Estructuras de datos y funciones

Organización de un Proyecto en Python y modularización

{desafío} latam_

Utilizar estructuras de datos apropiadas para la elaboración de un algoritmo que resuelve un problema acorde al lenguaje Python.

Codificar un programa utilizando funciones para la reutilización de código acorde al lenguaje Python.

- Unidad 1:
  Introducción a Python
- Unidad 2:
  Sentencias condicionales e iterativas
- Unidad 3:
  Estructuras de datos y funciones

Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Explica el sentido de utilizar funciones dentro de un programa distinguiendo su definición versus su invocación.*
- *Utiliza funciones preconstruidas y personalizadas por el usuario con paso de parámetros y que obtienen un retorno.*

{desafío} latam_

/* Organización de un Proyecto en Python */

Docstrings

Es muy probable que a medida que un proyecto en Python crece se generen muchas funciones, que van quedando guardadas, pero con el tiempo uno va olvidando su funcionalidad. Es por esto que siempre se recomienda como buena práctica que cuando un proyecto comience a crecer se vaya **documentando de manera apropiada**.

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.

Docstrings

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
```

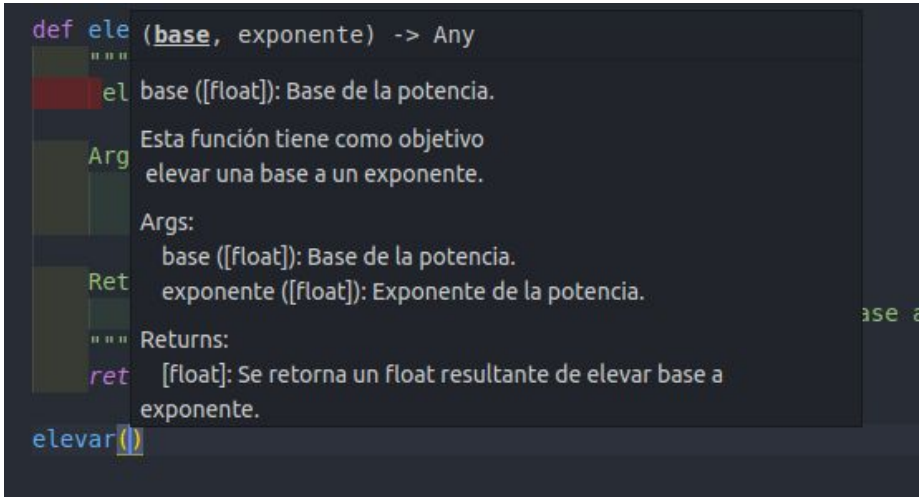
Si invocamos la función, los **editores de texto** serán capaces de mostrarnos el Docstring asociado, por lo que, dependiendo del grado de detalle que se dé a la documentación podremos entender de mejor manera nuestro código.

Tipos de Docstrings

Google

Fomenta generar un pequeño resumen de lo que hace la función, definir los parámetros con el tipo de datos que se espera de ellos, y una descripción y el tipo del retorno de la función.

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



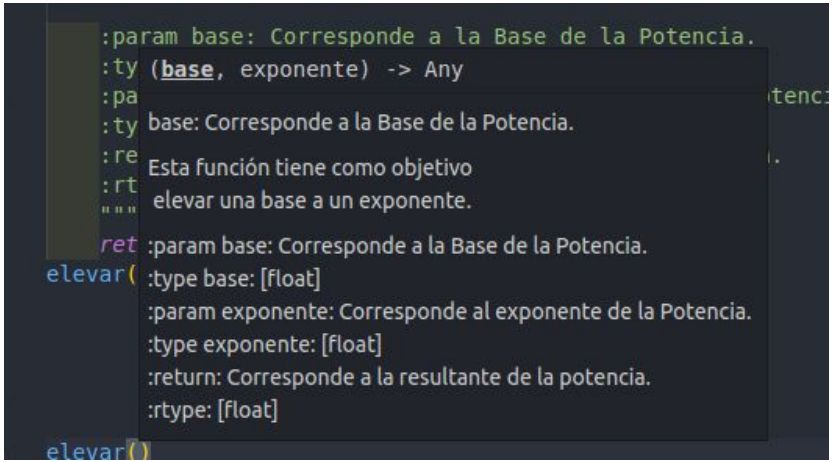
{desafío} latam_

Tipos de Docstrings

Sphinx

Herramienta especializada en la creación automática de documentación. En este caso, es muy similar a la de Google solo que definen los parámetros y sus tipos en líneas distintas.

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



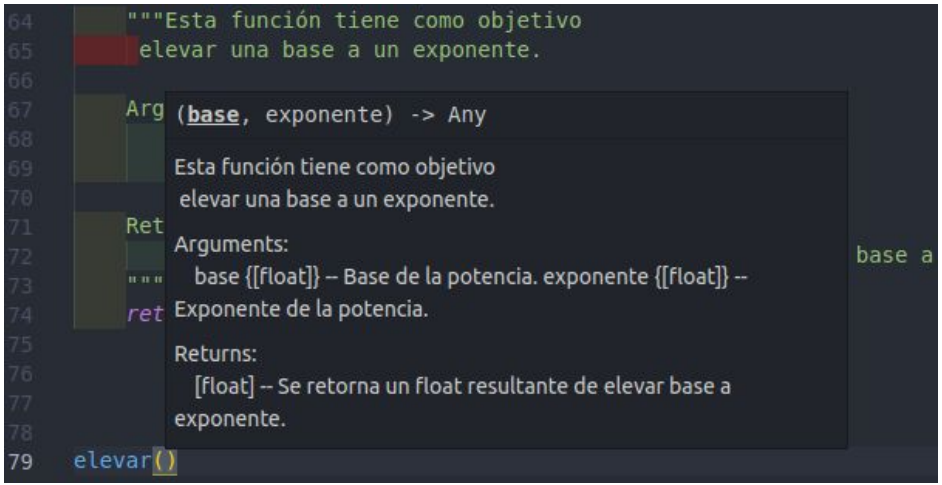
{desafío} latam_

Tipos de Docstrings

Docblockr

Casi idéntico que el de Google, pero con otro tipo de separadores. Si bien indica que se está esperando un argumento, en este caso el argumento base, no genera esa confusión y sensación de error de repetir el argumento al inicio.

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



{desafío} latam_

Tipos de Docstrings

Numpy

Librería de Computación Científica muy popular en Ciencia de Datos. El formato utilizado es reconocido como texto enriquecido en editores como VS Code, por lo que destaca algunos títulos generando una documentación más elegante.


