



Funciones y Modularización

Sesión Conceptual 2





Inicio

{desafío}
latam_



10 minutos



¿Qué es un Proyecto en Python?



Desarrollo

{desafío}
latam_



70 minutos

/* Organización de un proyecto en Python */

Docstrings

Los docstrings son una documentación que nosotros mismos podemos implementar dentro de nuestras funciones para poder recordar, cuando pasa el tiempo, cuál es la intención de la función, cómo funciona y qué parámetros son necesarios para que funcione de manera apropiada.

El docstring se implementa al inicio de la función utilizando 3 pares de comillas (pueden ser simples o dobles):

```
def elevar(base, exponente):  
    """Esta función tiene como objetivo  
    elevar una base a un exponente"""  
    return base**exponente
```

Docstrings

El poder implementar esta mini-documentación permite que editores de texto como Jupyter, Atom o VSCode reconozcan esto:

```
34 def elevar(base, exponente):  
35     """Esta función tiene como objetivo  
36     elevar una base a un exponente"""  
37     return base**exponente  
38  
39  
40     (base, exponente) -> Any  
41     Esta función tiene como objetivo elevar una base a un exponente  
42  
43 elevar()
```



```
def elevar(base, exponente):
```

```
    """[summary]
```

Args:

```
    base ([type]): [description]
```

```
    exponente ([type]): [description]
```

Returns:

```
    [type]: [description]
```

```
    """
```

```
    return base**exponente
```

```
def ele (base, exponente) -> Any
```

```
    """
```

```
    el base ([float]): Base de la potencia.
```

Arg

Esta función tiene como objetivo
elevar una base a un exponente.

Arg

Args:

base ([float]): Base de la potencia.

Ret

exponente ([float]): Exponente de la potencia.

```
    """
```

Returns:

```
    ret
```

[float]: Se retorna un float resultante de elevar base a
exponente.

```
elevar()
```

```
def elevar(base, exponente):  
    """[summary]  
  
    :param base: [description]  
    :type base: [type]  
    :param exponente: [description]  
    :type exponente: [type]  
    :return: [description]  
    :rtype: [type]  
    """  
    return base**exponente
```

```
    :param base: Corresponde a la Base de la Potencia.  
    :ty (base, exponente) -> Any  
    :pa  
    :ty base: Corresponde a la Base de la Potencia.  
    :re  
    :rt Esta función tiene como objetivo  
    """ elevar una base a un exponente.  
    ret :param base: Corresponde a la Base de la Potencia.  
elevar( :type base: [float]  
        :param exponente: Corresponde al exponente de la Potencia.  
        :type exponente: [float]  
        :return: Corresponde a la resultante de la potencia.  
        :rtype: [float]  
elevar()
```

```
def elevar(base, exponente):  
    """[summary]  
  
    Arguments:  
        base {[type]} -- [description]  
        exponente {[type]} -- [description]  
  
    Returns:  
        [type] -- [description]  
    """  
    return base**exponente
```

```
64     """Esta función tiene como objetivo  
65     elevar una base a un exponente.  
66  
67     Arg (base, exponente) -> Any  
68  
69     Esta función tiene como objetivo  
70     elevar una base a un exponente.  
71  
72     Ret  
73     Arguments:  
74     """ base {[float]} – Base de la potencia. exponente {[float]} –  
75     ret Exponente de la potencia.  
76  
77     Returns:  
78     [float] – Se retorna un float resultante de elevar base a  
79     elevar() exponente.
```

```
def elevar(base, exponente):  
    """Esta función tiene como objetivo  
    elevar una base a un exponente.
```

Parameters

base : [float]

Base de la Potencia.

exponente : [float]

Exponente de la Potencia

Returns

[float]

Retorna el resultado de elevar base a exponente

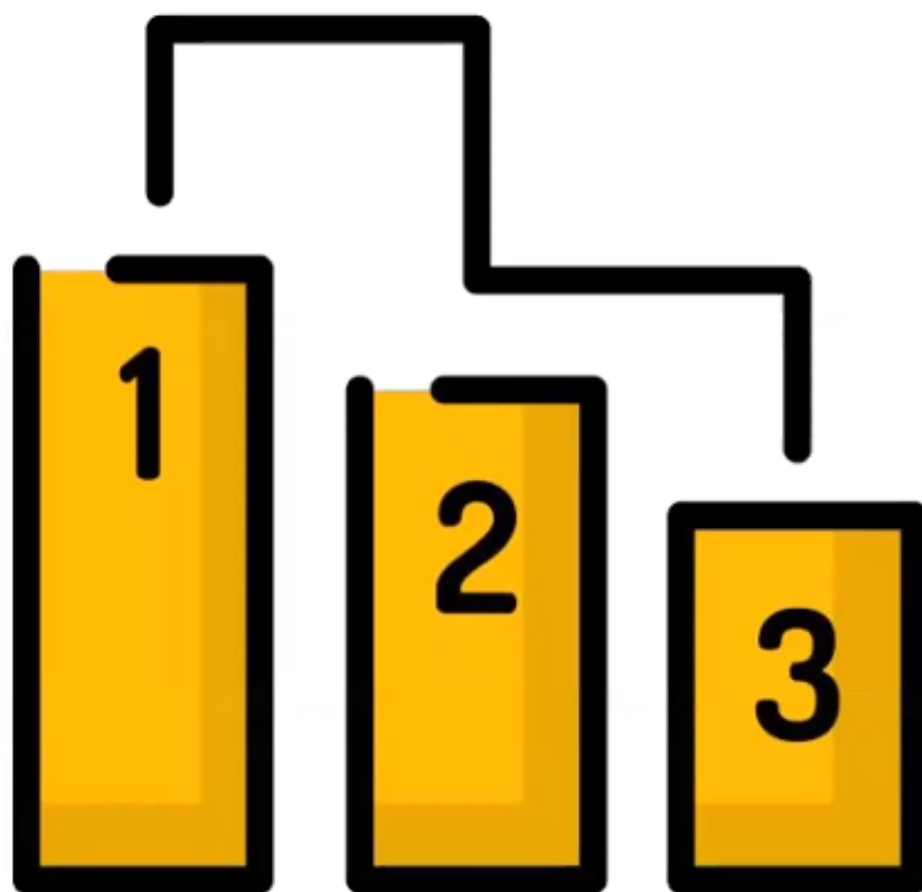
"""

return base**exponente

```
base : [float]  
(base, exponente) -> Any  
exp  
Esta función tiene como objetivo  
elevar una base a un exponente.  
Ret  
--- Parameters  
[fl  
base : [float]  
""" Base de la Potencia.  
ret exponente : [float]  
Exponente de la Potencia  
Returns  
elevar()
```

Refactorización

Al momento de crear una solución, esta no aparece de manera directa, sino por etapas, donde uno va haciendo pruebas hasta verificar que el código implementado efectivamente solucione el problema. Una vez alcanzada la solución muchas veces notamos que el código podría ser organizado y estructurado de manera mucho más eficiente de lo que lo tenemos actualmente.





Quiz



/* Modularización */

Ventajas de la modularización

1

Cuando el código es refactorizado, habrán funciones necesarias para llevar a cabo nuestro programa. El problema es que el código ejecutable de nuestro programa puede quedar muy abajo en el script lo que impide un buen entendimiento del código.

2

Existen ocasiones que la solución creada en un proyecto puede ser útil en otro proyecto, por lo tanto, es posible reutilizar dicho código utilizándolo como un módulo.

