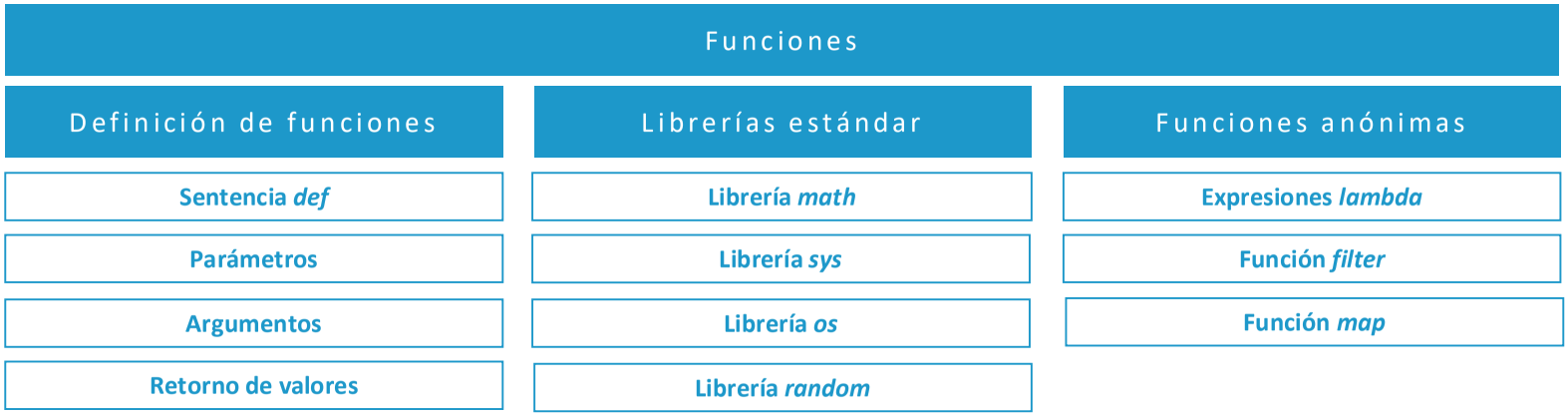
Introducción a la programación y al análisis de datos con Python

Funciones

Índice

Esquema



Ideas clave

4.1. Introducción y objetivos

Las funciones tienen dos principales ventajas en la programación. En primer lugar, una función puede implementar una funcionalidad que vamos a utilizar más veces dentro de un programa. En segundo lugar, las funciones nos permiten organizar el código de manera que podamos agrupar bloques de código en funcionalidades concretas. En este tema vamos a explicar cómo implementar las funciones en Python y cómo podemos usarlas. Los objetivos son los siguientes:

- Aprender cómo definir las funciones.
- Comprender cómo se usan los parámetros y los argumentos en las funciones.
- Conocer cómo se devuelven resultados con las funciones.
- Conocer algunas de las funciones básicas incluidas en Python.
- Aprender cómo se definen las funciones anónimas en Python.

4.2. Definición de funciones

Las funciones nos permiten encapsular un bloque de instrucciones para que podamos utilizarlo varias veces dentro de nuestros programas. Para hacer esto, es necesario identificar ese bloque de instrucciones usando la sintaxis de funciones que incluye Python. Las funciones se definen con la sentencia `def`. Esta palabra clave indica a Python que vamos a crear un objeto ejecutable que podrá ser llamado más adelante durante la ejecución del programa. El formato para definir una función en Python es:

```
def NOMBRE_FUNCION(ARG1, ARG2,...):
    sentencia_1
    sentencia_2
    ...
    return OBJETO_DEVOLVER
```

Como podemos observar, la primera línea viene definida por la palabra reservada `def`, seguida del nombre que queramos darle a la función. Este nombre será el identificador que nos permitirá llamar a la función más adelante. A continuación, indicaremos la lista de parámetros que tendrá la función, separados por comas y entre paréntesis. Estos parámetros serán valores u objetos que se necesitarán para ejecutar la función. En el caso de que nuestra función no necesitase parámetros, dejaremos los paréntesis vacíos. Al final de la declaración de la función usaremos dos puntos (:) para comenzar a escribir las sentencias que se ejecutarán. El bloque de sentencias debe tener una sangría de 4 espacios con respecto a la definición de la función para que el Python entienda que son instrucciones que pertenecen a la función. En el caso de que la función tenga que devolver un valor, usaremos la palabra reservada `return` junto con el identificador o la expresión que devolverá el valor.

Vamos a ver la definición de una función a través de un ejemplo. Imaginemos que queremos crear una función que nos calcule el área de un triángulo. Para poder calcular el área del triángulo, la función necesita dos datos: su base y su altura. Estos dos datos serán los parámetros de la función. Por último, devolveremos el resultado del cálculo usando la sentencia `return`. La función para calcular el área del triángulo quedaría como se muestra a continuación:

```
def area_triangulo(base, altura):
    area = (base * altura) / 2
    return area
```

Para ejecutar la función solo debemos usar el nombre de la función e insertar los valores de la base y la altura:

```
area_triangulo(6, 5) # Devolverá 15.0
```

Esta es la forma general de definir una función en Python. Sin embargo, existen muchas opciones que nos permiten definir los parámetros de las funciones.

4.3. Parámetros

En la mayoría de las ocasiones, necesitamos introducir información a las funciones para que puedan realizar la acción que deseamos. Por este motivo, podemos declarar unos parámetros que la función usará para leer esa información. Hay que saber diferenciar entre los **parámetros,** que son los valores que definimos en una función, y los **argumentos,** que son los valores que introducimos a la función en el momento de la ejecución.

En el ejemplo anterior, cuando hemos declarado la función `area_triangulo`, los elementos `base` y `altura` eran los parámetros. Más adelante, cuando hemos indicado que la base era 6 y la altura 5, esos valores son los argumentos de la función.

A la hora de llamar a una función existen dos formas para introducir los argumentos de la función:

- **Argumentos por posición:** los argumentos se envían en el mismo orden en el que se han definido los parámetros. Esta es la forma que hemos escogido para llamar a la función `area_triangulo` en el ejemplo anterior.
- **Argumentos por nombre:** los argumentos se envían utilizando los nombres de los parámetros que se han asignado en la función. Para ello, usaremos el nombre del parámetro, seguido del símbolo igual (=) y del argumento. En el ejemplo anterior sería de la siguiente forma:

```
area_triangulo(base=10, altura=4) # Devolverá 20.0
```

Cuando una función tiene definido unos parámetros, es obligatorio que, a la hora de llamarlo, la función reciba el mismo número de argumentos. En caso de que no recibiese alguno de esos argumentos, Python devolvería un error de tipo. Sin embargo, en el momento de declarar una función, podemos asignar un valor por

defecto a cada parámetro. Este valor por defecto se usará solo si no se ha indicado un argumento en el parámetro correspondiente.

Para asignar un valor por defecto a un parámetro, definiremos el nombre del parámetro, seguido por el símbolo = y el valor por defecto que le queramos dar. Por ejemplo, vamos a asignar un valor por defecto para la `base` y la `altura` en la función `area_triangulo`.

```
def area_triangulo(base=10, altura=10):
    area = (base * altura) / 2
    return area
```

En este caso, si llamásemos a la función `area_triangulo` sin ningún argumento, usaría los valores por defecto que hemos indicado para calcular el área.

```
area_triangulo() # Devolverá 50.0
```

También podemos indicar únicamente uno de los parámetros. En el caso de que lo hagamos por posición, reconocerá que ese argumento es el del parámetro base; en cambio, si utilizamos los argumentos por nombre, podemos aplicarlo a cualquiera de los dos parámetros.

Muchas de las funciones que existen en Python tienen parámetros con valores por defecto. Por este motivo, es importante revisar la documentación de una función para saber cómo van a ser los argumentos de la función antes de usarla.

4.4. Parámetros indeterminados

En algunas ocasiones, puede que necesitamos definir una función que necesite un número variable de argumentos. Para estos casos Python nos permite usar los parámetros indeterminados en las funciones. Estos parámetros nos permiten incluir tantos argumentos como queramos en el momento de la ejecución de la función.

Como pasaba con los parámetros normales, existen dos maneras de asignar los argumentos:

- **Argumentos por posición:** se deben definir los parámetros como una lista dinámica. Para ello, a la hora de definir el parámetro, se incluirá un asterisco (*) antes del nombre del parámetro. Los parámetros indeterminados se recibirán por posición. A estos parámetros se les puede pasar cualquier tipo de dato en cada función. Un ejemplo de su uso sería el siguiente:

```
def imprime_numeros(*args):
    for numero in args:
        print(numero)

imprime_numeros(1, 7, 89, 46, 9394)
```

- **Argumentos por nombre:** para recibir varios argumentos por nombre, sin saber la cantidad, es necesario definir los parámetros como un diccionario dinámico. Para ello se usa dos asteriscos (**) antes del nombre del parámetro. Un ejemplo de su uso sería el siguiente:

```
def imprime_valores(**args):
    for argumento in args:
        print(argumento, '=>', args[argumento])

imprime_valores(arg1='Hola', arg2=[2,3,4], arg3=876.98)
```

A la hora de declarar una función, podemos combinar las dos formas de declarar parámetros normales con las formas que hemos visto para declarar parámetros indeterminados.

4.5. Retorno de valores

Uno de los objetivos más comunes en las funciones es que realicen una operación y devuelvan el resultado de la misma. Para devolver uno o varios valores se utiliza la sentencia `return`. En el siguiente ejemplo tenemos una función que devuelve la potencia entre dos números.

```
def potencia(base, exponente):
    return base ** exponente
```

Como vemos, la única instrucción que tenemos dentro de la función es una sentencia `return` que devuelve el resultado de la operación. Hay que tener en cuenta que la instrucción `return` debe ser la última instrucción para ejecutarse, ya que en ese momento Python saldrá de la función.

En Python, las funciones pueden devolver más de un valor a la vez. Para hacer esto, solo tenemos que separar por comas todos los valores que queramos devolver después de la sentencia `return`. En el siguiente ejemplo, la función devuelve tres objetos de diferentes tipos:

```
def ejemplo():
    return "Hola", 3546, [3,90]
```

Cuando devolvemos más de un valor en una función, lo que obtenemos es una tupla con todos los valores. Por este motivo, si queremos que cada uno de los resultados esté en una variable distinta, es necesario hacer un desempaquetado de la tupla. En el ejemplo anterior sería así:

```
var1, var2, var3 = ejemplo() # Nos dará var1 = “Hola”, var2 = 3546, var3 = [3, 90]
```

4.6. Documentar funciones

Cuando estamos desarrollando *software,* es muy importante documentar bien las secciones de código que vamos implementando. Esto nos permitirá recordar qué hace el código que implementamos o muestra a otros desarrolladores lo que queremos hacer. La documentación cobra mayor importancia cuando se trata de funciones. Las funciones encapsulan un comportamiento dentro de un identificador

y otros usuarios no deberían acceder al código de una función para saber qué es lo que se quiere hacer.

Para poder documentar las funciones se utilizan los `docstring`. En Python todos los objetos cuentan con una variable por defecto llamada `doc`, y nos permite acceder a la documentación de dicho objeto. Para documentar una función usando los `docstring`, únicamente tenemos que incluir un comentario inmediatamente después de la cabecera de la función. A continuación, se muestra cómo podríamos documentar la función potencia que hemos definido antes:

```
def potencia(base, exponente):
    """
    Función que calcula la potencia de dos números.

    Argumentos:
    base -- base de la operación.
    exponente -- exponente de la operación.
    """
    return base ** exponente
```

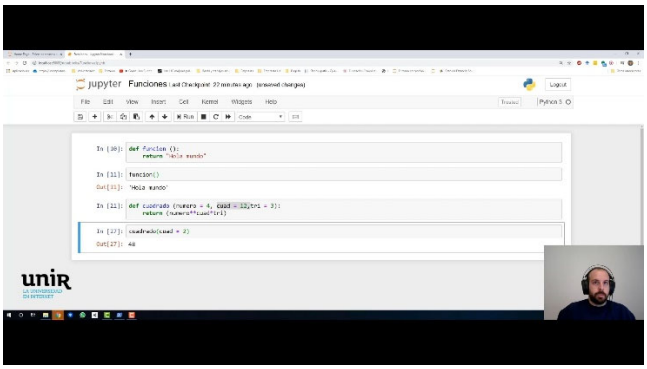
Al documentar de esta forma las funciones, podemos usar la sentencia `help()` con el nombre de la función para saber la documentación que tiene. En el ejemplo anterior se vería del siguiente modo:

```
help(potencia)
```

```
Help on function potencia in module __main__:

potencia(base, exponente)
    Función que calcula la potencia de dos números.

    Argumentos:
    base -- base de la operación.
    exponente -- exponente de la operación.
```

Figura 1. Sentencia `help()`.

En la guía de estilos oficial de Python (PEP8), hay varias reglas y consejos para documentar correctamente nuestro código usando los `docstring`.



Vídeo 1. Funciones.

Accede al vídeo a través del aula virtual

4.7. Funciones incluidas en Python

Python cuenta con bastantes módulos que incluyen funciones muy útiles para desarrollar nuestros programas. A continuación, veremos los módulos y las funciones más importantes.

Módulo `math`

El módulo `math` es un módulo matemático que incluye numerosas funciones matemáticas que nos pueden ser útiles en el desarrollo de nuestros proyectos. Para utilizar estas funciones es necesario importar el módulo `math` al principio de nuestro código usando la sentencia `import`:

```
import math
```

A continuación, enumeraremos las funciones más útiles agrupadas por tipos.

Funciones aritméticas

Estas funciones nos permiten realizar varias operaciones aritméticas como calcular los valores superiores o inferiores de un número, cálculos factoriales o el máximo común divisor. Las funciones más importantes son:

- **`fabs(x)`:** esta función devuelve el valor absoluto del valor x.

```
math.fabs(-1234) # Devolverá 1234
```

- **`gcd(x,y)`:** esta función devuelve el máximo común divisor de dos valores x e y.

```
math.gcd(34, 82) # Devolverá 2
```

- **`floor(x)`:** esta función devuelve el valor entero más grande que sea menor o igual a x.

```
math.floor(245.89) # Devolverá 245
```

- **`ceil(x)`:** esta función devuelve el valor entero más pequeño que sea mayor o igual a x.

```
math.ceil(245.89) # Devolverá 246
```

- **`factorial(x)`:** esta función calcula el factorial del número x.

```
math.factorial(5) # Devolverá 120
```

- **`trunc(x)`:** esta función devuelve la parte entera del número x.

```
math.trunc(5.7836) # Devolverá 5
```

Funciones trigonométricas

Estas funciones nos permiten hacer cálculos trigonométricos que relacionan los lados de los triángulos con sus ángulos. En todas estas funciones debemos tener en cuenta que se trabaja con los ángulos en radianes. Las funciones más importantes son:

- **`sin(x)`:** devuelve el valor del seno del ángulo x en radianes.

```
math.sin(math.pi/4) # Devolverá 0.7071067811865476
```

- **`cos(x)`:** devuelve el valor del coseno del ángulo x en radianes.

```
math.cos(math.pi) # Devolverá -1.0
```

- **`tan(x)`:** esta función devuelve el valor de la tangente del ángulo x en radianes.

```
math.tan(math.pi/2) # Devolverá 1.633123935319537e+16
```

- **`asin(x)`:** esta función calcula el valor del ángulo para que su seno sea x.

```
math.asin(1) # Devolverá 1.5707963267948966
```

- **`acos(x)`:** esta función calcula el valor del ángulo para que su coseno sea x.

```
math.acos(1) # Devolverá 0.0
```

- **`atan(x)`:** esta función calcula el valor del ángulo para que su tangente sea x.

```
math.atan(1) # Devolverá 0.7853981633974483
```

- **`hypot(x, y)`:** esta función calcula la longitud de la hipotenusa de un triángulo rectángulo a partir de los valores de los dos catetos (x, y).

```
math.hypot(10, 7) # Devolverá 12.206555615733702
```

Funciones exponenciales y logarítmicas

Este conjunto de funciones nos permite hacer cálculos sencillos de valores logarítmicos o exponenciales. Las funciones más importantes son:

- **`log(x, [base])`:** esta función permite calcular el logaritmo de x. Se puede especificar la base del algoritmo, aunque este argumento no es obligatorio y por defecto se calcula sobre base e.

```
math.log(148.41315910257657)     # Devolverá 5.0
math.log(148.41315910257657, 2)  # Devolverá 7.213475204444817
math.log(148.41315910257657, 10) # Devolverá 2.171472409516258
```

- **`log2(x)`:** esta función permite calcular el logaritmo de x con base 2. Devuelve un resultado más preciso que usando la función `log(x, 2)`.

```
math.log2(148.41315910257657)  # Devolverá 7.2134752044448165
```

- **`log10(x):`** esta función permite calcular el logaritmo de `x` con base 10. Devuelve un resultado más preciso que usando la función `log(x, 10)`.

```
math.log10(148.41315910257657)  # Devolverá 2.171472409516259
```

- **`pow(x, y):`** esta función permite calcular el valor de `x` elevado a la potencia `y`.

```
math.pow(2, 3)  # Devolverá 8
```

- **`sqrt(x):`** esta función calcula la raíz cuadrada del valor `x`.

```
math.sqrt(256)  # Devolverá 16
```

Además, el módulo `math` cuenta con dos valores constantes que nos pueden ser de utilidad:

- **`pi:`** contiene el valor del número pi.

```
math.pi  # Devolverá 3.141592653589793
```

- **`e:`** contiene el valor del número e.

```
math.e  # Devolverá 2.718281828459045
```

Módulo `sys`

Este módulo proporciona variables y métodos relacionados directamente con el intérprete de Python. En primer lugar, veremos algunas de las variables más destacadas de este módulo:

- **`argv:`** devuelve una lista con todos los argumentos que se han pasado por línea de comandos al ejecutar el *script*. Por ejemplo, si ejecutásemos la siguiente instrucción en nuestra consola de comandos:

```
python test.py Hola 25
```

Podríamos obtener los argumentos de la siguiente manera:

```
sys.argv  # Devolverá ['test.py', 'Hola', 25]
```

- **executable:** devuelve la ruta absoluta del fichero ejecutable del intérprete de Python.

```
sys.executable # Devolverá '/usr/local/Python/bin/python3.7'
```

- **version:** devuelve la versión de Python que se está ejecutando.

```
sys.version # Devolverá '3.7.8'
```

- **platform:** devuelve la plataforma sobre la que se está ejecutando Python.

```
sys.platform # Devolverá 'darwin
```

Algunos de los métodos más destacados del módulo `sys` son los siguientes:

- **exit():** termina la ejecución del intérprete de Python.

```
sys.exit() # Cerrará el intérprete de Python
```

- **getdefaultenconding():** devuelve el sistema de codificación por defecto del sistema.

```
sys.getdefaultencoding() # Devolverá 'utf-8'
```

Módulo os

Este módulo proporciona acceso a variables y funciones que interactúan directamente con el sistema operativo. Este módulo está incluido siempre en las distribuciones de Python. A continuación, veremos los elementos más importantes.

Métodos para archivos y directorios

Estos métodos nos permiten trabajar con las funcionalidades del sistema operativo para crear, eliminar o modificar archivos y directorios. Los métodos más destacados son:

- **getcwd():** devuelve la ruta del directorio en el que nos encontramos.

```
os.getcwd() # Devolverá /User/UNIR/Python (por ejemplo)
```

- **mkdir(path):** crea un nuevo directorio en la ruta que se ha especificado en el argumento.

```
os.mkdir('/NuevaCarpeta') # Creará una carpeta en la dirección /NuevaCarpeta
```

- **rmdir(path):** elimina el directorio de la ruta que se ha especificado en el argumento.

```
os.rmdir('/NuevaCarpeta') # Eliminará la carpeta /NuevaCarpeta
```

- **remove(path):** elimina el fichero de la ruta que se ha especificado en el argumento.

```
os.remove('/NuevaCarpeta/fichero.txt') # Eliminará el archivo /NuevaCarpeta/fichero.txt
```

- **rename(name1, name2):** renombra el fichero con nombre name1 y los sustituye por name2.

```
os.rename('/NuevaCarpeta/fichero.txt', '/NuevaCarpeta/archivo.txt') # Modificará el nombre de fichero.txt por archivo.txt
```

El módulo `os` incluye otro módulo llamado `path` que nos permite acceder a métodos asociados a los nombres de los ficheros y sus rutas. Para ello tenemos que importar el módulo `os.path`.

```
import os.path
```

Los métodos más importantes son:

- **abspath(ruta):** devuelve la ruta absoluta de un fichero.

```
os.path.abspath('./fichero.txt') # Devolverá /NuevaCarpeta/fichero.txt
```

- **basename(ruta):** devuelve el último componente de la ruta que se pasa por parámetro.

```
os.path.basename('/NuevaCarpeta/fichero.txt') # Devolverá fichero.txt
```

- **exists(ruta):** comprueba si un directorio existe en la ruta especificada.

```
os.path.exists('/NuevaCarpeta') # Devolverá True si existe
```

- **isfile(ruta):** comprueba si la ruta especificada corresponde a un fichero.

```
os.path.isfile('/NuevaCarpeta/fichero.txt') # Devolverá True
```

- **isdir(ruta):** comprueba si la ruta especificada corresponde a un directorio.

```
os.path.isdir('/NuevaCarpeta') # Devolverá True
```

Módulo `random`

Este módulo nos proporciona métodos para obtener valores aleatorios. Entre los métodos que destacamos se encuentran los siguientes:

- **randint(x, y):** devuelve un número aleatorio entre x e y.

```
random.randint(1,10) # Devolverá, por ejemplo, 7
```

- **choice(secuencia):** devuelve un dato aleatorio de los datos de la secuencia.

```
random.choice(['Hola', 3, [2, 3], True]) # Devolverá, por ejemplo,
True
```

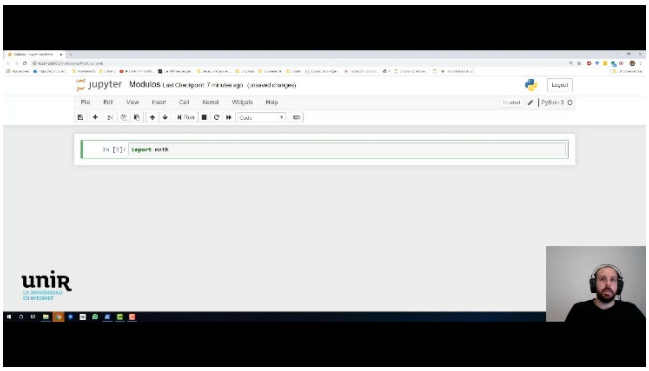
- **shuffle(secuencia):** permuta los elementos de una secuencia de forma aleatoria.

```
lista = ['Hola', 3, [2, 3], True]
random.shuffle(lista)
lista # Mostrará, por ejemplo, [3, 'Hola', True, [2, 3]]
```

- **sample(secuencia, n):** devuelve n elementos aleatorios de una secuencia.

```
lista = ['Hola', 3, [2, 3], True]
random.sample(lista, 2) # Devolverá, por ejemplo, [[2, 3], True]
```

Además de todos estos módulos, existen muchos más que nos permiten hacer conexiones a Internet, cortar la longitud de un texto, etc. Todos estos módulos se pueden consultar desde la documentación oficial de Python.



Vídeo 2. Funciones incluidas en Python.

Accede al vídeo a través del aula virtual

4.8. Funciones anónimas

Las funciones anónimas son funciones a las cuales no les vamos a asignar un identificador para ejecutarlas. Es decir, no usaremos la cabecera `def NOMBRE_FUNCION` para definirlas. El objetivo de estas funciones es el mismo que el de las funciones normales, con la salvedad de que en estas funciones no podemos incluir un bloque de código, solo una única expresión.

Para implementar las funciones anónimas en Python, usaremos las expresiones `lambda`. Estas expresiones son muy potentes, aunque algo confusas, sobre todo cuando se empiezan a utilizar.

Para explicar una función `lambda` usaremos un ejemplo. Imaginemos que tenemos una función normal que recibe un número y devolvemos su valor al cuadrado. Esta función se definiría del siguiente modo:

```
def cuadrado(x):
    resultado = x ** 2
    return resultado
```

Podríamos ejecutar esta función y vemos que nos devolverá el cuadrado del número que hemos introducido:

```
cuadrado(8) # Devolverá 16
```

Para convertir una función normal como esta en una función anónima, necesitamos reducirlo a una única expresión. En este ejemplo, se puede ver claramente que la expresión que calcula el cuadrado es `x ** 2`. Con esta expresión podríamos convertir la función en una expresión `lambda`. Para ello seguimos este esquema:

```
lambda parámetro_1, parámetro_2: expresión
```

En primer lugar, utilizamos la palabra reservada «lambda» seguida de los parámetros que tendrá la función, todos ellos separados por comas. A continuación, ponemos dos puntos (:) y la expresión que queremos evaluar. Para nuestro ejemplo de calcular el cuadrado de un número, su expresión `lambda` seria la siguiente:

```
lambda x: x ** 2
```

Esta expresión nos devuelve un objeto función, el cual podemos asignar a una variable y utilizarlo más adelante:

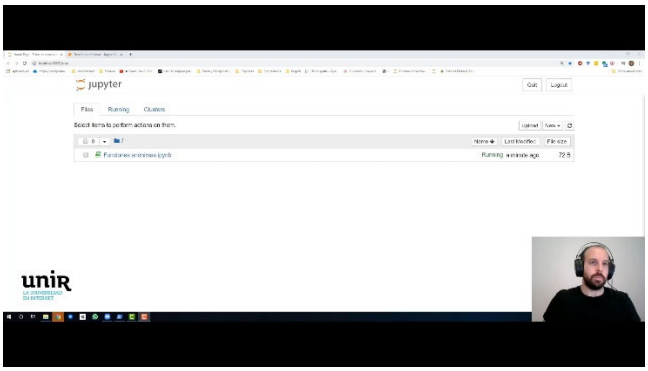
```
cuadrado = lambda x: x ** 2
cuadrado(8) # Devolverá 16
```

La principal ventaja de utilizar este tipo de funciones es combinándolo con las funciones `filter()` o `map()`. La función `filter` nos permite que filtremos elementos de una secuencia si cumplen una función condicional. Esta función debe recibir dos argumentos, el primero de ellos, una función y el segundo, un objeto de tipo secuencia. Vamos a ver un ejemplo en el que buscamos los números pares de una lista.

```
# Definimos la lista con los números que vamos a analizar.
lista = [1,2,3,4,5,6,7,8,9,10]

# Definimos la expresión lambda que analiza si un número es par o no.
es_par = lambda numero: numero % 2 == 0

# Aplicamos el filtro para obtener los números pares. Tenemos que insertarlo en una lista para tener el resultado en formato lista.
list(filter(es_par, lista)) # Nos devolverá [2, 4, 6, 8, 10]
```

La función `map` nos permite aplicar una función a todos los elementos de una secuencia. Para ello, pasamos como argumentos de la función `map`, en primer lugar, la función que queremos aplicar y, luego, el objeto con los elementos a los que se les quiere aplicar la función. Por ejemplo, vamos a aplicar la función `cuadrado` a la lista anterior:

```
# Definimos la lista con los números que vamos a analizar.
lista = [1,2,3,4,5,6,7,8,9,10]

# Aplicamos el filtro para obtener los números pares. Tenemos que insertarlo en una lista para tener el resultado en formato lista.
list(map(cuadrado, lista)) # Nos devolverá [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```



Vídeo 3. Funciones anónimas.

Accede al vídeo a través del aula virtual