prog_datasci_3_python_entrega_cast

October 5, 2021

1 Fundamentos de Programación

1.1 PAC 3 - Enunciado

En este Notebook se encontraréis el conjunto de actividades evaluables como PEC de la asignatura. Veréis que cada una de ellas tiene asociada una puntuación, que indica el peso que tiene la actividad sobre la nota final de la PEC. Adicionalmente, hay un ejercicio opcional, que no tiene puntuación dentro de la PEC, pero que se valora al final del semestre de cara a conceder las matrículas de honor y redondear las notas finales. Podréis sacar la máxima nota de la PAC sin necesidad de hacer este ejercicio. El objetivo de este ejercicio es que sirva como pequeño reto para los estudiantes que quieran profundizar en el contenido de la asignatura.

Veréis que todas las actividades de la PEC tienen una etiqueta, que indica los recursos necesarios para llevarla a cabo. Hay tres posibles etiquetas:

- NM **Sólo materiales**: las herramientas necesarias para realizar la actividad se pueden encontrar en los materiales de la asignatura.
- EG **Consulta externa guiada**: la actividad puede requerir hacer uso de herramientas que no se encuentran en los materiales de la asignatura, pero el enunciado contiene indicaciones de dónde o cómo encontrar la información adicional necesaria para resolver la actividad.
- EI **Consulta externa independiente**: la actividad puede requerir hacer uso de herramientas que no se encuentran en los materiales de la asignatura, y el enunciado puede no incluir la descripción de dónde o cómo encontrar esta información adicional. Será necesario que el estudiante busque esta información utilizando los recursos que se han explicado en la asignatura.

Es importante notar que estas etiquetas no indican el nivel de dificultad del ejercicio, sino únicamente la necesidad de consulta de documentación externa para su resolución. Además, recordad que las **etiquetas son informativas**, pero podréis consultar referencias externas en cualquier momento (aunque no se indique explícitamente) o puede ser que podáis hacer una actividad sin consultar ningún tipo de documentación. Por ejemplo, para resolver una actividad que sólo requiera los materiales de la asignatura, puedéis consultar referencias externas si queréis, ya sea tanto para ayudaros en la resolución como para ampliar el conocimiento!

En cuanto a la consulta de documentación externa en la resolución de los ejercicios, recordad **citar siempre la bibliografía utilizada** para resolver cada actividad.

1.2 Ejercicios para la PEC

A continuación encontraréis los ejercicios que se deben completar en esta PEC y que forman parte de la evaluación de esta unidad.

1.2.1 Ejercicio 1

Dada la lista:

[ ]:
```
lista=[10, "python", 3 , "UOC", True]
```

a) Comenta los dos códigos siguientes: a1) y a2). ¿Qué diferencia hay entre utilizar `break` o `continúe`? Razona la respuesta. NM **(0.5 puntos)**

[ ]:
```
#a1)

for e in lista:
    if e == "UOC":
        break
    print(e)
```

[ ]:
```
#a2)

for e in lista:
    if e == "UOC":
        continue
    print(e)
```

[ ]:
```
# Respuesta
```

b) Escribe un código que, dada la lista que estamos trabajando, 1) detecte los elementos string de la lista, y 2) los muestre por pantalla. NM **(0.5 puntos)**

[ ]:
```
# Respuesta
```

c) Escribe un código que, dada la lista que estamos trabajando, 1) detecte los elementos de la lista que són un número, 2) identifique si son un numero primo o no, y 3) muestre por pantalla un mensaje informando de los resultados. Por ejemplo, para la lista que hemos definido al inicio del ejercicio, el código podría mostrar el siguiente resultado:

10 is not a prime number

3 is a prime number

Nota: Podéis consultar como detectar si un número es primo en este post de stack overflow.