3

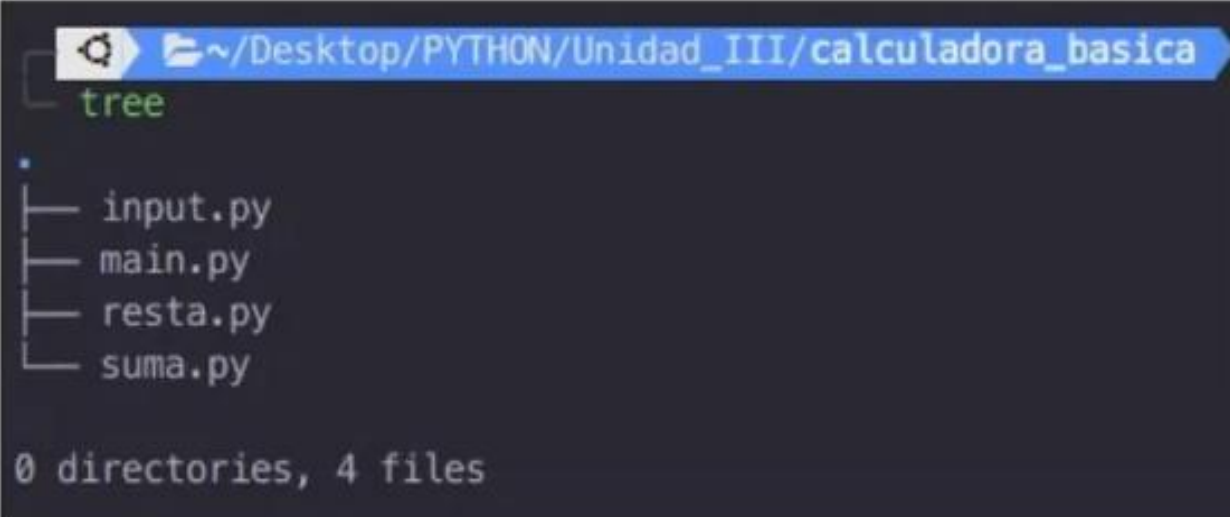
En el desarrollo de proyectos de gran envergadura, rara vez serán realizados por completo por un solo desarrollador, es por eso que modularizar permite aislar tareas para que distintos desarrolladores las ejecuten.

4

Permite generar estructuras ordenadas y escalables en caso de que el desarrollo necesite de la adición de más features en el futuro.

¿Cómo se puede aplicar el concepto de modularización acá?

1. Primero que todo es conveniente crear una carpeta correspondiente a nuestro programa. En nuestro caso se llama `calculadora_basica`. Esta carpeta contiene los archivos `main.py`, `suma.py`, `resta.py` e `input.py`. Cada uno de estos scripts corresponderá a nuestros módulos.

A terminal window with a dark background. The title bar shows the path `~/Desktop/PYTHON/Unidad_III/calculadora_basica`. The command `tree` has been executed, showing a directory structure with four files: `input.py`, `main.py`, `resta.py`, and `suma.py`. At the bottom, it says `0 directories, 4 files`.

```
~/Desktop/PYTHON/Unidad_III/calculadora_basica
tree
.
├── input.py
├── main.py
├── resta.py
└── suma.py

0 directories, 4 files
```

¿Cómo se puede aplicar el concepto de modularización acá?

2. Cada uno de los módulos alojarán cada función:

```
suma.py > ...  
1  def sumar(x,y):  
2      print(f'El resultado es {x + y}')3
```

```
resta.py > ...  
1  def restar(x,y):  
2      print(f'El resultado es {x - y}')3
```

```
input.py > ...  
1  def tomar_datos():  
2      x = int(input('Ingrese el primer número: '))  
3      y = int(input('Ingrese el segundo número: '))  
4      return x, y
```

¿Cómo se puede aplicar el concepto de modularización acá?

3. Una vez creado cada módulo, estos deben ser invocados desde el archivo principal. Para ello las invocaremos como si se tratara de librerías; donde lo usual es referirse a ellas en alguna de estas 3 formas:

```
import modulo  
import modulo as alias  
from iarchivo import función
```


Experiencia de usuario

Pausas

```
import time
time.sleep(3)
print('Han pasado 3 segundos')
```

`time.sleep(n)` permitirá hacer que Python espere `n` segundos para la siguiente línea.

+

Limpiar la Pantalla

Podemos utilizar la librería `sys`, para detectar nuestro Sistema Operativo. `sys.platform` permite detectar cuál es el sistema operativo-

+

Terminar el Programa

En ocasiones será prudente terminar el programa, debido a que se alcanzó el final de este o porque alguna de las opciones del programa considera una finalización adelantada. Para ello Python provee el comando `exit()` el cual permitirá finalizar el programa.



Ejercicio Guiado

"Pizza App"





Cierre



{desafío}
latam_

*Academia de
talentos digitales*

www.desafiolatam.com



/DesafioLatam



/DesafioLatam



/DesafioLatam



/DesafioLatam