Preguntas teóricas

1. ¿Qué es un condicional?

   Un condicional (`if`) es un tipo de proposición que permite ejecutar un "bloque de código" sujeto a una determinada condición. Dicho tipo de proposiciones posee la siguiente estructura formal: si la condidición es evaluada `True` entonces se ejecuta el "bloque de còdigo", en caso que sea `False`, se ignorará el bloque (es decir, no se imprimirá nada en el terminal).

   ```
   precio = 50
   if precio < 70:
       print("precio es menor que 70")
   ```

   Ahora bien, *no* es necesario colocar una sola condición en un *script*; se pueden colocar 2 o más condiciones, buscando abarcar los distintos escenarios posibles. Para ello se utliza la clausula `elif`, una síntesis entre las clausulas `else` e `if`.

   ```
   precio = 70
   if precio < 70:
       print("precio es menor que 70")
   elif precio == 70:
       print("precio es igual que 70")
   ```

   Junto con una proposición `if` se puede colocar también una condición `else` que permita, a su vez, definir un "bloque de código" ejecutable en caso que la condición `if` sea evaluada como `False`.

   ```
   precio = 70
   if precio == 70:
     print("precio es igual a  70")
   else:
     print("precio es distinto que 70")
   ```

2. ¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

   En *python* existen 2 grandes tipos de bucles o *loops*: *for* y *while*. A veces se menciona un tercer tipo de bucle, *nested loop*[1], pero si se lo entiende como un bucle "dentro" de otro, no parece constituir un bucle tan básico o fundamental como los otros dos.

[1] Véase, por ejemplo, https://www.codecademy.com/resources/docs/python/loops.

- Bucle *while*

  Dicho tipo de bucle se caracteriza por ejecutar un código una y otra vez, hasta que la(s) condicion(es) especificada(s) sea(n) falsa(s). A continuación analizaremos algunos ejemplos para aclarar un poco más dicha caracterización.

```
a = 1
while a < 10:
  print("¡Hola mundo!")
```

  Si ejecutamos este código, veremos que el programa imprimirá *infinítamente* el mensaje que aparece ahí. ¿Por qué infinitamente? Básicamente, porque dicho código posee dos *proposiciones* que de alguna manera no se "cruzan" nunca. Por un lado, una primera proposición que define el valor inicial de `a` como una constante e igual a `1`. Y por el otro lado, una proposición *condicional* que plantea que, mientras `a` sea menor que `10`, se imprima el mensaje que aparece entre comillas. Dado que `a` es definida como una constante o una variable estática, es decir, que no varía su valor luego de cada bucle, `a` nunca será mayor que `10`, haciendo que el valor de verdad de dicha condición sea siempre verdadera.

  Para que no suceda lo anterior, a saber, una iteración que tienda al infinito[2], es necesario que el bucle cuente con lo que se llama un *contador*. Aquello permite que la variable se comporte de manera dinámica, para decirlo de alguna manera, aumentando o disminuyendo después de cada bucle.

```
a = 1
while a < 10:
    print("¡Hola mundo!")
    a = a + 1
```

  Al ejecutar dicho código, veremos que la iteración se lleva a cabo en este caso solo en nueve ocasiones. ¿Por qué? Porque al incorporar un *contador* en el código, en cada *ciclo* del bucle la variable `a` aumentará en uno, hasta el punto en que `a` sea igual a `10`, haciendo que la condición formulada sea falsa y, de ese modo, finalice el bucle.

- Bucle *for*

  Un bucle tipo *for* se utiliza, a su vez, para iterar en general sobre una sequencia: lista, tuple, dictionario, conjunto y/o *string*. Veámos a continuación algunos ejemplos.

[2] Claramente un *script* o código que "dé vueltas en redondo" presenta un error.

```
frutas = ["manzanas", "peras", "fresas"]
for x in frutas:
    print(x)
```

Si ejecutamos dicho bucle en el terminal, veremos que se imprime hacia abajo el nombre de cada fruta incluida en el lista.

```
manzanas
peras
fresas
```

Algo parecido resulta cuando intentamos realizar un bucle sobre un determinado *string*.

```
for x in "frutas":
    print(x)
```

En este caso, si ejecutamos el bucle, veremos que se imprime en el terminal "hacia abajo" cada letra del término *frutas*.

```
f
r
u
t
a
s
```

Es importante señalar que en ambos tipos de bucles (*for* y *while*) es posible parar o "quebrar" la iteración antes que ella finalece utilizando la siguiente expresión: `break`.

```
frutas = ["manzanas", "peras", "fresas"]
for x in frutas:
  if x == "peras":
    break
  print(x)
```

Como se puede observar en el ejemplo anterior, para realizar dicho `break`, es necesario introducir una proposición *condicional* (`if`) que señale en qué `item` se realizará el quiebre. Es importante tener presente que al

imprimir un bucle que incorpore un `break`, se imprimirán todos los `items` o elementos justo hasta aquel que preceda o anteceda el del quiebre[3].

Con la expresión `continue` es posible, a su vez, *detener* la iteracion en "curso", para pasar a la siguente. Al igual que en el caso de `break`, es posible hacerlo para ambos tipos de bucles (*for* y *while*).

```
frutas = ["manzanas", "peras", "fresas"]
for x in frutas:
  if x == "peras":
    continue
  print(x)
```

En este caso, solo se imprimirán en el terminal las `manzanas` y las `fresas`, ya que al momento de iterar sobre las `peras`, el código ordena pasar al próximo item antes de poder imprimirlo.

```
manzanas
fresas
```

3. ¿Qué es una lista por comprensión en Python?

   Una manera de entender en qué consiste una lista de comprensión es decir que se trata, básicamente, de un tipo de sintaxis *corta* para crear una nueva lista a partir de una lista *ya* existente.

   Ahora bien, para aclarar un poco más esta caracterización general, quizás lo mejor sea ver un pequeño ejemplo. Definiremos una pequeña lista, `frutas`, a partir de la cual crearemos una `nueva_lista` en función de una determinada condición, a saber, que esté formada por términos que contengan al menos una vez la letra `e`. Existen dos maneras para crear la `nueva_lista`. Veamos cada una por separado.

```
frutas = ["manzanas", "peras", "fresas", "kiwis"]
nueva_lista = [ ]
```

   En caso de *no* querer crear la `nueva_lista` utilizando la sintáxis de la "lista de comprensión", tendríamos que emplear un bucle `for` con una proposición condicional `in` en su interior. Como se ve en el cuadro que presentamos a continuación, aquello implica escribir un código de tres líneas para llevar a cabo ese objetivo.

[3] En sentido estricto, eso sí, aquello dependerá si la orden de impresion se encuentra antes o después de la proposición condicional.

```
for x in frutas:
  if "e" in x:
    nueva_lista.append(x)
print(nueva_lista)
```

Mucho más corto y, por ende, eficiente resulta la sintáxis de la "lista de comprensión". En *una* sola línea es posible crear la `nueva_lista` a partir de la anterior. Aquello puede que no sea muy relevante en un código pequeño, pero cuando se trata de uno grande, sí resulta importante.

```
nueva_lista = [x for x in frutas if "a" in x]
print(nueva_lista)
```

4. ¿Qué es un argumento en Python?

   Un *argumento* es el nombre o término que aparece cuando "llamamos" a una función para ejecutarla. Es importante distinguirla del *parámetro*, que corresponde más bien al término o nombre que aparece al inicio cuando "definimos" una función. Si bien ambos términos se utilizan de manera indistinta (un uso habitual pero, en sentido estricto, incorrecto), es necesario tener presente la distinción. Veámos un ejemplo para *ver* con más claridad dicha diferencia y, de ese modo, entender mejor qué es un argumento.

```
def add_num(a, b):
    sum = a + b
    print(sum)
add_num(3, 4)
```

   Definimos (`def`) en primer lugar una función llamada `add_num` que, como se ve en la línea siguiente, simplemente suma dos términos (`a`, `b`). Dichos términos reciben el nombre de *parámetros* cuando definimos la función. Ahora bien, cuando a dichos términos le asignamos un valor específico para querer ejecutar la función, `3` y `4` en nuestro ejemplo, ellos adoptan el nombre de *argumento*.

5. ¿Qué es una función Lambda en Python?

   Una función lambda es, lo que se llama, un función *anónima*. ¿Qué significa eso? Básicamente, que *no* tiene nombre. Ahora bien ¿qué consecuencia tiene eso? ¿Cuál es la utilidad o beneficio de eso? Que la puedo ejecutar simplemente invocando la variable que la define, ya que ella misma almacena la función en sí. Analicemos un par de ejemplos: una función *normal* y luego una *lambda*.

```
def sum(a):
    x = a + 10
    print(x)
sum(5)
```

En principio, si se quiere crear una función es necesario definirla (`def`) asignándole un nombre (`sum`) y uno(s) parámetros (`a`). Luego se requiere precisar la función propiamente tal en términos de una expresión formada por los parámtro(s) previamente definido(s), junto con las operacion(es) involucrada(s) (`a + 10`) en ella. Dicha expresión la identificamos, a su vez, con una variable (`x`) de manera de poder referenciarla posteriormente. Por último, en el caso que queramos "ejecutar" e "imprimir" dicho función, debemos invocar la orden `print(x)` y luego "llamar" a la función, asignándole un valor al parametro o un argumento (`5`).

```
x = lambda a : a + 10
print(x(5))
```

En el caso de la función lambda, ya que se trata de una función *anónima* el proceso de definición y ejecución resulta mucho más corto y directo. Basta simplemente con identificar el o los parámetros de la función (`a`) junto con la expresión que la caracteriza (`a + 10`). Luego, al querer ejecutarla e imprimirla, debemos simplemente invocar la orden `print` haciendo referencia a la variable que la contiene (`x`) junto con el argumento que queramos evaluar (`5`).

6. ¿Qué es un paquete pip?

   Se llama *pip* en python a un "sistema de gestion de paquetes". Su función principal es instalar y administrar los paquetes existentes de *python*. Dichos paquetes son obtenidos del Python Package Index, aunque es posible descargarlos de otros índices tambien[4].

   Una de las grandes ventajas de *pip* es la facilidad con la cual se pueden instalar y desinstalar los paquetes de software desde la "línea de comandos": basta con un simple comando como el siguiente para hacerlo:

```
C:> py -m pip install nombre-paquete
```

   Tal como está señalado en la documentación de *pip*, para poder empezar a utilizarlo, es necesario tenerlo *ya* instalado en nuestro sistema junto

[4] El índice de paquetes de Python (PyPI) es un gran repositorio de *software* para el lenguaje de programación python.

con *python*. Podemos revisar si aquello es así a través de los siguientes comandos[5]:

```
C:> py --version
Python 3.N.N
C:> py -m pip --version
pip X.Y.Z from ... (python 3.N.N)
```

Ahora bien, en el caso que *pip* no se encuentre ya instalado en su sistema, existen dos maneras de hacerlo:

- `ensurepip`
- `get-pip.py`

En el caso de utilizar `get-pip.py`, es necesario ejecutar los dos siguientes pasos:

- Descargar el *script* desde https://bootstrap.pypa.io/get-pip.py.
- Abrir el terminal, luego `cd` a la carpeta que contiene el archivo recién descargado `get-pip.py`, y finalmente ejecutar:

  ```
  C:> py get-pip.py
  ```

Una vez *ya* instalado el programa *pip* en nuestro sitema, podemos entonces comenzar a descargar los distintos paquetes que necesitemos mediante el comando inicial.

[5] Dependiendo del sistema operativo que tengamos en nuestro ordenador, sea Linux, MacOS o Windows, el comando empleado podrá variar un poco. En el siguiente cuadro solo he colocado los comandos utilizados en Windows.