```numpy
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
```

{desafío} latam_

/* Refactorización */

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.

Supongamos el siguiente caso:

```
valor_entrada = 10
valor_1 = valor_entrada**2 + valor_entrada**3
valor_2 = valor_1*2 + valor_1*3 + valor_1*4
valor_3 = valor_2**2 + valor_2**3
valor_4 = valor_3*2 + valor_3*3 + valor_3*4
valor_5 = valor_4**2 + valor_4**3
valor_6 = valor_5*2 + valor_5*3 + valor_5*4
```

Tenemos el siguiente código, el cual realiza un cálculo muy difícil. Si lo miramos bien notamos ciertos patrones, como, por ejemplo, los valores 1, 3 y 5 siguen una misma lógica de sumar el cuadrado y el cubo de un número.
Por otro lado, los valores 2, 4 y 6 están sumando el doble, el triple y el cuádruple de un valor.

Refactorización

La refactorización consistirá en abstraer este código y generar funciones que se encarguen del cálculo de patrones similares, básicamente significa, aplicar el principio DRY.

Creemos las siguientes funciones:

```
def cuadrado_cubo(valor):
    return valor**2 + valor**3
def mult_234(valor):
    return valor*2 + valor*3 + valor*4
```

Al generar estas dos funciones el código puede refactorizarse de la siguiente manera:

```
def cuadrado_cubo(valor):
    return valor**2 + valor**3
def mult_234(valor):
    return valor*2 + valor*3 + valor*4
valor_entrada = 10
valor_1 = cuadrado_cubo(valor_entrada)
valor_2 = mult_234(valor_1)
valor_3= cuadrado_cubo(valor_2)
valor_4 = mult_234(valor_3)
valor_5 = cuadrado_cubo(valor_4)
valor_6 = mult_234(valor_5)
```

Refactorización

Mirando esto notamos que, de hecho, el código se puede refactorizar aún más. Podríamos crear la siguiente función:

```
def op_combinada(valor):
    var_intermedia = cuadrado_cubo(valor)
    return mult_234(var_intermedia)
```

Esta función abstrae pares de operaciones que se están aplicando, luego el código se puede refactorizar de la siguiente manera:

```
def cuadrado_cubo(valor):
    return valor**2 + valor**3
def mult_234(valor):
    return valor*2 + valor*3 + valor*4
def op_combinada(valor):
    var_intermedia = cuadrado_cubo(valor)
    return mult_234(var_intermedia)
valor_entrada = 10
valor_2 = op_combinada(valor_entrada)
valor_4 = op_combinada(valor_2)
valor_6= op_combinada(valor_4)
```

Refactorización

En este caso, sí se acorta el número de operaciones necesarias, pero hay que decir que esto se puede refactorizar aún más:

Podríamos crear la siguiente función:

```
def compose(f, n):
      def fn(x):
      for _ in range(n):
            x = f(x)
      return x
      return fn
```

{desafío} latam_

Permite repetir la ejecución de una función n de veces sobre el resultado que ella misma arroja.

