Estructuras de datos y funciones

Modularización y experiencia de usuario

{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_

/* 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

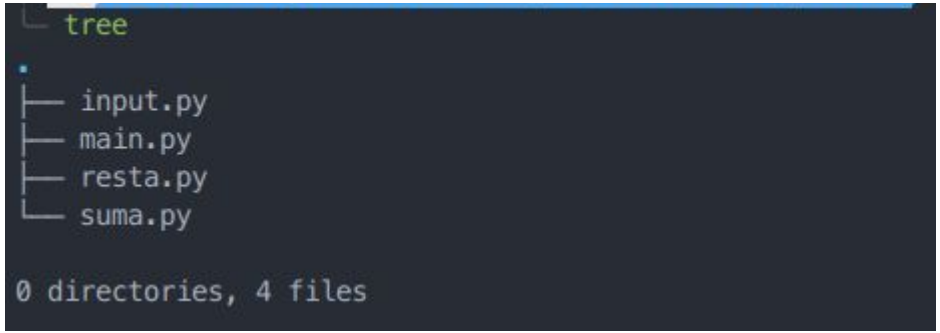
Permite generar estructuras ordenadas y escalables en caso de que el desarrollo necesite de la adición de más features en el futuro.

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.

¿Qué es la modularización?

Próxima sesión...

- *Desafío evaluado.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales