Práctica 4: Funciones

Índice

Objetivos y planificación

Objetivos

- Declaración de funciones.
- Documentación de funciones.
- Uso de doctests.
- Uso de módulos.
- Lanzamiento de excepciones
- Uso de Tuplas.

Planificación

Desarrollaremos:

- Una aplicación de cálculo de áreas.
- Una aplicación de operaciones con vectores.

Idea general

Con la aplicación de cálculo de áreas aprenderemos a:

- Declarar funciones.
- Documentar funciones.
- Probar funciones usando doctests.
- Lanzar excepciones.
- Crear e importar módulos.

Idea general

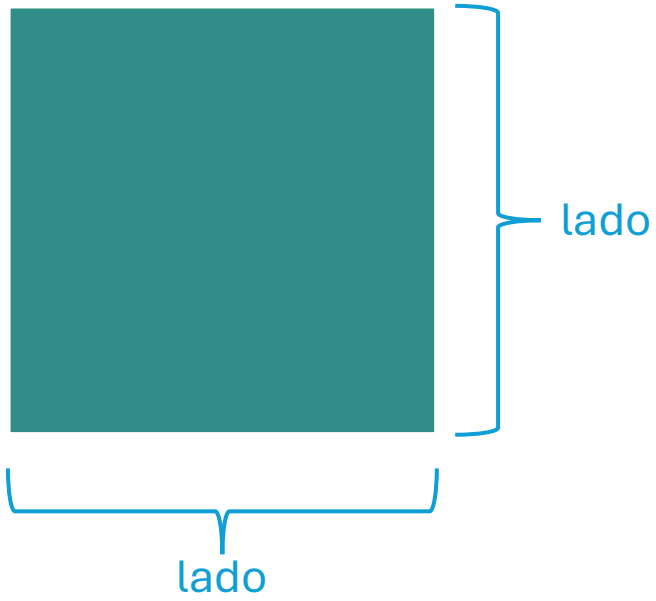
Con la aplicación de operaciones con vectores aprenderemos a:

- Definir y utilizar tuplas.
- Realizar retorno múltiple en funciones.

Declaración de funciones

Cálculo de áreas

Vamos a calcular el área de un cuadrado



$$\text{área} = \text{lado}^2$$

Creación del modulo areas_logic.py

- Vamos a crear un modulo de Python que nos permita calcular el área de un cuadrado.
- Para ello, crea en tu directorio un fichero `areas_logic.py`

Función para calcular el área de un cuadrado

Una función necesita:

- Un nombre.
- Unos parámetros de entrada.
- Devolver un resultado.

¿Qué nombre, parámetros, y valor de retorno debería tener nuestra función?

Definición de una función

Sintaxis para definir de funciones

```
def function_name(param1, param2, ...):
    # Code for the function
    # …
    return <expression>
```

Función para calcular áreas

En el modulo `areas_logic.py`

- Define una función:
  - De nombre `calculate_square_area`.
  - Que reciba el parámetro `lado`.
- Dentro de la función:
  - Declara una variable `area` que calcule el área utilizando el `lado` recibido.
  - Devuelve el valor de `area`.

Documentación de funciones

Documentación de funciones

Como mínimo es necesario documentar.

- Qué hace.
- Qué parámetros recibe.
- Qué valor de retorno devuelve.

Para ello, en Python se utilizan *docstrings*.

Docstrings

```
def function_name(param1, param2, ...):
    """One-line description of what it does.

    Further explanations if necessary.

    Args:
        param1 (type): Description of the first parameter.
        param2 (type): Description of the second parameter.
        ...: Description of additional parameters.

    Returns:
        type: Description of the return value.
    """
    # Code for the function
```

Documenta tu función

Utiliza los docstrings de Python para documentar tu función `calculate_square_area`

Pruebas

Es muy importante comprobar si nuestra función está funcionando correctamente.

No es necesario esperar a finalizar el programa completo, podemos probar cada función por separado.

Doctest

Dentro del docstring de una función, se pueden definir **doctests**.

```
def add(number1, number2):
    """     ... previous documentation
    Examples:
        >>> add(3, 4)
        7
    """
    result = number1 + number2
    return result
```

`>>>` Sentencia python

Respuesta que debería mostrar la consola

Ejecución de doctest

Para ejecutar los doctest de todas las funciones, añade este código al final de tu modulo y ejecútalo.

```
if __name__ == "__main__":
    import doctest

    doctest.testmod(verbose=True)
```

Doctest

Podemos definir tantos doctest como queramos dentro de una misma función.

```
def add(number1, number2):
    """... previous documentation
    Examples:
        >>> add(3, 4)
        7
        >>> add(0, 0)
        0
    """
    result = number1 + number2
    return result
```

Prueba tu función

Añade al docstring de la función `calculate_square_area` dos pruebas utilizando doctest.

Módulos e importación

Módulos

Una aplicación de python se divide en módulos.

- Cada fichero `.py` es un modulo.
- Cada modulo contendrá funciones.
- Todas las funciones de un mismo modulo estarán relacionadas.
  - Por ejemplo, `areas_logic.py` contendrá distintas funciones para el cálculo de áreas de diferentes figuras geométricas.

Módulo main

Cuando ejecutamos un programa en Python, siempre ejecutamos un único modulo. Nos referimos a ese módulo como *Módulo main*.

Si una aplicación tiene varios módulos, solo ejecutaremos el *Módulo main*, y este se encargará de importar el resto de módulos que necesite.

Módulo main

El programa comienza su ejecución en esta línea.

- Podemos escribir código libremente, pero seguiremos siempre la convención de llamar a la función `main`.

```
import areas_logic

def main():
    """Main function."""
    # Here we add the code we want to run when executing the module.

if __name__ == "__main__":
    main()
```

Módulo main

En la función `main` escribiremos el código que queramos que se ejecute al arrancar el modulo main.

```
import areas_logic

def main():
    """Main function."""
    # Here we add the code we want to run when executing the module.

if __name__ == "__main__":
    main()
```

Módulo main

Esta línea nos permite indicar que vamos a utilizar las funciones que hemos definido en el modulo `areas_logic`

```
import areas_logic

def main():
    """Main function."""
    # Here we add the code we want to run when executing the module.

if __name__ == "__main__":
    main()
```

Módulo main

Para llamar a `calculate_square_area`, escribimos el nombre del modulo importado y el de la función separados por un punto.

```
import areas_logic

def main():
    """Main function."""
    square_area = areas_logic.calculate_square_area(4)
    # More code below

if __name__ == "__main__":
    main()
```

