Sesión 2 - undamentos de programación.

Contenidos:

Flujo del programa.

Llamamos flujo al recorrido del programa por las diferentes instrucciones. Hasta ahora solo hemos realizado programas de flujo secuencial, es decir, programas que ejecutan sus instrucciones de una en una. En el tema de concurrencia veremos que pueden existir varios flujos paralelos que ejecutan el código en paralelo.

Controlarlo si una instrucción de ejecuta o no y cuantas veces es básico para realizar cualquier algoritmo. Según el teorema de la programación estructurada todos los algoritmos pueden realizarse solo con estructuras de selección e iteración en un flujo secuencial.

Estructuras de selección.

Las estructuras de selección permiten tomar decisiones. Técnicamente se puede decir que modifican el flujo del programa en función de una variable de entrada.

En Python tenemos dos estructuras de selección, `if` y `match`. Esta última es bastante reciente y requiere de conocimientos que ahora mismo no se han tratado. Se verá con detalle en el futuro.

La estructura `if` funciona como en cualquier otro lenguaje: ejecuta el bloque de código a continuación si la expresión se evalua como `True`. Su sintaxis básica es:

```
if expresion:
    # Bloque de código
```

> En Python no existen los bloques de llaves `{}` sino que se utiliza indentación (normalmente 4 espacios) para crear bloques en estructuras, funciones, clases, etc.

> Si queremos dejar un bloque de código vacío utilizaremos la palabra reservada `pass`. No hace nada, simplemente evita el error de sintaxis.

Podemos amplíar la estructura básica añadiendo verificaciones adicionales si no se cumple la primera (`elif`) y un bloque de código si no se cumple ninguna (`else`). La sintaxis quedaría:

```
if expresion_1:
    # Código si expresion_1 es True
elif expresion_2:
    # Código si expresion_1 es False y expresion_2 es True
else:
    # Código si expresion_1 y expresin_2 son False
```

Veamos un ejemplo:

```
print('Descuento en entradas de espectáculo.')
edad = input('Introduce tu edad: ')

if edad >= 30:
    print('Entrada normal.')
elif edad >= 18:
    print('Carnet joven. 50% descuento.')
else:
    print('Menor de edad. Entrada gratis')
```

Ejercicio de clase

En la UPCT existe una beca de comedor para comer gratis todos los días en la cantina. Sus condiciones son:

- Tener más de un 8 de media en el expediente.
- Cumplir una o más de estas condiciones: renta familiar <20.000€ al año, ser huérfano o tener una discapacidad de más del 75%.
- Haber solicitado la beca MEC.

Elabora un formulario y determina si el usuario puede recibir la beca. Asegúrate de que la entrada del usuario tiene sentido.

Las estructuras `if` a veces la podemos encontrar comprimida en expresiones para seleccionar valores:

```
x = 5, y = 6
mayor = x if x > y else y
```

Encontraremos estas estructuras comprimidas en más situaciones.

Estructuras de iteración.

La iteración se define como la repetición de un proceso o conjunto de operaciones para lograr un resultado. El número de repeticiones dependerá de si se cumple una condición (o varias).

La iteración nos permite:

- Realizar la misma operación con distintos valores. Ej: calcular números primos.
- Realizar una misma operación a la salida de la anterior operación. Ej: método de newton.
- Aplicar un método general a una lista de casos particulares. Ej: extraer información de un flujo de datos.

Las estructuras de iteración que encontramos en Python son los bucles `while` y `for`.

> Se puede utilizar indistintamente bucle y estructura de iteración.

La estructura `while` funciona como un `if` que ejecuta el bloque de código en bucle mientras la condición sea cierta. La sintaxis básica es:

```
while condicion:
    # Bloque que se repite mientras condicion == True
```

La estructura básica se puede ampliar:

- Con una salida antes de tiempo con `break`:

  ```
  while condicion:
      # Código 1
      if otra_condicion:
          break  # Salimos del bucle sin ejecutar Código 2
      # Código 2
  ```

- Saltando a la siguiente iteración con `continue`:

  ```
  while condicion:
      # Código 1
      if otra_condicion:
          continue  # Volvemos a Código 1 sin pasar por 2.
      # Código 2
  ```

- Con un bloque de escape:

  ```
  while condicion:
      # Código que se ejecuta repetidamente.
  else:
      # Codigo de salida (una vez).
  ```

Los bucles `while` se suelen utilizar con la expresión constante `True` para iteraciones que no suelen acabar salvo excepciones u condiciones de salida. Por ejemplo:

```
while True:
    command = controller.wait_command() # Espera recibir instrucciones del
controlador.
    if command == Commands.exit:
        # Si la instrucción es exit, salimos del bucle y del programa
        break
    ... # Analiza otros comandos
# Fin del program
```

Ejercicio de clase

Repite el ejemplo anterior sustituyendo el controlador por un `input()` y que acabe cuando la instrucción valga `exit`.

En el bucle `for` iteramos sobre valores que genera un elemento llamado iterador los cuales se van asignando a una variable. La sintaxis básica es:

```
for variable in iterador:
    # variable cambia de valor en cada iteración
```

En Python la iteración se acaba cuando se solicita al iterador un nuevo valor y este no tiene más valores que dar. Algunos ejemplos son:

```
for i in [1, 2, 3, 4]:
    print(i)  # imprime 1, 2, 3, 4

for c in 'Hola':
    print(c, end='-')  # imprime H-o-l-a-
```

> La estructura de `[]` se llama lista y se tratará en la próxima sesión. Por ahora puedes usarla como un contenedor genérico.

Existen algunas funciones que funcionan como iteradores a las que llamamos **generadores**. El generador más conocido es `range()` que permite generar números enteros en un rango:

```
for i in range(10):
    print(i)  # Imprime de 0 a 9
```

```
for i in range(-5,6):
    print(i)  # Imprime de -5 a 5

for i in range(10, 0, -1):
    print(i)  # Imprime de 10 a 1
```

Los elementos que amplían la estructura básica de `while`, también pueden encontrarse en `for`:

```
for i in range(10):
    print(f'El número {i} al cuadrado es menor que 60.')
    if i**2 > 60:
        break
else:
    print(f'El número {i} es el primero que al cuadrado es mayor que 60.')
```

También se da el caso en el que se obtienen varios valores a la vez de un iterador:

```
for fruit, count in [('manzanas', 10), ('peras', 5), ('papayas', 1)]:
    print(f'Tengo {count} {fruit if count > 1 else fruit[:-1]}')
```

> Los paréntesis `()` definen una estructura inmutable llamada tupla y se utiliza para pasar conjuntos de valores. Lo veremos en detalle en la próxima sesión.

> Se llama desempaque (*unpacking*) al proceso de asignación múltiple. Junto con las estructuras se verá en detalle en la próxima sesión.

Los bucles se pueden anidar y combinar con cualquier otra estructura. Como ejemplo, este código calcula todos los 100 primeros primos:

```
count = 0
candidate = 1
while count < 100:
    is_prime = True
    for divisor in range(2, candidate//2):
        if candidate%divisor == 0:
            is_prime = False
            break
    if is_prime:
        print(f'El número {candidate} es el #{1+count:03d} primo.')
        count += 1
    candidate += 1
```

Ejercicio de clase

Utiliza el algoritmo babilónico para calcular la raíz cuadrada de un número introducido por el usuario con una precisión de 4 dígitos.

El algoritmo babilónico es un algoritmo por iteración que permite calcular raíces cuadradas intentando asemejar el área de un rectángulo a un cuadrado. Su expresión matemática es:

$$f_0(x)=x$$ $$f_n(x)=\frac{1}{2}(\frac{x}{f_{n-1}(x)}+f_{n-1}(x))$$ $$f_\infty(x) = \sqrt{x}$$

Ejercicio de clase

Vamos a mejorar el parser de la sesión anterior. Imaginemos que recibimos un texto como este de una estación meteorológica cada hora:

```
hora:14-temp:297.5-hum:78.4-rain:0
```

Siendo:

- `temp` → Temperatura K
- `hum` → Humedad en %Relativo
- `rain` → Precipitación mm/h

Calcula:

- Temperatura y humedad máxima y mínima y a qué hora se dieron.
- Precipitación acumulada durante el día.

Puedes utilizar los siguientes datos:

```
hora:0-temp:297.5-hum:78.4-rain:0
hora:1-temp:296.3-hum:79.2-rain:0
hora:2-temp:294.1-hum:78.1-rain:0
hora:3-temp:292.5-hum:78.0-rain:0
hora:4-temp:291.8-hum:78.6-rain:0
hora:5-temp:292.4-hum:80.4-rain:10
hora:6-temp:294.7-hum:82.4-rain:11
hora:7-temp:295.5-hum:88.4-rain:40
hora:8-temp:298.6-hum:92.4-rain:42
hora:9-temp:300.5-hum:96.4-rain:44
hora:10-temp:301.2-hum:95.6-rain:20
hora:11-temp:302.2-hum:95.7-rain:22
hora:12-temp:302.9-hum:95.4-rain:10
hora:13-temp:302.9-hum:95.4-rain:12
hora:14-temp:303.2-hum:92.2-rain:6
hora:15-temp:303.4-hum:90.1-rain:2
hora:16-temp:302.5-hum:88.0-rain:0
hora:17-temp:300.8-hum:88.6-rain:0
hora:18-temp:299.4-hum:88.4-rain:0
hora:19-temp:297.7-hum:89.4-rain:0
hora:20-temp:294.5-hum:87.4-rain:0
hora:21-temp:291.6-hum:89.4-rain:0
```

```
hora:22-temp:290.5-hum:90.4-rain:16
hora:23-temp:290.2-hum:93.6-rain:21
```

Ejercicios extra de iteración.

1. Realiza un programa que tras introducir un polinomio real de grado n encuentre la raíz más cercana a 0 mediante el método de Newton.
2. Realiza una función que calcule los primeros N decimales de PI usando la Serie de Leibniz.

Funciones

En Python una función es un bloque de código separado del bloque principal, el cual puede ser invocado en cualquier momento desde ésta u otra función.

Cuando una función es invocada (llamada) el flujo del programa se redirige hacia el código de la función. Las funciones intercambian información con el programa principal con mediante argumentos y la expresión de retorno.

Las funciones en Python, como el resto de elementos del lenguaje, son objetos. Las implicaciones se irán revelando en esta y futuras sesiones.

Definición de funciones.

La sintaxis básica para definir funciones es:

```
def nombre_funcion([arg1, ..., argN]):
    # Bloque de código
    [return out]
```

Los argumentos y el retorno son opcionales. Si no se indica retorno se devuelve `None`. Veamos algunos ejemplo:

```
# Ejemplo función normal.
def sum_of_squares(a, b):
    'Devuelve la suma de cuadrados (a+b)²'
    return a**2+b**2+2*a*b

# Función sin retorno
def print_with_promt(variable):
    'Añade >> previo a la impresión'
    print('>> ',variable)

# Función sin argumento ni retorno
def say_hi():
    'Dice hola'
    print('Hola')
```

> Existen una norma para comentar funciones apropiadamente PEP257 recogida en la documentación oficial. Sin embargo, lo más común para el desarrollo de código **que no va a ser una API pública** es realizar un **comentario simple** de qué hace la función y respetar el **type hinting** que veremos más adelante.

Podemos definir valores por defecto para las funciones:

```
def say_hi(idioma='es'):
    'Dice hola en varios idiomas'
    if idioma == 'es':
        print('Hola')
    elif idioma == 'en':
        print('Hello')
    elif idioma == 'fr':
        print('Baguette')
    else:
        print('No sé tu idioma')
```

Python soporta recursividad entre funciones:

```
def factorial(n):
    'Calcula el factorial de n.'
    if n <= 1:
        return n
    else:
        return n * factorial(n-1)
```

Y definición de funciones dentro de otras funciones:

```
def super_funcion():
    'Función con muchas responsabilidades.'
    def auxiliar_1():
        ...
    def auxiliar_2():
        ...
    def auxiliar_3():
        def ayudante_de_auxiliar():
            ...

    return auxiliar_1() + auxiliar_2() + auxiliar_3()
```

Argumentos y retornos múltiples.

Podemos definir funciones con un número no definido de argumentos de entrada. Veamos el siguiente ejemplo:

```
def guardar_datos(*datos):
    'Simula guardar datos'
    for dato in datos:
        print('Guardado:', dato)
guardar_datos(1, 2, 3)
```

Salida:

```
>> Guardado 1
>> Guardado 2
>> Guardado 3
```

En el ejemplo el argumento `datos` es un contenedor (`tuple`) donde se guardan todos los elementos que se pasan como argumento. Estos contenedores se generan cuando a un argumentos se le marca con el símbolo `*`.

Para devolver varios valores a la vez simplemente los separamos por comas:

```
def dame_doble_mitad(num):
    'Devuelve el doble y la mitad de un número'
    return num*2, num//2

doble, mitad = dame_doble_mitad(8)
```

Ámbito de las variables.

Las variables que creamos dentro de funciones o las que hacen refencia a los argumentos se elimina automáticamente al finalizar la función. Esto se debe a que son variables de **ámbito local**.

Llamamos **ámbito de la variable** a las partes del programa donde estas existen. Las funciones crean su propio ámbito donde las varibles que se definen dejan de existir al salir y no se pueden acceder desde el exterior.

Llamamos al ámbito del cuerpo principal de programa **ámbito global**. Si definimos una variable en el ámbito global será accesible desde cualquier parte, incluidas funciones.

Sin embargo, es posible intentar cambiar el valor de una variable en una función con el mismo nombre que una variable del ámbito global. Esto provoca un problema de interpretación (¿quieres crear una variable local o cambiar el valor de la global?) y hay que indicar si se hace referencia a la variable de ámbito global con la etiqueta `global`.

```
x = 10

def print_x():
    'Imprime la x global'
```

```
    print(x) # -> 10

def print_local_x():
    'Imprime la x local'
    x = 5
    print(x)

def print_set_x(n):
    'Cambia el valor de x global y lo imprime.'
    global x
    x = n
    print(x)
```

Variables estáticas.

Podemos crear variables estáticas (que guardan su valor entre llamadas) añadiendo un atributo nuevo a la función con el operador de acceso (`.`).

```
# Primera forma (accedemos desde fuera y creamos un atributo nuevo).
def counter():
    'Cuenta las veces que llamas a la función'
    counter.c += 1
    return counter.c
counter.c = 0  # Asignado desde fuera

# Segunda forma (comprobamos si contamos con el atributo desde dentro).
def counter():
    'Cuenta las veces que llamas a la función'
    if not hasattr(counter, 'c'):
        counter.c = 0
    counter.c += 1
    return counter.c
```

> La función `hasattr` permite saber si un objeto tiene un atributo concreto.

Ejercicio de clase

Modifica el ejercicio anterior del parser para organizarlo con funciones. Para ello crea una función genérica que extraiga cualquier dato de una cadena con el formato:

> `var1:dato-var2:dato-(...)-varN:dat`

Un ejemplo de la interfaz de la función es:

```
cadena = 'temp:29-time:23:45'
temp = extraer_dato(cadena, 'temp')
hora = extraer_dato(cadena, 'time')
```

Módulos

Si sales del intérprete de Python y vuelves a entrar, las definiciones que habías hecho (funciones y variables) se pierden. Por lo tanto, si quieres escribir un programa más o menos largo, es mejor que utilices un editor de texto para preparar la entrada para el intérprete y ejecutarlo con ese archivo como entrada. Esto se conoce como crear un script. A medida que tu programa crezca, quizás quieras separarlo en varios archivos para que el mantenimiento sea más sencillo. Quizás también quieras usar una función útil que has escrito en distintos programas sin copiar su definición en cada programa.

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlos en un script o en una instancia del intérprete. Este tipo de ficheros se llama **módulo**; las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal (la colección de variables a las que tienes acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Un módulo es un fichero conteniendo definiciones y declaraciones de Python. El nombre de archivo es el nombre del módulo con el sufijo `.py` agregado.

En el tema de diseño de aplicaciones veremos como instalar módulos en nuestro sistema y crear colecciones de módulos llamadas **paquetes**. Por ahora colocaremos los módulos que hagamos en la misma carpeta que nuestro script principal.

Para importar un módulo utilizamos la etiqueta `import`:

```
import math
```

Ahora podemos utilizar las funciones y variables que contiene accediendo mediante el operador de acceso (`.`).

```
print(math.sqrt(25))   # -> 5
```

Se pueden importar elementos concretos de un módulo mediante `from ... import`:

```
from math import pi, cos
print(cos(pi))   # -> -1

from time import asctime as timestamp # también se permite poner motes
print(timestamp())   # Imprime la fecha
```

También se permite importar todos los elementos de un módulo, pero no es recomendable.

```
from random import *

random_data = gauss(mu=0.0, sigma=1.0)
```

Python cuenta con una gran colección de módulos presinstalados llamada *The Python Standard Library*. Durante lo que resta del curso se hará uso intensivo de ella durante el curso.

Ejercicio de clase

Crea un módulo con las funciones del parser de los ejercicios anteriores y llama a la función de análisis desde otro script.

Manejo de errores.

Cuando se generan errores en tiempo de ejecución la máquina virtual nos manda una señal de que se ha producido un error y nos da la oportunidad de reconducir el flujo del programa en vez de terminar con él. Los errores en tiempo de ejecución se les conoce en Python como **excepciones**.

A la captura de excepciones y a las acciones posteriores se les conoce como **manejo de errores** (*error handling*).

Para el manejo de errores se utiliza la estructura `try-except-else`. La sintaxis es:

```
try:
    # Bloque de código donde pueden haber errores
except ExcepcionX:
    # Bloque de código si se da la Excepción X.
else:
    # Código que se ejecuta si no se han producido errores
```

Veamos un ejemplo:

```
print('Programa divisor')
dividendo = input('Introduce dividendo: ')
divisor = input('Indroduce divisor: ')
try:
    resultado = int(dividendo) / int(divisor)
except ValueError:
    print('No has introducido correctamente los números')
except ZeroDivisionError:
    print('No puedes dividir por cero')
else:
    print(f'{dividendo}/{divisor} = {resultado:.2e}')
```

`ValueError` y `ZeroDivisionError` son excepciones predefinidas. Python tiene incorporadas de serie y cada módulo suele contener las suyas propias para dar más información sobre el tipo de error. Algunas de las excepciones estándar más comunes son:

- **TypeError**: Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

- **ValueError**: Ocurre cuando el valor que introduces en una función no es válido (por ejemplo: al intentar convertir a un entero un texto sin formato numérico).
- **ZeroDivisionError** : Ocurre cuando se itenta dividir por cero.
- **IndexError** : Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.
- **KeyError** : Ocurre cuando se intenta acceder a un diccionario con una clave que no existe.
- **FileNotFoundError**: Ocurre cuando se intenta acceder a un fichero que no existe en la ruta indicada.
- **ImportError**: Ocurre cuando falla la importación de un módulo.

Existe una excepción genérica, `Exception`, de la que derivan todas las demás y que puede servir para capturar cualquier excepción.

```
try:
    # código
except Exception:
    # código si se da cualquier error
```

Podemos añadir una cláusula más a la estructura de manejo de errores. La etiqueta `finally` que se encarga de operaciones que se realizarán se den o no errores y si estos son gestionados o no. Si se dan errores, tras ejecutar el código de `finally`, estos serán relanzados.

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

Otra función interesante del manejo de errores es la capacidad de capturar errores en una variable:

```
x, y = 1, 0
try:
    x/y
except ZeroDivisionError as zerr:
    print('Se ha producido un error:', zerr)
```

> Aquí la variable `zerr` no es un `str`, lo que vemos es una representación del objeto (su mensaje en este caso).

Esto será más útil cuando definamos nuestros propios errores.

Lanzar errores.

Se pueden generar errores a voluntad con la instrucción `raise`.

```
raise Exception('Aquí un mensaje de error')
```

Podemos lanzar cualquier excepción con el mensaje que queramos. También se permite relanzar errores:

```
x, y = 1, 0
try:
    x/y
except ZeroDivisionError as zerr:
    raise Exception(f'Se ha producido un error: {zerr}')
```

Asserts

Existe una excepción que se lanza cuando no se cumple una condición definida por el usuario. Esto es útil cuando quieres que se cumplan ciertas condiciones en tu programa. Para realizar estas comprobaciones se utiliza la declaración `assert`. Su sintaxis es:

```
assert condition, "Mensaje de erorr opcional"
```

Si `condition` no se cumple se lanzará un `AssertionError`. Un ejemplo:

```
radio = float(input('Introduce radio de una esfera: '))
assert radio > 0, 'El radio de una esfera debe ser mayor que 0'
```

Combinándolo con manejo de errores y errores definidos por el usuario (se verá más adelante) se convierte en una herramienta muy potente para realizar múltiples comprobaciones:

```
def registrar_usuario(nombre, dni, edad):
    'Registra un usario'
    try:
        assert type(nombre) is str, 'El nombre debe ser texto'
        assert check_dni(dni), 'El DNI no es correcto'
        assert edad < 18, 'Debe ser mayor de edad para registrarse.'
    except AssertionError as err:
        raise UserRegistrationError(str(err))  # me he inventado el error
    # Código de registro
```