```
def cuadrado_cubo(valor):
    return valor**2 + valor**3
def mult_234(valor):
    return valor*2 + valor*3 + valor*4
def op_combinada(valor):
    var_intermedia = cuadrado_cubo(valor)
    return mult_234(var_intermedia)
def compose(f, n):
      def fn(x):
          for _ in range(n):
              x = f(x)
          return x
      return fn
valor_entrada = 10
valor_6 = compose(op_combinada, 3)(valor_entrada)
```

Refactorización

- Uno puede refactorizar cuantas veces sea necesario, pero ¿es siempre esto lo óptimo? La respuesta es categórica: NO. En este caso particular este nivel de factorización no genera demasiados beneficios, e introduce una función compose() extremadamente compleja, ya que a muy pocos programadores se les ocurriría por sí solos.

- En conclusión, técnicas como la refactorización tienen que utilizarse con cautela, pues la idea de la factorización es siempre introducir funciones que faciliten el entendimiento del código, no que lo compliquen. Por lo tanto, dependiendo de las necesidades, es responsabilidad del desarrollador determinar cuántos niveles de refactorización utilizar y siempre pensando en facilitar la estructura del código y no introducir complejidades innecesarias.

Stackoverflow es la web más grande de ayuda a los programadores. Programadores de todo nivel comparten conocimiento de código acerca de cómo resolver un problema. Es común que ante cualquier duda que uno pueda tener, sin importar el lenguaje de programación, siempre existirá alguna sugerencia al respecto.

/* Modularización */

Ventajas de la modularización

Algunas razones

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

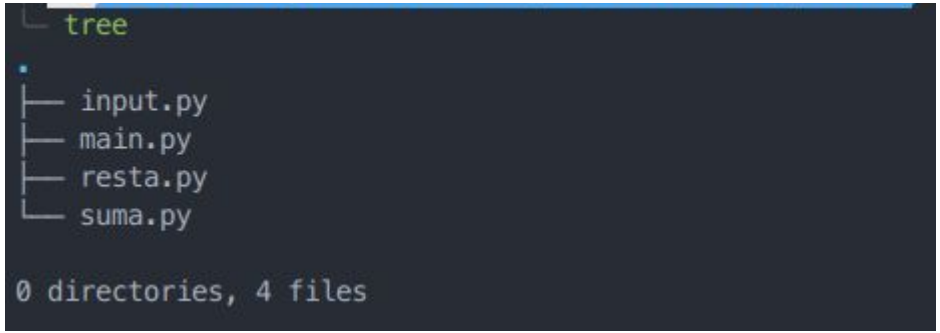
Permite generar estructuras ordenadas y escalables en caso de que el desarrollo necesite de la adición de más features en el futuro.

Cuando se crea un proyecto en Python se recomienda dividir las distintas partes de nuestro proyecto en módulos.
Estos módulos serán distintos scripts de Python (archivos `.py`) que se encargarán de resolver un problema en específico, normalmente encapsulado en una función, la cual se llamará desde un script maestro normalmente llamado `main.py`.

Ejemplo de modularización

Ejemplo de modularización

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.



Ejemplo de modularización

Cada uno de los módulos alojarán cada función:

suma.py > ...

```python
def sumar(x,y):
    print(f'El resultado es {x + y}')

```

resta.py > ...

```python
def restar(x,y):
    print(f'El resultado es {x - y}')

```

input.py > ...

```python
def tomar_datos():
    x = int(input('Ingrese el primer número: '))
    y = int(input('Ingrese el segundo número: '))
    return x, y
```

Ejemplo de modularización

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 archivo import función
```

Ejemplo de modularización

A modo demostrativo utilizaremos las 3 nomenclaturas para entender su uso:

```
import suma
import resta as r
from input import tomar_datos
opcion = input("""Esto es una calculadora:
¿Qué operación le gustaría realizar?
1. Sumar
2. Restar
0. Salir
> """)
if opcion == '1':
    x, y = tomar_datos() # 3ra forma de importar
    suma.sumar(x,y) # 1era forma de importar
elif opcion == '2':
    x, y = tomar_datos() # 3ra forma de importar
    r.restar(x,y) # 2da forma de importar
elif opcion == '0':
    print('Nos vemos a la próxima')
else:
    print('No existe esta Operación')
```

Ejemplo de modularización

Dependiendo de la manera de importar el módulo es cómo tiene que ser referenciado:

- Para el caso de **`import suma`**, la función sumar que está en su interior se debe llamar de la forma **`modulo.funcion`**, es decir, **`suma.sumar()`**.
- Para el caso de **`import resta as r`**, la función resta que está en su interior se debe llamar de la forma **`alias.funcion`**, es decir, **`r.restar()`**.
- Si se utiliza la forma **`from input import tomar_datos`**, se puede llamar la función de manera directa, es decir, **`tomar_datos()`**.