Módulo

Función

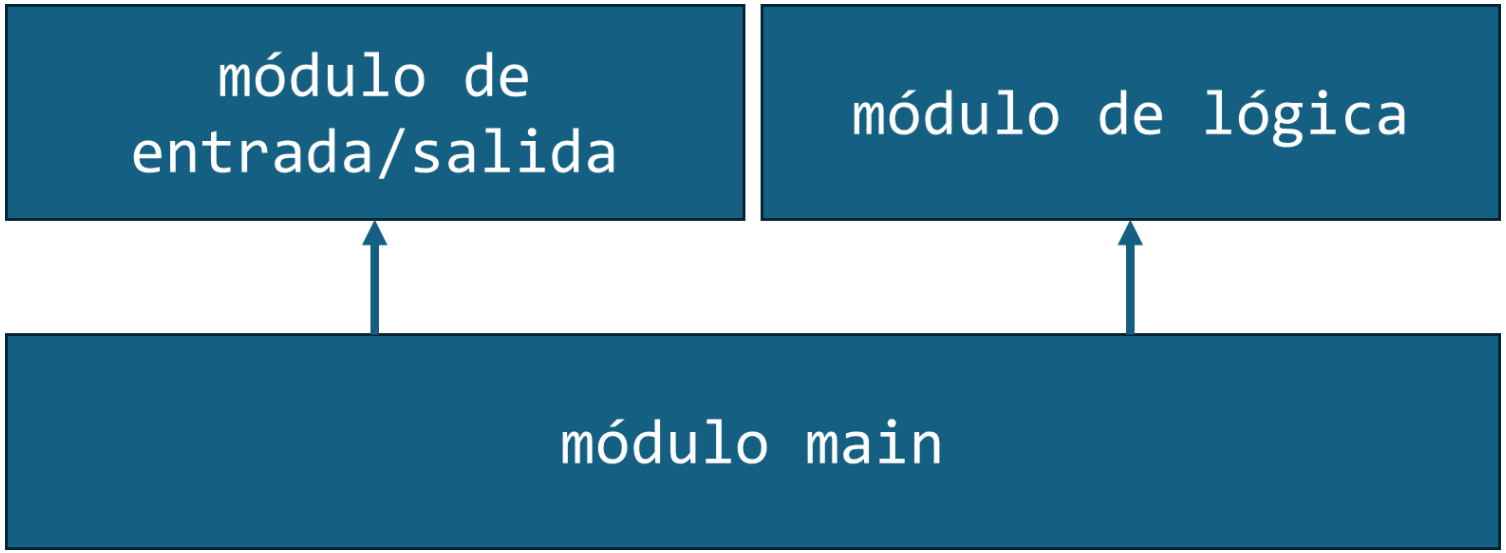
Módulo main

Modifica la función `main` para que:

- Solicite al usuario el lado del cuadrado.
- Calcule el área utilizando el dato introducido.
- Imprima un mensaje mostrando el lado introducido y el área resultante.

División de responsabilidades

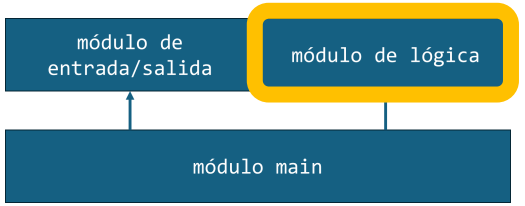
Durante la asignatura organizaremos los módulos según sus responsabilidades:



División de responsabilidades

Módulos de lógica:

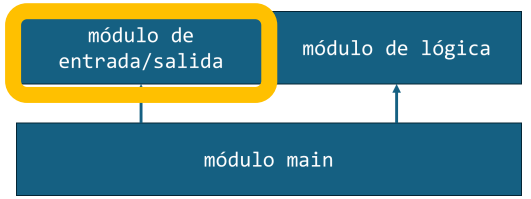
- Solo contienen funciones.
- No realizan **nunca** entrada/salida (no se utilizan las funciones `print` ni `input`, ni llaman a funciones que las utilicen).
- Todas las funciones devuelven siempre un valor.
- Los nombraremos `<prefix>_logic.py`



División de responsabilidades

Módulos de entrada/salida:

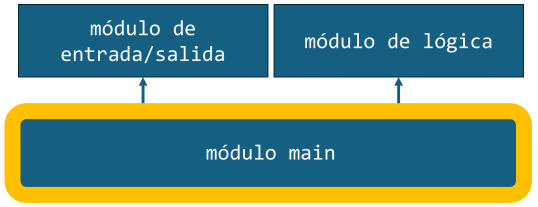
- Solamente contienen funciones que realizan entrada/salida (llaman a `print` o a `input`).
- Facilitan las tareas de imprimir en la consola, pedir datos al usuario y validarlos.
- Los nombraremos `<prefix>_io.py`
- No lo usaremos por el momento
  - (Lo crearemos al final de la clase.)



División de responsabilidades

Módulo main:

- Siempre tiene una función main que comienza la ejecución del programa.
- Importa los módulos de lógica y entrada/salida y los utiliza.
- Puede realizar entrada/salida (llamar a `print` y a `input`, o funciones que las utilicen).
- Los nombraremos `<prefix>_main.py`



Excepciones

Validación de argumentos

¿Cuál debería ser el resultado de nuestra función si introducimos un valor negativo?

```
def calculate_square_area(side):
    """...previous documentation
    Examples:
        >>> calculate_square_area(-3)
        ???
    """
```

Validación de argumentos

Si aplicamos la fórmula que tenemos (lado^2), el área de -3 debería ser 9 ¿Tiene esto sentido?

```
def calculate_square_area(side):
    """...previous documentation
    Examples:
        >>> calculate_square_area(-3)
        9
    """
```

¿Tiene sentido este resultado?

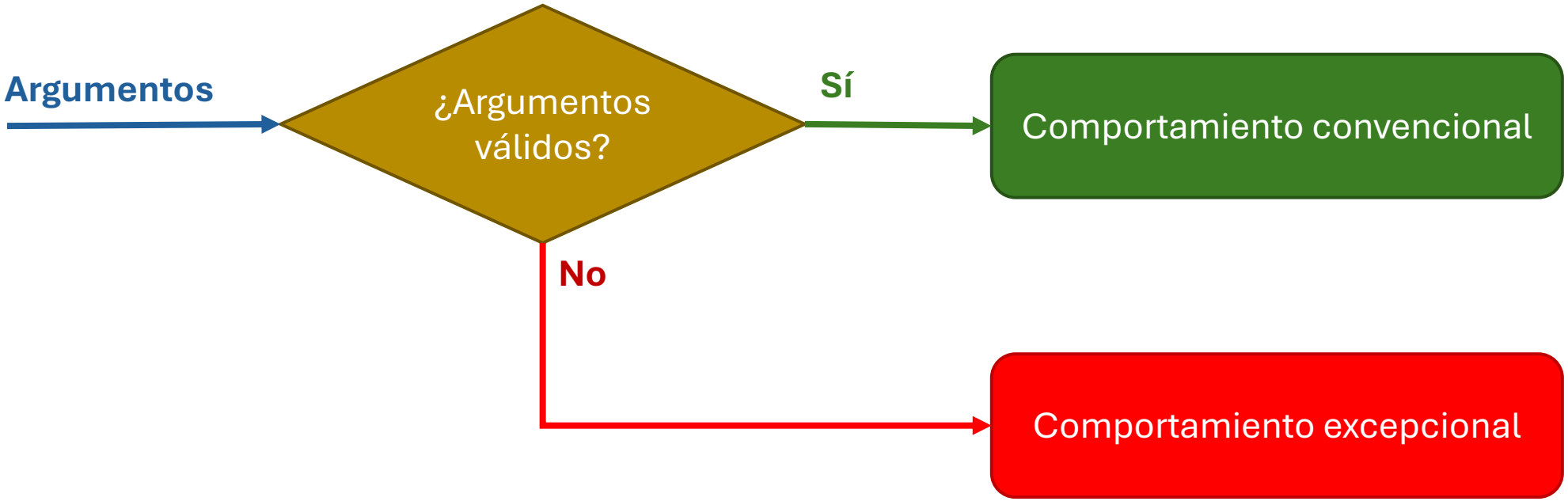
¿Tiene sentido un lado de longitud negativa?

Validación de argumentos

Si se introduce un valor inválido, estamos ante una situación excepcional.

Las situaciones excepcionales deben tratarse de forma diferente a las convencionales.

Validación de argumentos



Validación de argumentos

Si hay valores que no son válidos para nuestra función, debemos controlarlos al inicio de la misma.

```
def calculate_square_area(side):
    """documentation"""
    if side < 0:
        # Do something
    area = side**2
    return area
```

Excepciones

Cuando recibamos argumentos inválidos lanzaremos un error que finalizará el programa.

Llamamos a estas señales que nos permiten comunicar un error "*Excepciones*".

Lanzamiento de excepciones

Para lanzar una excepción, usamos la palabra reservada *raise,* seguida del tipo de error y el mensaje informando de qué ha sucedido.

```
def calculate_square_area(side):
    """documentation"""
    if side < 0:
        raise ValueError("The side must be positive.")
    area = side**2
    return area
```

Lanzamiento de excepciones

La palabra reservada raise para lanzar la exepción

El tipo de error:
En este caso, ValueError

Mensaje que informa del motivo por el que se lanza la excepción

```
def calculate_square_area(side):
    """documentation"""
    if side < 0:
        raise ValueError("The side must be positive.")
    area = side**2
    return area
```

Validación de argumentos

Haz que tu función `calculate_square_area` lance una excepción de tipo `ValueError` si recibe un valor no válido para el lado.

Validación de argumentos

A continuación, ejecuta el modulo `areas_main.py` y comprueba qué sucede cuando introduces un valor negativo.

Traza de error

Verás un mensaje similar a este.

```
Traceback (most recent call last):
  File “areas_logic.py", line 14, in <module>
    calculate_square_area(-3)
  File " areas_logic.py ", line 4, in
calculate_square_area
    raise ValueError("The side must be positive.")
ValueError: The side must be positive.
```

La línea en la que se ha llamado a la función `calculate_square_area`.

La línea en la que se ha lanzado el error.

El tipo de excepción y el mensaje de error

Doctest con excepciones

Cuando hacemos pruebas, debemos comprobar que nuestra función se comporta correctamente:

- Cuando recibe los argumentos esperados.
  - Muestra el resultado correcto.
- Cuando recibe argumentos no válidos.
  - Lanza el error.

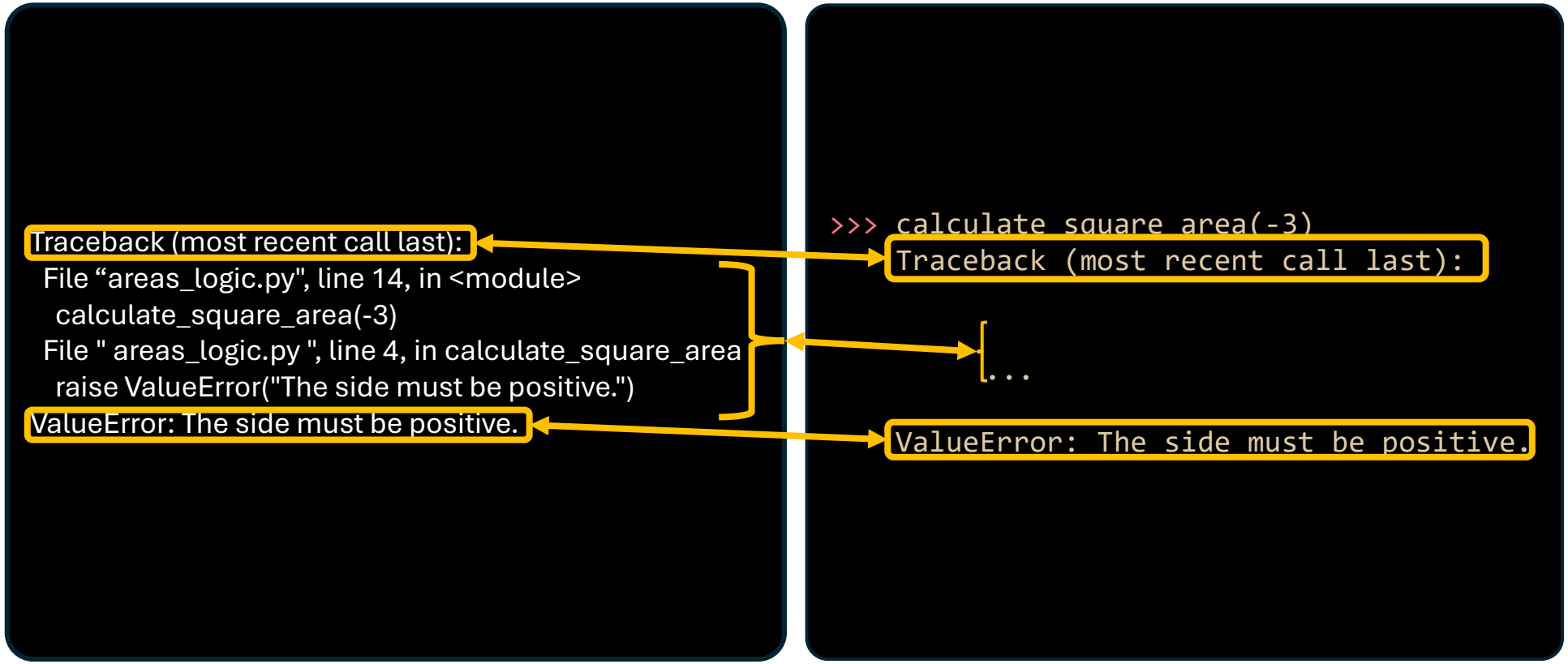
Doctest con excepciones

```
def calculate_square_area(side):
    """...previous documentation
    Examples:
        >>> calculate_square_area(5)
        25
        >>> calculate_square_area(-3)
        Traceback (most recent call last):
            ...
        ValueError: The side must be positive.
    """
    if side < 0:
        raise ValueError("The side must be positive.")
    area = side**2
    return area
```

Estas dos líneas siempre son iguales

El tipo de error y el mensaje deben coincidir con el esperado

Doctest con excepciones



Los 3 puntos se llaman *elipsis* y sirven para evitar comprobar el texto entre la comprobación previa y posterior.

Suele utilizarse en excepciones, ya que basta comprobar el inicio y el final de la traza.

¿Qué probar en los doctests?

Cuando se hacen pruebas en los doctests, deberíamos realizar:

- Una prueba con argumentos convencionales.
- Una prueba con argumentos inválidos (si los hay).
- Una prueba con argumentos límite (si los hay).
  - En el límite entre los válidos e inválidos.

Valores límite

¿Qué debería suceder cuando el lado es 0?

```
def calculate_square_area(side):
    """...previous documentation
    Examples:
        >>> calculate_square_area(0)
        ???
    """
```

Valores límite

¿Tiene sentido decir que un cuadrado de lado 0 tiene área 0?

- Sí

¿Tiene sentido impedir introducir lados de valor 0?

- Depende del contexto de nuestro programa, en algunos casos podría tener sentido impedirlo.
- Incluirlo o no es una *decisión de diseño.*
  - Ninguna opción es intrínsecamente incorrecta, pero hay que elegir una que sea adecuada para nuestro problema.
  - En este caso, aceptaremos lados de longitud 0 como válidos.

Valores límite

Añadir pruebas para los argumentos límite es importante:

- Ayuda a razonar y tomar decisiones sobre dónde poner los límites.

Reutilización de funciones

Área del rectángulo

En el modulo `areas_logic.py`, añade una nueva función para calcular el área de un rectángulo.

Aplica todo lo que hemos visto hasta ahora:

- Dale un nombre adecuado y elige los parámetros de entrada.
- Implementa la función.
- Documéntala.
- Define las pruebas y comprueba que funcionan.
- Utilízala desde el modulo main.

Reutilizando funciones

El área del cuadrado es un caso específico de el área de un rectángulo.

Modifica la función `calculate_square_area` para resolverla utilizando la nueva función que has creado para calcular el área de un rectángulo.

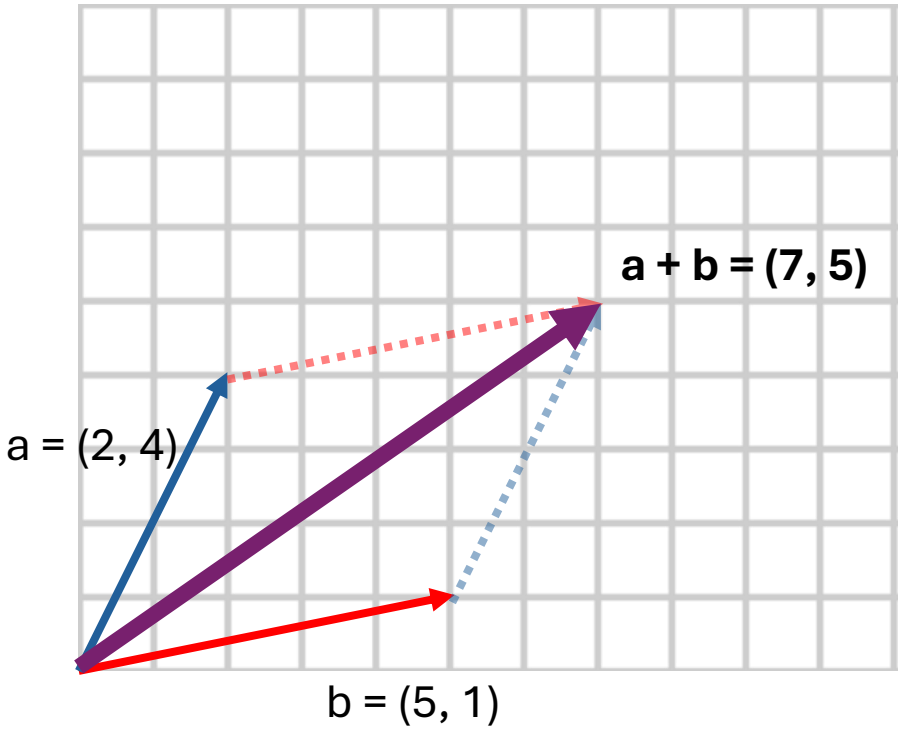
Reutilizando funciones

¿Has necesitado cambiar algo en la documentación de tu función `calculate_square_area`?

Tuplas

Suma de vectores

Vamos a hacer un nuevo programa que sume vectores.



$a = (a_x, a_y)$

$b = (b_x, b_y)$

$a + b = (a_x + b_x, a_y + b_y)$

Suma de vectores

Un vector bidimensional se compone de dos valores:

- Coordenada x
- Coordenada y

Queremos hacer una función que:

- Reciba dos vectores.
- Devuelva el vector resultante de su suma.

Suma de vectores

¿Podemos hacer esto con lo que hemos visto hasta ahora?

Datos estructurados

En ocasiones, necesitamos representar un dato como una combinación de valores relacionados.

- Hora: hora, minutos, segundos
- Fecha: día, mes, año
- Persona: nombre, apellido_1, apellido_2, edad

Tuplas

Cuando necesitamos representar datos estructurados, podemos utilizar **tuplas**, con la siguiente sintaxis:

```
time = (13, 33, 45)  # (hour, minute, second)
birthdate = (15, 6, 1995)  # (day, month, year)
coordinates_oviedo = (43.3619, -5.8494)  # (latitude, longitude)
person = ("Lara", "Pérez", "Gómez", 25)  # (name, surname_1, surname_2, age)
house = (3, 2, False, True)  # (Bedrooms, Bathrooms, Pool, Barbecue)
```

Tuplas

Cada elemento de la tupla tiene un índice, y podemos utilizarlo para acceder a él con la siguiente sintaxis:

```
          0   1   2
          ↓   ↓   ↓
time = (13, 33, 45)  # (hour, minute, second)

hour = time[0]  # 13
minute = time[1]  # 33
second = time[2]  # 45
```

Tuplas

Sin embargo, no podemos asignar valores a una tupla ya creada, son **inmutables.**

```
time = (13, 33, 45)  # (hour, minute, second)

hour = time[0]  # 13
minute = time[1]  # 33
second = time[2]  # 45

time[0] = 14 ✖  # This will raise an error because tuples are immutable
```

Tuplas

Podemos sin embargo crear tuplas nuevas con valores actualizados.

```
time = (13, 33, 45)  # (hour, minute, second)

hour = time[0]  # 13
minute = time[1]  # 33
second = time[2]  # 45

new_hour = 14
new_time = (new_hour, minute, second)  # New tuple with the updated hour
```

Suma de vectores

1. Crea un nuevo modulo `vectors_logic.py`.
2. Crea en él la función `add_vectors` :

```
def add_vectors(vector1, vector2):
    """Documentation"""
    # 1. Obtain the x and y coordinates from vector 1
    # 2. Obtain the x and y coordinates from vector 2
    # 3. Create a new_vector with the sum of the coordinates
    return new_vector
```

Suma de vectores

3. Crea un nuevo modulo `vectors_main.py`.
4. Crea en él una función `main`, importa `vectors_logic`, y prueba la función `add_vectors`.
   - El programa debería solicitar al usuario las coordenadas de cada vector, y mostrar el resultado de su suma.

Desempaquetado de tuplas

Cuando necesitamos asignar el contenido de una tupla a distintas variables, podemos hacer desempaquetado de tuplas en vez de asignar a través de cada índice.

```
time = (13, 33, 45)  # (hour, minutes, seconds)

hour, minute, second = time  # hour = 13, minute = 33, second = 45

# Same thing as doing this:
# hour = time[0]  # 13
# minute = time[1]  # 33
# second = time[2]  # 45
```

Desempaquetado de tuplas

Revisa tu aplicación de suma de vectores y utiliza desempaquetado de tuplas donde sea posible.

Módulos de Entrada/Salida

Validación de entrada de usuario

Si observas las funciones `main` de tus dos programas, verás que en ellas utilizas a menudo la función `input`.

Antes de llamar a una función utilizando datos proporcionados por el usuario, es importante comprobar que los datos son correctos.

Funciones de entrada

En vez de hacer todas esas comprobaciones en el main, podemos crear un modulo aparte para funciones de entrada y salida llamado `user_io.py`

Funciones de entrada

En ese modulo podemos crear funciones de entrada que soliciten datos al usuario y verifiquen si son correctos.

- Si no lo son, lanzamos una excepción.

Funciones de entrada

Crea una función `ask_integer`:

- Recibe como parámetro el mensaje para mostrar al usuario.
- Devuelve el entero introducido por el usuario.

Crea una función `ask_positive_integer`:

- Recibe como parámetro el mensaje para mostrar al usuario.
- Si el número introducido es menor que 0, lanza una excepción ValueError.
- Devuelve el entero introducido por el usuario.
- **¿Puedes reutilizar alguna función aquí?**

Módulo de entrada/salida

Ahora que dispones del módulo `user_io.py`, impórtalo tanto en el módulo `areas_main.py` como en `vectors_main.py` y utiliza sus funciones para pedir la entrada de usuario adecuada en cada momento.

Entrada/Salida

En el módulo `vectors_main` crea una función `ask_vector`:

- Recibe un mensaje para mostrar al usuario.
- Solicita al usuario el valor x para el vector.
- Solicita al usuario el valor y para el vector.
- Devuelve el vector como una tupla.

Modifica el código de la función `main` para utilizar `ask_vector` en vez de pedir cada coordenada.

Entrada/Salida

¿Por qué ubicamos `ask_vector` en `vectors_main` y no en `user_io`?

Tal y como hemos diseñado nuestra aplicación, `user_io` es un modulo de carácter general para ser usado por cualquier aplicación.

Añadir una operación específica de vectores no tendría sentido en este caso, por eso la ubicamos en el `vectors_main`.

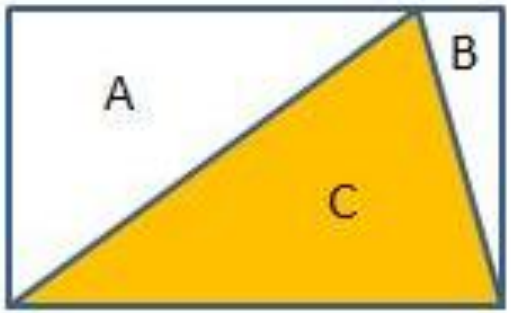
Si en el futuro nuestra aplicación `vectors_main` creciera y tuvieramos multitud de funciones de entrada y salida para vectores, podríamos crear un módulo `vectors_user_io` y ubicarlas ahí.

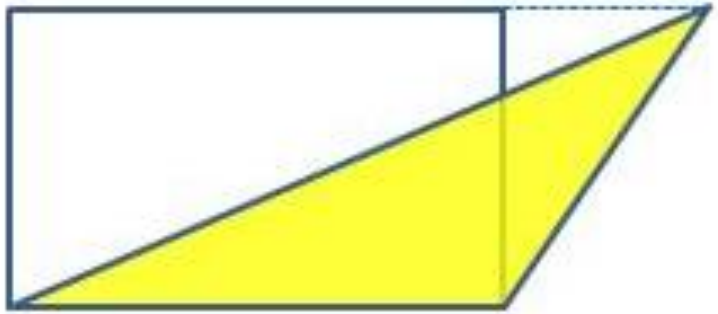
Trabajo en casa

Área de un triángulo

Añade al modulo `areas_logic` la función `calculate_triangle_area`.

- ¿Puedes reutilizar alguna de las funciones que ya tienes?





Operaciones con vectores

Añade al modulo `vectors_logic` las siguientes funciones.

- `substract_vectors`
  - Recibe dos vectores devuelve el resultante de restar el segundo al primero.
- `multiply_vector_by_scalar`
  - Recibe un vector, un número real, y devuelve el resultado de su multiplicación.

Completa las aplicaciones

Para las funciones que acabas de crear, asegúrate de que:

- Validas las entradas y lanzas excepciones cuando sea necesario.
- Documentas las funciones.
- Añades pruebas para argumentos convencionales, argumentos inválidos y argumentos límite.
- Utilízalas en el modulo main correspondiente, utilizando las funciones de entrada/salida de las que dispones.