0

        # Mientras que el contador 'i' sea menor que la longitud
        # de la lista 'monsters':
        while i < len(monsters):
            # Imprime el valor de la lista en la posición 'i'
            print(i, monsters[i])
            # No nos olvidemos de actualizar el valor de 'i'
            # sumándole 1 o tendremos un bucle infinito
            i += 1
```

```
0 Kraken
1 Leviathan
```

```
2 Uroborus
3 Hydra
```

En este momento seríamos capaces de calcular la serie de Fibonacci hasta un determinado valor:

```
In [20]: # Calculamos el valor de la serie hasta un
         # valor n=100
         n = 100

         # Asignamos dos valores a la vez, 0 a la variable
         # 'a' y 1 a la variable 'b'
         a, b = 0, 1

         # Ahora vamos a iterar mientras se cumpla la condición
         # de que la variable 'a' contiene un valor más pequeño
         # que el de la variable 'n':
         while a < n:
             print(a, end=" ")
             # Cuando añadimos el parámetro 'end', evitamos
             # que se imprima un salto de línea
             a, b = b, a+b
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

La función range() En Python disponemos de una función muy útil para generar una secuencia de números, que podemos utilizar de diferentes formas:

```
In [25]: # Podemos utilizarla para iterar:
         for i in range(10):
             print(i, end=" ")
         print()
```

```
0 1 2 3 4 5 6 7 8 9
```

```
In [26]: # Podemos definir el rango de acción. Por ejemplo,
         # calcula los números entre 5 y 20 de 3 en 3:
         for i in range(5,20,3):
             print(i, end=" ")
         print()
```

```
5 8 11 14 17
```

Iterando un diccionario Es posible, dado un diccionario, iterar entre los pares (clave, valor) almacenados:

```
In [28]: # Dado el siguiente diccionario de códigos de teléfono:
         country_codes = {34: 'Spain', 376: 'Andorra',
                          41: 'Switzerland', 424: None}

         # Iteramos por clave:
         print("Iterando por clave:")
         for country_code in country_codes.keys():
             print(country_code)
         print()

         # Iteramos por valor:
         print("Iterando por valor:")
         for country in country_codes.values():
             print(country)
         print()

         # Iteramos por los dos a la vez:
         print("Iterando por clave y valor a la vez:")
         for country_code, country in country_codes.items():
             print(country_code, country)
```

```
Iterando por clave:
34
376
41
424

Iterando por valor:
Spain
Andorra
Switzerland
None

Iterando por clave y valor a la vez:
34 Spain
376 Andorra
41 Switzerland
424 None
```

1.3 Funciones

Otra forma muy importante de organizar el flujo de ejecución es encapsulando cierta porción de código en una función que podamos llamar en diferentes partes del código.

Una función en Python utiliza el mismo concepto que una función matemática. Por ejemplo, imaginemos la función matemática:

$suma(x, y) = x + y$

En Python podemos definir la misma función de la siguiente forma:

```
In [30]: # La función suma se define mediante la palabra especial
         # 'def' y tiene dos argumentos: 'x' e 'y':
         def suma(x, y):
             # Devolvemos el valor de la suma
             return x + y

         # En este momento, podemos llamarla con cualquier valor:
         print("2 + 4 =", suma(2, 4))
         print("5 + (-5) =", suma(5, -5))
         print("3.5 + 2.5 =", suma(3.5, 2.5))
```

```
2 + 4 = 6
5 + (-5) = 0
3.5 + 2.5 = 6.0
```

```
In [31]: # Podemos definir una función que no haga nada
         # utilizando la palabra especial 'pass':

         def dummy():
             pass

         dummy()
```

```
In [32]: # Podríamos volver a definir el trozo de código de la
         # secuencia de Fibonacci como una función:

         def fibonacci(n=100):
             a, b = 0, 1
             while a < n:
                 print(a, end=" ")
                 a, b = b, a+b

         print('El valor de los primeros números de la secuencia de Fibonacci hasta 10 son:')
         fibonacci(10)
```

```
El valor de los primeros números de la secuencia de Fibonacci hasta 10 son:
0 1 1 2 3 5 8
```

```
In [33]: # En el anterior ejemplo, hemos definido que el argumento
         # n tenga un valor por defecto. Esto es muy útil
         # en los casos en los que utilicemos la función siempre con
         # un mismo valor, queramos dejar constancia de un
         # caso de ejemplo o por defecto. En este caso, podemos
         # ejecutar la función sin pasarle ningún valor:
         fibonacci()
```

```
0 1 1 2 3 5 8 13 21 34 55 89
```

```
In [35]: # Podemos definir parte de los argumentos con valores
         # por defecto y otro sin.
         # Los argumentos sin valor por defecto siempre han de
         # estar más a la izquierda en la definición de la función:

         def potencia(a, b=2.):
             # Por defecto, elevaremos al cuadrado
             return a**b

         print("3 elevado al cuadrado es:", potencia(3))
         print("2 elevado al cubo es:", potencia(2, 3))
```

```
3 elevado al cuadrado es: 9.0
2 elevado al cubo es: 8
```

Recomendamos la lectura de la documentación oficial para acabar de fijar conocimientos: https://docs.python.org/2/tutorial/controlflow.html

1.4 Leer y escribir desde ficheros

Una tarea habitual es leer líneas de texto de un fichero o escribir líneas de texto en un fichero. En Python, leer y escribir ficheros se hace a través de la librería *os*. Una librería es un conjunto de código que tiene cierto sentido agrupar para ser usado por otros usuarios. En nuestro caso, *os* se trata de una librería que agrupa funciones relacionadas con el sistema operativo (**o**perating **s**ystem).

Para cargar una librería, utilizaremos la palabra reservada **import**. Más adelante explicaremos algunas particularidades de utilizar e importar librerías. A continuación os explicamos cómo escribir y leer un fichero:

1.4.1 Escribir en un fichero

Primero de todo, escribiremos en un fichero que después podremos leer en el siguiente apartado:

```
In [36]: # Importamos la librería 'os':
         import os

         # Abrimos un fichero de nombre 'a_file.txt' para
         # escritura (de ahí la 'w', 'writing').
         # Atención a la instrucción especial with.
         # Asignamos el descriptor de fichero a una variable
         # de nombre out:
         with open('a_file.txt', 'w') as out:
             # Vamos a escribir 10 líneas, cada una con un número de 0 a 9
             for i in range(10):
                 # La siguiente línea escribe en el fichero todo
                 # lo que pongamos dentro de out.write()
```

```
        # En nuestro caso, es un string del tipo '0\n', '1\n',
        # etc. Esto lo conseguimos utilizando las wildcards %d y %s.
        # %s representa un string o cadena de caracteres
        # y %d un número entero.
        # Concatenamos en este caso un número con un string
        # que se trata del salto de línea, os.linesep, que
        # equivale en Linux a '\n':
        out.write("Línea %d%s" % (i, os.linesep))
```

Después de ejecutar el código anterior, deberíamos encontrar un fichero en la misma ruta que este fichero de notebook de nombre 'a_file.txt'. Si lo abrimos, veremos que efectivamente contiene 10 líneas:

```
$ cat a_file.txt
Línea 0
Línea 1
Línea 2
Línea 3
Línea 4
Línea 5
Línea 6
Línea 7
Línea 8
Línea 9
```

1.4.2 Leer de un fichero

Ahora vamos a hacer el paso contrario, vamos a leer el contenido del fichero que acabamos de generar. Lo haremos de tres formas distintas, equivalentes entre sí:

Leyendo de un fichero (I)

```
In [38]: f = open('a_file.txt')
         for line in f:
             print(line)
         f.close()

Línea 0

Línea 1

Línea 2

Línea 3

Línea 4

Línea 5
```

```
Línea 6

Línea 7

Línea 8

Línea 9
```

Leyendo de un fichero (II)

```
In [39]: f = open('a_file.txt')
         lines = f.readlines()
         f.close()
         for line in lines:
             print(line)

Línea 0

Línea 1

Línea 2

Línea 3

Línea 4

Línea 5

Línea 6

Línea 7

Línea 8

Línea 9
```

Leyendo de un fichero (III)

```
In [40]: with open('a_file.txt') as f:
             for line in f:
                 print(line)

Línea 0
```

```
Línea 1

Línea 2

Línea 3

Línea 4

Línea 5

Línea 6

Línea 7

Línea 8

Línea 9
```

1.5 Organización del código

Un módulo de Python es cualquier fichero con extensión *.py* que esté bajo la ruta del path de Python. El path de Python puede consultarse importando la librería sys:

```
In [ ]: import sys
        print(sys.path)
```

Por defecto, Python también mira las librerías que se hayan definido en la variable de entorno $PYTHONPATH (esto puedo cambiar ligeramente en un entorno Windows, https://docs.python.org/2/using/cmdline.html).

Un paquete en Python es cualquier directorio que contenga un fichero especial de nombre *__init__.py* (este fichero estará vacío la mayoría de veces).

Un módulo puede contener diferentes funciones, variables u objetos. Por ejemplo, definamos un módulo de nombre *prog_bio.py* que contenga:

```
In [43]: # prog_bio.py

         PI = 3.14159265

         def suma(x, y):
             return x + y


         def resta(x, y):
             return x - y
```

Para utilizar desde otro módulo o script estas funciones, deberíamos escribir lo siguiente:

```
In [ ]: from prog_bio import PI, suma, resta

        # Y entonces podríamos utilizarlas normalmente
        siete = suma(2,5)
```

En Python también podemos utilizar la directiva *from prog_bio import **, pero su uso está **totalmente desaconsejado**. La razón es que estaríamos importando gran cantidad de código que no utilizaremos (con el consiguiente aumento del uso de la memoria), pero además, podríamos tener colisiones de nombres (funciones que se llamen de la misma forma en diferentes módulos) sin nuestro conocimiento. A no ser que sea imprescindible, no utilizaremos esa directiva e importaremos una a una las librerías y sus funciones que vayamos a necesitar.

Podéis aprender más sobre importar librerías y definir vuestros propios módulos aquí: http://life.bsc.es/pid/brian/python/#/5

1.6 Más ejemplos

Disponéis de más ejemplos online de manejo de ficheros en el siguiente enlace:

Python recipes for Bioinformatics beginners