EG **(1 punto)**

[ ]:
```
# Respuesta
```

1.2.2 Ejercicio 2

Python dispone de un idiom muy útil conocido como **list comprehension** y esto es lo que vamos a trabajar en este ejercicio a partir de la lista del ejercicio anterior:

[ ]:
```
lista=[10, "python", 3 , "UOC", True]
```

Nota: Para realizar esta actividad necesitaréis investigar qué son las *list comprehension* y qué sintaxis utilizan. Para ello, se recomienda en primer lugar que utilicéis un buscador para encontrar información genérica sobre esta construcción. Después, os recomendamos que consultéis stack-overflow (un sitio de preguntas-y-respuestas muy popular entre programadores) para ver algunos ejemplos de problemas que se pueden resolver con esta construcción:

https://stackoverflow.com/questions/12555443/squaring-all-elements-ina-list

https://stackoverflow.com/questions/57166908/using-list-comprehension-i-want-to-printodd-even-with-string-indicating-even/57167016

a) Recorre los elementos de la lista mediante *list comprehension* y luego imprímelos por pantalla. **(0.25 puntos)** EG

b) Utiliza una *list comprehension* para detectar los strings de la lista y luego muestra el resultado por pantalla. **(0.25 puntos)** EG

c) De la misma manera, utiliza una *list comprehension* para detectar solamente el número primo de la lista y después muestra el resultado por pantalla. **(0.5 puntos)** EG

[ ]:
```
# Respuesta
```

d) Usa los casos de prueba detallados en la tabla siguiente para comprobar que tu código funciona correctamente. NM **(0.25 puntos)**

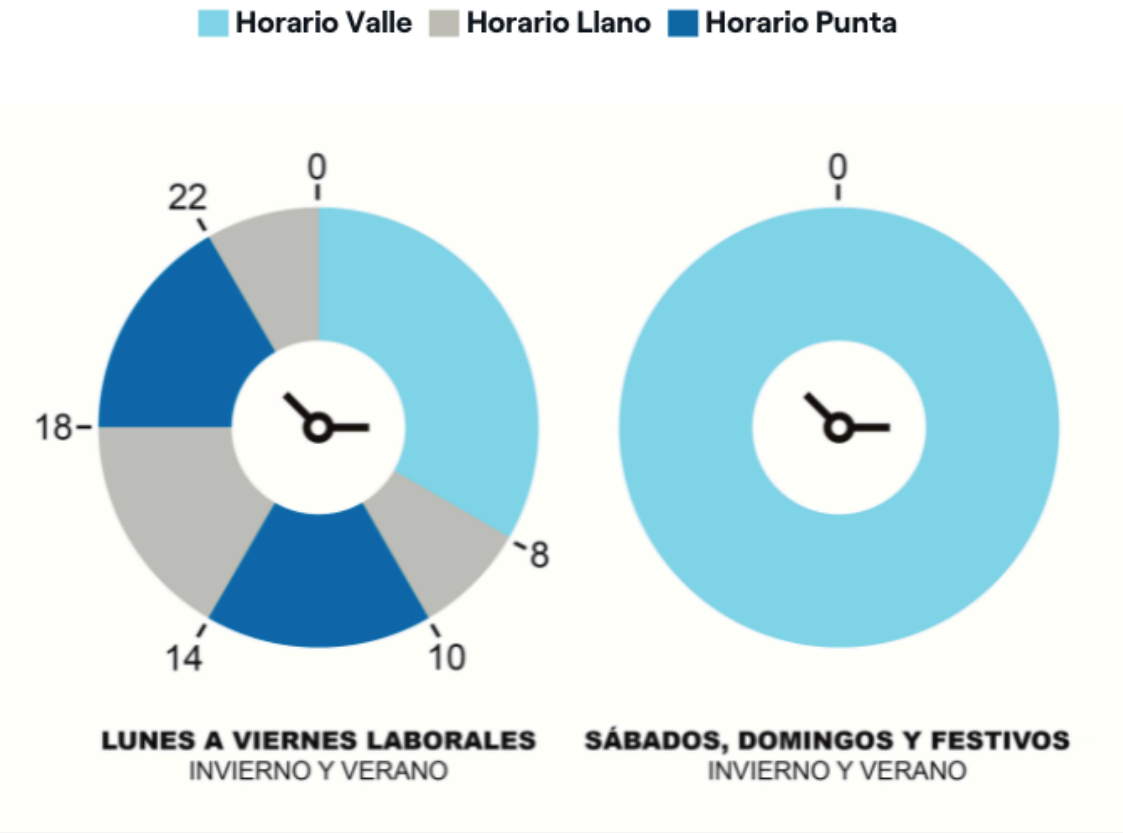
lista	a)	b)	c)
[‘hola’, False, 25, 41, True]	[‘hola’, False, 25, 41, True]	[‘hola’]	[41]
[“oso”, “perro”, “gato” , “conejo”, True]	[“oso”, “perro”, “gato” , “conejo”, True]	[‘oso’, ‘perro’, ‘gato’, ‘conejo’]	[]
[44, 7, 26, 14, 100]	[44, 7, 26, 14, 100]	[]	[7]

[ ]:
```
# Respuesta
```

1.2.3 Ejercicio 3

En Junio de 2021 se introdujo el cambio de tarifas de la luz. En algunas tarifas el consumo de electricidad puede tener diferentes precios a lo largo del día.

Las tres franjas horarias que se han establecido son las siguientes: horas valle, horas llanas y horas punta.



Estos horarios de la luz pueden afectar a lo que acabas pagando en la factura. Por este motivo, queremos crear un código que, dada una hora y día de la semana, nos indique a que franja corresponde. **(2.5 puntos)**

El código tiene que tener los siguientes elementos: - Se debe crea dos variables iniciales: una para contener el dia de la semana (por ejemplo, "Lunes") y la otra la hora (de 0-23h). NM **(0.5 puntos)** - Se debe crear un diccionario con los 3 tramos y sus respectivas horas (por ejemplo, valle : 0-7) para los días entre semana (no hace falta crear un diccionario para el fin de semana ya que siempre estamos en el tramo valle). **Pista**: Para crear el diccionario tenemos los nombres del tramo y sus respectivas horas. Recordad que la key del diccionario debe ser un valor único, no se puede repetir. EG **(1 punto)** - Se debe crear un código que, dado el día y la hora especificados en el apartado a), muestre por pantalla un mensaje explicando a que tramo pertenece (por ejemplo, "Estás en el tramo punta, vigila tu consumo de electricidad!"). NM **(1 punto)**

[ ]:
```
# Respuesta
```

1.2.4 Ejercicio 4

Basándonos en el código anterior, queremos mejorarlo para que pueda ser utilizado fácilmente por otros usuarios.

a) Para lograrlo, vamos a crear una función (por ejemplo, `calculadora_tramos_luz`) con los siguientes elementos:

- La función debe tener dos inputs: el día y la hora que queremos evaluar. NM **(0.5 puntos)**
- Tendremos que vigilar que los elementos introducidos (dia y hora) sean correctos. En el caso de haber algún error, se debe mostrar un mensaje de error por pantalla (por ejemplo, "[ERROR] El rango de horas es de 0 a 23h"). NM **(1 punto)**
- La función tiene que mostrar por pantalla el mensaje del tramo. En el caso de haber algún error, se tiene que mostrar por pantalla un mensaje explicando donde está el error. NM **(1 punto)**

(2.5 puntos)

[ ]:
```
# Respuesta
```

b) Usa los casos de prueba detallados en la tabla siguiente para comprobar que la función que has creado funciona correctamente. Para mejorar la usabilidad, el día y la hora se pedirán al usuario por pantalla, que introducirá los datos de la tabla manualmente y comprobará que el resultado obtenido es el esperado. EG **(0.5 puntos)**

Nota: En este enlace encontraréis información sobre la función *input*, que permite al usuario introducir datos mediante el teclado.

día	hora	Mensaje
Lunes	17	Estás en el tramo llano, ten un consumo de luz moderado
Domingo	13	Estás en el tramo valle, aprovecha para hacer lavadoras!
Mircoles	5	[ERROR] El formato del dia de la semana introducido no es válido
Viernes	25	[ERROR] El rango de horas es de 0 a 23h

[ ]:
```
# Respuesta
```

1.2.5 Ejercicio 5

a) Como ya somos unos expertos de los tramos de la luz, queremos crear un fichero en formato texto (.txt) con la información de las horas (de 0 a 23h) y su correspondiente tramo (para los días laborables). NM **(0.5 puntos)**

Este fichero debe tener una línea para cada hora, y, en cada línea, se tiene que especificar la **hora** y su **tramo**. Un ejemplo de las primeras filas del fichero es el siguiente:

La hora 0 es el tramo: valle

La hora 1 es el tramo: valle

La hora 2 es el tramo: valle

La hora 3 es el tramo: valle

Nota: Para hacer este ejercicio podéis utilizar el diccionario de tramos y horas creado en el ejercicio 3.

[ ]:
```
# Respuesta
```

b) Una vez creado, lee el contenido del fichero y muéstralo por pantalla. NM **(0.25 puntos)**

[ ]:
```
# Respuesta
```

c) Abre el fichero, recórrelo fila a fila, y obtén el numero total de horas, y cuántas de ellas son del tramo punta. NM **(0.5 puntos)**

[ ]:
```
# Respuesta
```

1.2.6 Ejercicio Opcional

Con los ejercicios que hemos ido trabajando hemos visto que, cuando ocurre algún error en el código, Python detiene la ejecución y devuelve una excepción, que nos indica que ha occurrido un error en el programa.

Supongamos que tenemos una función que calcula la raiz del número que se le introduce como input. En este caso, si intentamos calcular la raiz de un número negativo, salta un error:

`ValueError: math domain error`

Con este mensaje, Python nos avisa que hay un error del tipo *ValueError* en la tercera línea, y termina la ejecución. Este error nos indica que no se puede obtener la raiz real de un número negativo.

[ ]:
```
def raiz(n):
    # Importamos la libreria para poder utilizar la función raíz (sqrt)
    import math
    # Calculamos la raíz cuadrada de n
    m = math.sqrt(n)
    # Mostramos el resultado
    print(m)
```

[ ]:
```
raiz(-5)
```

Saber que ha ocurrido una excepción o problema en el código es útil, pero normalmente no queremos que el programa se pare, sinó que queremos capturar la excepción y seguir adelante. Para hacer esto podemos usar la sentencia `try-except`.

Dado un codigo, vamos a probar (try) de ejecutarlo y, en caso de encontrar un error, lo vamos a capturar y hacer algo al respecto (except) en vez de detener el programa.

En el ejemplo anterior podríamos hacer lo siguiente:

[ ]:
```
def raiz_advanced(n):
    # Importamos la libreria para poder utilizar la función raíz (sqrt)
    import math
    # Probamos (try) de hacer este trozo de código
    try:
        # Calculamos la raíz cuadrada de n
        m = math.sqrt(n)
        # Mostramos el resultado
        print(m)
    # Si nos encontramos un error ValueError, lo capturamos y mostramos un␣
↪mensaje de error
    except ValueError:
        print("El número introducido no es válido")
```

[ ]:
```
# Comprobamos que el try-except funciona correctamente
raiz_advanced(-5)
```

Como podéis ver, en este caso, el error `ValueError` es capturado y, en vez de interrumpirse el programa, se lanza un mensaje de error hecho por nosotros describiendo el problema.

Dada una función que divide los dos números que se especifican como input:

[ ]:
```
def test_fails(a,b):
    c = a / b
    print(c)
```

Utiliza la sentencia try-except para evitar que se interrumpa el código siguiente: EI

[ ]:
```
test_fails(5,0)
```

[ ]:
```
# Respuesta
```