Ejemplo de modularización

Para evitar errores de choques de nombre, es una buena práctica evitar que un módulo y la función al interior de dicho módulo tengan el mismo nombre.

Como se puede apreciar, **main.py** solo se concentra en el funcionamiento de la App. Todas las funcionalidades, normalmente representadas por funciones que representan códigos más complejos, serán manejadas en módulos independientes.

```
if __name__ == '__main__':
```

Una práctica muy común en Python es utilizar el código mencionado en el título para impedir que se disparen salidas inesperadas de nuestros módulos.

{desafío} latam_

Revisa otro ejemplo en el material complementario: Ejemplo de modularización

/* Experiencia de usuario */

Experiencia de usuario

Pausas

A veces el código se ejecuta muy rápido y no permite que el usuario pueda leer apropiadamente lo que ocurre.

Para ello, cuando corresponda, puede ser bueno agregar pausas. Las pausas en Python se pueden agregar mediante la librería time:

```
import time

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

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

Experiencia de usuario

Limpiar la pantalla

Todas las impresiones de pantalla quedan dentro del mismo terminal, lo que en ocasiones, puede terminar en un terminal lleno de texto que empaña la experiencia de usuario y no permite entender completamente el contenido de nuestras salidas.

Una dificultad es que esto se hace de distinta manera dependiendo del sistema operativo.

Podemos utilizar la **librería sys**, para detectar nuestro Sistema Operativo. **sys.platform** permite detectar cuál es el sistema operativo de acuerdo a la siguiente tabla:

System	platform value
AIX	`'aix'`
Linux	`'linux'`
Windows	`'win32'`
Windows/Cygwin	`'cygwin'`
macOS	`'darwin'`

Experiencia de usuario

Limpiar la pantalla

Adicionalmente, **os** permite utilizar comandos propios del sistema operativo para limpiar la pantalla. En un terminal de windows se puede limpiar el terminal utilizando **'cls'** mientras que en **macOS** o **Linux** se hace mediante **'clear'**.

```
import os
os.system('cls') # Windows
os.system('clear') # macOS o Linux
```

Finalmente, se puede generar un código genérico que detecte el sistema operativo y que limpie la pantalla:

```
import os
import sys
# detecta el OS
clear = 'cls' if sys.platform == 'win32' else 'clear'

# ejecuta la limpieza
os.system(clear)
```

Experiencia de usuario

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"

{desafío} latam_

Pizza app

Pizza JAT, una empresa de pizzería a nivel mundial que desea automatizar su proceso de solicitud de Pizzas. Para ello, se le solicita generar un prototipo rápido que abarque los siguientes elementos:

1. Un menú interactivo que permita al usuario personalizar su Pizza.

2. Que se permita cambiar el tipo de masa. Actualmente la Pizzería trabaja con:
    - Masa Tradicional, Masa Delgada, Masa con Bordes de Queso

3. Que se permita cambiar el tipo de salsa. Actualmente la Pizzería trabaja con:
    - Salsa de Tomate, Salsa Alfredo, Salsa Barbecue, Salsa Pesto

Pizza app

4. Que se permita modificar ingredientes:
   - Agregar Ingredientes
   - Eliminar Ingredientes

   Actualmente la pizzería trabaja con los siguientes ingredientes: Tomate, Champiñones, Aceituna, Cebolla, Pollo, Jamón, Carne, Tocino, Queso

5. Estimar el tiempo que tomará en que la pizza esté lista.
   - Un menú que confirme si es que desea ordenar.
   - El tiempo para estar lista serán 20 minutos + 2 minutos por cada ingrediente excluyendo masa y salsa.

6. Una opción que permita mostrar los ingredientes que actualmente tiene la pizza.

Pizza app

Solución

La resolución de este ejercicio es muy larga y se recomienda tomarla paso a paso para entender lo que se hace. Se recomienda ir replicando el problema a la par reescribiendo el código de manera manual y no copiando y pegando.

Revisa el archivo: *"**Solución Ejercicio guiado - Pizza app.pdf**"*

¿Qué es la modularización?

Próxima sesión...

- *Desafío evaluado.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales