UNIVERSIDAD CATÓLICA DE SANTA MARÍA
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

SESIÓN N° 01:

FUNCIONES Y CARACTERÍSTICAS DE PYTHON

I

OBJETIVOS

- Aplicar el uso de excepciones en Python.
- Conocer las funciones en Python.
- Aprender a diseñar funciones en Python.
- Apreciar las características y ventajas provistas por las funciones para resolver problemas de programación (tipos de valores, número indefinido de parámetros, etc.).

II

TEMAS A TRATAR

- Excepciones.
- Tipos de Excepciones.
- Introducción a Funciones.
- ¿Qué es una función con argumentos?
- Funciones con parámetros por defecto.
- ¿Qué es una función con una cantidad variable de parámetros?
- Funciones con una cantidad variable de parámetros.

III

MARCO TEÓRICO

1. EXCEPCIONES

Los errores que se presentan en ejecución son llamadas excepciones. Durante la ejecución de un programa, si dentro de una función aparece una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó.

Python posee un manejo de excepciones avanzado para detectar eventos y modificar el flujo de la ejecución del programa.

Ejemplo uso de excepciones con listas

```
#creamos una lista con 2 elementos
lista = [1,2]
#intentamos visualizar el cuarto elemento
print(lista[3])
```

Ejemplo 1: Excepciones con listas

Ello genera el error IndexError como se muestra a continuación:

```
Traceback (most recent call last):
  File "c:\LAPTOP\UCSM\2023-II\LP1\Prácticas 2023\Lab01\ejemplo1.py", line 4, in <module>
    print(lista[3])
IndexError: list index out of range
```

Imagen 1: IndexError

Ahora realizaremos la captura de la excepción.

```
#creamos una lista con 2 elementos
lista = [1,2]
#intentamos visualizar el cuarto elemento
try:
    print(lista[3])
except IndexError:
    print("Error, índice no existe")
else:
   print("Todo bien")
finally:
   print("Finalizó la ejecución")
```

Ejemplo 2: Captura de la excepción

En este caso se detalla el error en específico:

```
Error, índice no existe
Finalizó la ejecución
```

Imagen 2: Detalle de error IndexError

Ahora veremos un ejemplo de validación de números:

```
#Validación solo de valores numéricos
while True:
    try:
        numero=int(input("Ingrese un numero:"))
        break
    except ValueError:
        print"Solo es permitido ingresar numeros")
```

Ejemplo 3: Validación de números

Como se verifica a continuación el ciclo finalizará solo cuando se realiza un ingreso válido de un valor numérico.

```
Ingrese un numero:a
Solo es permitido ingresar numeros
Ingrese un numero:$
Solo es permitido ingresar numeros
Ingrese un numero:3
```

Imagen 3: Ingreso válido de un número

2. TIPOS DE EXCEPCIONES

En Python existen diferentes tipos de excepciones que pueden controlarse, todas ellas derivan de una serie de excepciones base. Las excepciones base son las siguientes:

- **Excepción:** tipo de excepción más genérica, de ella derivan todas las excepciones existentes en Python.
- **ArithmeticError:** tipo de excepción genérica para errores aritméticos.
- **BufferError:** tipo de excepción genérica para errores relacionados con buffers.
- **LookupError:** tipo de excepción genérica para errores relacionados con acceso a datos de colecciones.

El grupo de excepciones base anterior da lugar a un grupo de excepciones concretas que te presentamos en la siguiente lista:

- **AssertionError:** excepción que se lanza cuando la instrucción assert falla.
- **AttributeError:** excepción que se lanza cuando hay un error a la hora de asignar un valor a un atributo o cuando se intenta acceder a él.
- **EOFError:** excepción que se lanza cuando la instrucción input no devuelve datos leídos.
- **FloatingPointError:** excepción que ya no se usa.
- **GeneratorExit:** excepción que se lanza cuando se cierra una función de tipo generator o coroutine.
- **ImportError:** excepción que se lanza cuando se intenta importar un módulo al programa y falla.
- **ModuleNotFoundError:** excepción que se lanza cuando se intenta importar un módulo y no se encuentra. Deriva de la anterior.
- **IndexError:** excepción que se lanza cuando se intenta acceder a una posición de una secuencia y ésta es superior a la posición mayor.
- **KeyError:** excepción que se lanza cuando se intenta acceder a la clave de un diccionario y no se encuentra.
- **KeyboardInterrupt:** excepción que se lanza cuando el usuario utiliza el comando de interrupción con el teclado (Control-C o delete).
- **MemoryError:** excepción que se lanza cuando el programa ejecuta una instrucción y ésta supera el máximo de memoria disponible.
- **NameError:** excepción que se lanza cuando el nombre local o global no se encuentra.
- **NotImplementedError:** excepción que se lanza cuando un método de una clase no ha sido implementado todavía y tiene que hacerlo.
- **OSError:** excepción que se lanza cuando el sistema operativo lanza una excepción al ejecutar una instrucción. Existen las siguientes excepciones específicas que puede lanzar el sistema operativo:
- **BlockingIOError:** excepción que se lanza cuando una operación se bloquea en un objeto que no debería de bloquearse.
- **ChildProcessError:** excepción que se lanza cuando una operación de un proceso hijo devuelve un error.
- **ConnectionError:** excepción genérica que se lanza para errores relacionados a conexión.
- **BrokenPipeError:** excepción que se lanza cuando se intenta escribir en un socket o pipe (tubería) y ya ha sido cerrado.
- **ConnectionAbortedError:** excepción que se lanza cuando durante un intento de conexión ésta es abortada por el otro extremo.
- **ConnectionRefusedError:** excepción que se lanza cuando durante un intento de conexión ésta es rechazada por el otro extremo.
- **ConnectionResetError:** excepción que se lanza cuando la conexión es reseteada por el otro extremo.
- **FileExistsError:** excepción que se lanza cuando se intenta crear un fichero o directorio y éste ya existe.
- **FileNotFoundError:** excepción que se lanza cuando se intenta acceder a un fichero o directorio y no existe o no se encuentra.
- **IsADirectoryError:** excepción que se lanza cuando se intenta ejecutar una operación relacionada con ficheros sobre un directorio.
- **NotADirectoryError:** excepción que se lanza cuando se intenta ejecutar una operación relacionada con directorios sobre algo que no es un directorio.

- **PermissionError:** excepción que se lanza cuando se intenta ejecutar una operación y no se tienen los permisos suficientes.
- **ProcessLookupError:** excepción que se lanza cuando se ejecuta un proceso que no existe y se ha indicado que sí.
- **TimeoutError:** excepción que se lanza cuando se sobrepasa el tiempo de espera en alguna función del sistema.
- **OverflowError:** excepción que se lanza cuando el resultado de una operación matemática es demasiado grande para ser representado.
- **RecursionError:** excepción que se lanza cuando el número de recursividades supera el máximo permitido.
- **ReferenceError:** excepción que se lanza al intentar acceder a ciertos atributos por parte de la clase proxy y que ya se encuentran en el recolector de basura.
- **RuntimeError:** excepción que se lanza cuando el error que ocurre no puede ser categorizado en ninguno de los tipos existentes.
- **StopIteration:** excepción que se lanza cuando se intenta acceder al siguiente elemento de un iterador y ya no tiene más elementos sobre los que iterar.
- **StopAsyncIteration:** excepción que se lanza cuando se intenta acceder al siguiente elemento de un iterador asíncrono y ya no tiene más elementos sobre los que iterar.
- **SyntaxError:** excepción que se lanza cuando el analizador sintáctico encuentra un error de sintaxis.
- **IndentationError:** excepción genérica que se lanza cuando se encuentran errores de indentación del código fuente.
- **TabError:** excepción que se lanza cuando se encuentran errores de uso en las tabulaciones y espaciados del código fuente. La excepción deriva de la anterior.
- **SystemError:** excepción que se lanza cuando el intérprete de Python encuentra un error interno mientras ejecuta el programa.
- **SystemExit:** excepción que se lanza al ejecutar la instrucción sys.exit() y que provoca que se pare la ejecución del programa.
- **TypeError:** excepción que se lanza cuando una operación o función se usa con un tipo de dato incorrecto.
- **UnboundLocalError:** excepción que se lanza cuando se utiliza una variable local en una función o método y no ha sido asignado ningún valor previamente.
- **UnicodeError:** excepción que se lanza cuando se produce un error a la hora de codificar o decodificar Unicode.
- **UnicodeEncodeError:** excepción que se lanza cuando se produce un error a la hora de realizar una codificación a Unicode.
- **UnicodeDecodeError:** excepción que se lanza cuando se produce un error a la hora de realizar una decodificación de Unicode.
- **UnicodeTranslateError:** excepción que se lanza cuando se produce un error a la hora de traducir Unicode.
- **ValueError:** excepción que se lanza cuando una operación o función recibe un parámetro del tipo correcto, pero con un valor incorrecto.
- **ZeroDivisionError:** excepción que se lanza cuando se realiza una división por cero."

3. INTRODUCCIÓN A FUNCIONES

Python, es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma.

Las funciones son porciones de código que están relacionados a un nombre, ejecutadas cuando se hace referencia a ellas (se les invoca) y retornan un valor. Se definen utilizando la palabra reservada `def`; después el nombre que se le asigne a la función, paréntesis “()”, dos puntos “:” y en la siguiente línea se realiza el indentado del cuerpo y al final el uso de la palabra reservada return seguido de lo que se devolverá.

Las funciones son bloques de código reutilizables que realizan tareas específicas en un programa. Son una parte fundamental de Python y permiten organizar y modularizar el código, lo que facilita el mantenimiento y la legibilidad.

La sintaxis es la siguiente:

```
#Sintaxis de una función
#Definición de una función
def nombre_funcion(argumento1, argumento2):
    # Cuerpo de la función
    resultado = argumento1 + argumento2
    return resultado
```

Ejemplo 4: Sintaxis de una función

La función, no se ejecuta hasta que no sea invocada. Para invocar una función, se la llama por su nombre:

```
#Invocación de una función
def mi_funcion():
    print("Hola Mundo")

mi_funcion()
```

Ejemplo 5: Invocación de una función

Cuando una función retorne datos, éstos, se pueden asignar a una variable:

```
#Retorno de valor de una función
def funcion():
    return "Hola Mundo"

saludo = funcion()
print(saludo)
```

Ejemplo 6: Retorno de valor de una función

Pasar valores a una función a través de sus parámetros o lista de argumentos es algo simple y muy útil, con ello podremos hacer que la función pueda recibir valores de diferentes tipos en ocasiones diferentes para poder obtener diferentes resultados.

En algunas ocasiones debemos de fijar valores que siempre son los mismos para un argumento o grupo de argumentos, estos argumentos son conocidos como argumentos por defecto.

La ventaja de ello radica en que al hacer la invocación no es necesario colocar al parámetro por defecto haciendo más flexible la utilización de la función.

Esta ventaja en el uso de funciones es importante para lograr que una función pueda ser utilizada de formas diversas, al hacer que una función pueda ser llamada con diferente número de argumentos hacemos una función más empleable.

A continuación, detallamos algunos conceptos importantes en funciones:

Argumentos y parámetros: Los argumentos son valores que se pasan a una función cuando se llama. Los parámetros son variables que representan estos valores dentro de la función.

Valor de retorno: Las funciones pueden devolver un valor usando la palabra clave “return”. Si no se especifica un valor de retorno, la función devuelve “None” de forma predeterminada.

Funciones sin argumentos: Una función puede no tener argumentos, pero aún puede realizar tareas específicas.

Funciones con valor de retorno: Una función puede realizar cálculos y devolver un resultado que puede utilizarse en otras partes del programa.

4. ¿QUÉ ES UNA FUNCIÓN CON ARGUMENTOS?

Definimos la función **impresion()**, que muestra un texto determinado repetidas veces y enmarcado por un símbolo a elegir. Su definición es la siguiente:

```
#Función con argumentos
def impresion(veces, simbolo):
    print(simbolo*28)
    for n in range(veces):
        print(simbolo,"Ejemplo de impresion ...",simbolo)
    print(simbolo*28)
```

Ejemplo 7: Función con argumentos

Los parámetros ***veces*** y ***simbolo*** son los argumentos de la función. Ejemplos de invocación posibles podrían ser:

```
impresion(3,'*')
```

La función, para ser invocada correctamente, necesita forzosamente dos argumentos. Decimos que ***veces*** y ***simbolo*** son argumentos requeridos.

El resultado del código anterior es el siguiente:

```
****************************
* Ejemplo de impresion ... *
* Ejemplo de impresion ... *
* Ejemplo de impresion ... *
****************************
```

Imagen 4: Salida de una función con argumentos

5. FUNCIONES CON PARÁMETROS POR DEFECTO

Observa ahora esta nueva definición de ***impresion()***:

```
#Función con parámetros por defecto
def impresion(veces, simbolo='+'):
    print(simbolo*28)
    for n in range(veces):
        print(simbolo,"Ejemplo de impresion ...",simbolo)
    print(simbolo*28)

impresion(3)
```

Ejemplo 8: Función con parámetros por defecto

Fíjate en cómo está descrito el argumento ***simbolo***, especificando directamente un valor.

Decimos que ***simbolo*** es un argumento por defecto y el valor especificado es el valor que tomará por defecto en el caso que no se facilite uno concreto.

Ya no es necesario especificar el segundo argumento, si no queremos. En caso de que no lo hagamos, ***simbolo*** tomará el valor por defecto (el signo '+').

```
++++++++++++++++++++++++++++
+ Ejemplo de impresion ... +
+ Ejemplo de impresion ... +
+ Ejemplo de impresion ... +
++++++++++++++++++++++++++++
```

Imagen 5: Salida de una función con parámetros por defecto

Si no queremos usar el valor por defecto, hay que recurrir a la especificación completa cuando invocamos:

```
impresion(3,"$")
```

Con ello obtenemos el siguiente resultado:

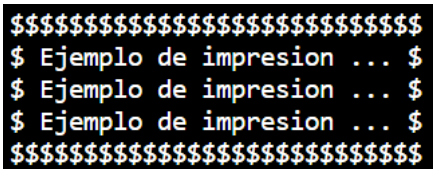


Imagen 6: Salida de una función con especificación de parámetros

Es importante tener en cuenta que los argumentos por defecto deben figurar siempre después de los requeridos, y nunca antes. Supongamos que hubiéramos querido dejar por defecto ***veces*** y requerido ***simbolo***:

```
def impresion(veces=5,simbolo):
    print(simbolo*28)
```

Obtendríamos un error sintáctico reflejando que hay argumentos que no son por defecto después de los que sí lo son. Si no fuera así Python podría liarse a la hora de saber qué argumentos son unos u otros.

Veamos un nuevo ejemplo:

```
#Función con tres parámetros, dos de ellos con valores por defecto
def mifuncion(a,b=2,c=3):
    print(a+b+c)

mifuncion(4,5,6)
mifuncion(6,4)
mifuncion(5)
```

Ejemplo 9: Función con tres parámetros, dos de ellos con valores por defecto

Toma tres parámetros y calcula la suma de ellos. Si no especificamos ***b*** ni ***c*** tomará sus valores por defecto.

En este ejemplo se ha invocado con tres, dos y un parámetro respectivamente obteniendo como resultado lo siguiente:

Imagen 7: Salida de una función tres parámetros, dos de ellos con valores por defecto

En la segunda invocación, los dos primeros argumentos corresponden a ***a*** y a ***b***. Como no hemos especificado c, toma su valor por defecto. Obteniendo como resultado el valor 13.

En la tercera invocación, hemos indicado sólo ***a***, el único que es requerido. Los demás argumentos toman sus valores por defecto. Obteniendo como resultado el valor 10.

Qué pasa si queremos llamar la función indicando el primer y último parámetro, esto solo se puede hacer si al llamar la función indicamos que dato se le pasa a cada parámetro:

```
mifuncion(a=10,c=3)
```

Lo cual da como resultado el valor 15.

6. ¿QUÉ ES UNA FUNCIÓN CON UNA CANTIDAD VARIABLE DE PARÁMETROS?

Python nos permite crear funciones que acepten un número indefinido de parámetros sin necesidad de que todos ellos aparezcan en la cabecera de la función. Los operadores * y ** son los que se utilizan para esta funcionalidad. En el primer caso, empleando el operador *, Python recoge los parámetros pasados a la función y los convierte en una tupla. De esta manera, con independencia del número de parámetros que pasamos a la función, esta solo necesita el operador * y un nombre.

Veamos un ejemplo:

```
#Función con una cantidad variable de parámetros con una invocación
def elementos(*args):
    print("Elementos: ", args)

elementos('a', 'b', 'c', 'd')
```

Ejemplo 10: Función con una cantidad variable de parámetros con una invocación

Note en este caso *args lo retorna como una tupla y es independiente a la cantidad de parámetros (no definido).

```
Elementos:  ('a', 'b', 'c', 'd')
```

Imagen 8: Salida de una función con una cantidad variable de parámetros con una invocación

Analicemos otro ejemplo:

```
#Función con una cantidad variable de parámetros con n invocaciones
def myFunction(*items):
    for item in items:
        print(item)
print("Primera invocacion: ") myFunction()
print("Segunda invocacion: ")
myFunction(1, 2, 3)
print("Tercera invocacion: ")
myFunction('a', 'b', 'c', 'd', 2, 3)
tupla = ('a', 1, 'b', 2, 'c', 3)
print("Cuarta invocacion: ") myFunction(tupla)
```

Ejemplo 11: Función con una cantidad variable de parámetros con n invocaciones

En el código anterior, se puede observar el comportamiento de la función siempre es el mismo, con independencia de la cantidad de número de argumentos que se le envía a la función al ser invocada.

Los resultados que se obtienen son los siguientes:

```
Primera invocacion:
Segunda invocacion:
1
2
3
Tercera invocacion:
a
b
c
d
2
3
Cuarta invocacion:
('a', 1, 'b', 2, 'c', 3)
```

Imagen 9: Salida de una Función con una cantidad variable de parámetros con n invocaciones

En un segundo caso, se dispone del operador **, con el cual es posible enviar argumentos especificando un identificador (***nombre***) para cada uno de los argumentos. Estructuralmente, el Lenguaje Python construye un diccionario con los parámetros enviados a la función para luego ser tratados de esta forma.

Veamos el siguiente ejemplo:

```
#Función con una cantidad variable de parám. Tupla diccionario
def elementos(*args, **kwargs):
    print("Argumentos posicionales:", args)
    print("Argumentos con nombre:", kwargs)

elementos(valor1='a', valor2='b', valor3='c', valor4='d')
elementos('a', 'b', 'c', 'd')
```

Ejemplo 12: Función con una cantidad variable de parámetros – tupla y diccionario

Obtenemos el siguiente resultado, de acuerdo al argumento:

```
Argumentos posicionales: ()
Argumentos con nombre: {'valor1': 'a', 'valor2': 'b', 'valor3': 'c', 'valor4': 'd'}
Argumentos posicionales: ('a', 'b', 'c', 'd')
Argumentos con nombre: {}
```

Imagen 10: Salida de una Función con una cantidad variable de parámetros – tupla y diccionario

Ahora veamos otro ejemplo:

```
#Función con una cantidad variable de parámetros – diccionario
def miFuncion(**argumentos): print(argumentos)
miFuncion(x = 1, y = 3)
miFuncion(x = 'a', y = 'b', z = 'c')
```

Ejemplo 13: Función con una cantidad variable de parámetros - diccionario

En este caso nos muestra argumentos con nombre:

```
{'x': 1, 'y': 3}
{'x': 'a', 'y': 'b', 'z': 'c'}
```

Imagen 11: Salida de una Función con una cantidad variable de parámetros - diccionario

También es posible que la cabecera de una función utilice uno o varios argumentos posicionales, seguidos del operador * o **. Esto nos da bastante flexibilidad a la hora de invocar a una función. Por ejemplo:

```
#Función con una cantidad variable de parámetros y otros
def myFunction(nombre, edad, **datos):
    print("Nombre: ", nombre)
    print("Edad: ", edad)
    for clave, valor in datos.items():
        print(clave, valor)

myFunction("Cristina", 24, Pais = "Estados Unidos")
myFunction("Carlos Santana", 25, Celular = 123456789, Pais
= "México")
myFunction("Carlos Santana", 25, Celular = 123456789, Pais
= "México", Ciudad ='Rivera Maya')
```

Ejemplo 14: Función con una cantidad variable de parámetros y otros

En este caso obtenemos el siguiente resultado:

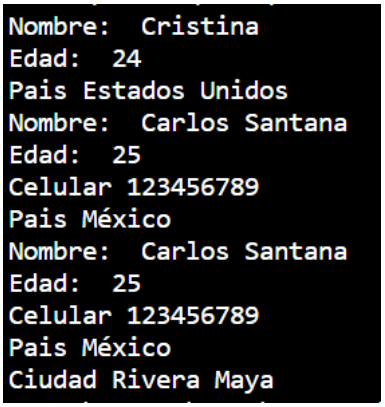


Imagen 12: Salida de una Función con una cantidad variable de parámetros y otros

7. FUNCIONES CON UNA CANTIDAD VARIABLE DE PARÁMETROS

Otra posibilidad en la declaración de una función en Python es la definición de una cantidad variable de parámetros.

Para definir una cantidad variante de parámetros debemos antecederle el carácter asterisco (*) al último parámetro de la función.

Supongamos que necesitemos implementar una función que le enviemos una serie de enteros y nos retorne la suma de todos ellos (como mínimo le enviamos 2 y no hay un máximo de valores). Lo que en realidad el lenguaje Python hace es una tupla con todos los valores de los parámetros a partir del tercero.

```
#Función suma con una cantidad variable de parámetros y otros
def sumar(x1,x2,*xn):
    s=x1+x2
    for valor in xn:
        s=s+valor
    return s

print(sumar(1, 2))
print(sumar(1,2,3,4))
print(sumar(1,2,3,4,5,6,7,8,,10))
```

Ejemplo 15: Función suma con una cantidad variable de parámetros y otros

Luego nuestro algoritmo debe recorrer la tupla para procesar los elementos propiamente dichos, en nuestro caso con un ***for in*** y los sumamos junto al primer y segundo parámetro.

Luego cuando hacemos la llamada a la función:

```
print sumar(1,2)
```

Si pasamos solo dos parámetros la tupla se inicializa vacía, por lo que el for in no ejecuta el bloque contenido.

Si llamamos la función con 10 parámetros:

```
print
    sumar(1,2,3,4,5,6,7,8,
    9,10)
```

Luego la tupla se crea con 7 elementos.

IV

ACTIVIDADES

I. Configuración VSCode

1. Descargue el aplicativo del siguiente URL: https://code.visualstudio.com/
2. Instalar Python desde el siguiente URL: https://www.python.org/.
3. Instalar la extensión Python en VSCode:

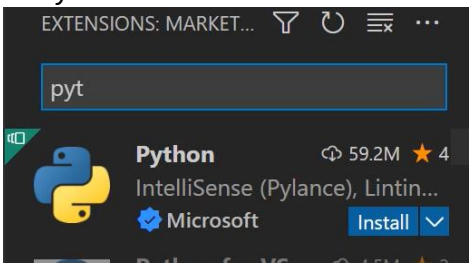
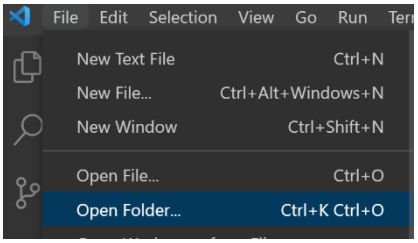


Imagen 13: Extensión Python de Microsoft

4. Instalar la extensión. run en VSCode.

Imagen 14: Extensión Code Runner

5. Crear una carpeta Lenguaje de Programación I desde el explorador. Luego seleccione dicha carpeta desde VSCode.



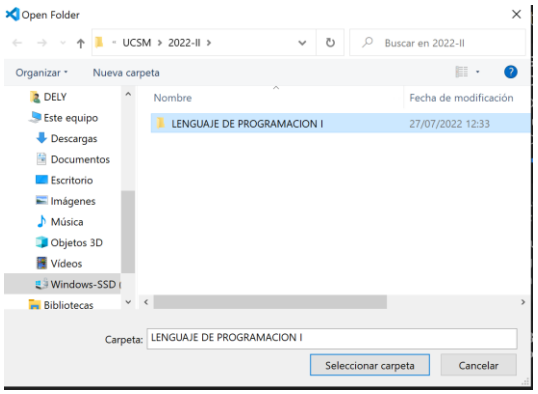


Imagen 15: Selección de carpeta en VSCode

6. Activar la opción Auto Save.

Imagen 16: Opción autoguardado

II. Funciones de Python

1. En el editor de Python escriba el siguiente código.

```
def saludo():
    print("Hola, esta es mi primera función")

saludo()
```

2. Guárdelo con el nombre de actividad1.py. Ejecute el programa con la opción Run.

Parámetros

3. En el editor de Python escriba el siguiente código.

```
def saludo_nombre(nombre):
    print(f"Hola, mi nombre es {nombre}.")

saludo_nombre("EPIS")
```

4. Guárdelo con el nombre de actividad2.py. Ejecute el programa con la opción Run.

Parámetros por omisión

5. En el editor de Python escriba el siguiente código.

```
def saludo_nombre(nombre,mensaje="Bienvenido al curso LP I"):
    print(f"{mensaje} estudiante {nombre}.")

saludo_nombre("EPIS")
saludo_nombre("María","Bienvenido Semestre Impar EPIS")
```

6. Guárdelo con el nombre de actividad3.py. Ejecute el programa con la opción Run.

Keywords con parámetros

7. En el editor de Python escriba el siguiente código.

```
def saludo(nombre, mensaje="Hola"):
    print (mensaje, nombre)

saludo(mensaje="Buen día", nombre="Juan")
```

8. Guárdelo con el nombre de actividad4.py. Ejecute el programa con la opción Run.

Parámetros arbitrarios

Para definir argumentos arbitrarios en una función, se antecede al parámetro un asterisco (*).

9. En el editor de Python escriba el siguiente código:

```
def suma(*args):
    total = 0
    for n in args:
        total += n
    return total

print(suma(2, 3, 5))
```

10. Guárdelo con el nombre de actividad5.py. Ejecute el programa con la opción Run.

Parámetros arbitrarios como pares de clave = valor

En estos casos, al nombre del parámetro deben precederlo dos asteriscos (**).

11. En el editor de Python escriba el siguiente código:

```
def suma(**kwargs):
    total = 0
    for value in kwargs.values():
        total += value
    return total

print(suma(a = 1, b = 3, c = 2))
```

12. Guárdelo con el nombre de actividad6.py. Ejecute el programa con la opción Run.

Desempaquetado de Parámetros

Puede ocurrir, además, una situación inversa a la anterior. Es decir, que la función espere una lista fija de parámetros, pero que éstos, en vez de estar disponibles de forma separada, se encuentren contenidos en una lista o tupla. En este caso, el signo asterisco (*) deberá preceder al nombre de la lista o tupla que es pasada como parámetro durante la llamada a la función.

13. En el editor de Python escriba el siguiente código:

```
def calcular(importe, descuento):
    return importe - (importe * descuento / 100)

datos = [1500, 10]
print(calcular(*datos))
```

14. Guárdelo con el nombre de actividad7.py. Ejecute el programa con la opción Run.

Llamadas de retorno

Es posible, llamar a una función dentro de otra, de forma fija y de la misma manera que se le llamaría, desde fuera de dicha función:

15. En el editor de Python escriba el siguiente código:

```
def funcion():
    return "Hola Mundo"

def llamada_de_retorno(func=""):
    """Llamada de retorno a nivel global"""
    return globals()[func]()

print (llamada_de_retorno("funcion"))

# Llamada de retorno a nivel local
nombre_de_la_funcion = "funcion"
print(locals()[nombre_de_la_funcion]())
```

16. Guárdelo con el nombre de actividad8.py. Ejecute el programa con la opción Run.

Llamadas recursivas

Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

17. En el editor de Python escriba el siguiente código:

```
def cuenta_regresiva(numero):
    numero -= 1
    if numero > 0:
        print(numero)
        cuenta_regresiva(numero)
    else:
        print("...........!")
    print("Fin de la función", numero)

cuenta_regresiva(4)
```

18. Guárdelo con el nombre de actividad9.py. Ejecute el programa con la opción Run.

V

EJERCICIOS

(La práctica tiene una duración de 4 horas)

Haciendo uso del lenguaje Python desarrollar los siguientes ejercicios, use excepciones donde lo sea posible, nombrarlos como: propuesto1.py, propuesto2.py, …, propueston.py.

1. Crear una excepción para la validación de una división entre cero y la validación en el ingreso permitido de soló valores numéricos.
2. Modificar la función impresión de la página 4 para que el texto a enmarcarse sea elegible por el usuario como un argumento adicional. Utilice la función ***len()***, que determina la longitud de una cadena y deje un espacio entre el texto y el símbolo para ambos lados.
3. Crear una función que permita obtener el valor de x en función de dos parámetros a y b.

   $x=a^2+2ab+b^2$
4. Crear una función que permita obtener el valor de raíz cuadrada (z) de x^2+y^2.
5. Crear una función que calcule el IMC de acuerdo al peso y la altura. IMC = peso/ altura^2.
6. Determine a través de una función si un numero entero ingresado es par o impar.
7. Calcule la suma de los n números utilizando una función de argumentos arbitrarios.
8. Elaborar una función que reciba n números enteros y nos retorne su promedio.
9. Realiza una función llamada relacion(a, b) que a partir de dos números cumpla lo siguiente:

   Si el primer número es mayor que el segundo, debe devolver 1.

   Si el primer número es menor que el segundo, debe devolver -1.

   Si ambos números son iguales, debe devolver un 0.
10. Desarrollar una función que nos retorne el perímetro de un rectángulo pasando como parámetro la base y la altura del rectángulo.
11. Crear una función que permita el ingreso del radio de un círculo y calcule el área.
12. Programar una función que valide un número celular ingresado (debe ser numérico y con 9 dígitos).
13. Escribir un programa que permita al usuario obtener un identificador para cada uno de los socios de un club (código). Para eso ingresará nombre completo y número de DNI de cada socio, indicando que finalizará el procesamiento mediante el ingreso de un nombre vacío.

    El formato del nombre de los socios será: nombre apellido la primera letra en mayúscula. Se debe validar que el número de DNI tenga 8 dígitos. En caso contrario, el programa debe dejar al usuario en un bucle hasta que ingrese un DNI correcto. Se recomienda hacer uso de un diccionario.

    Ejemplo:

    Código: 1

    Nombre: María Linares

    DNI: 25834910
14. Crear una función que realice la suma de cinco números utilizando solo *args. Debe imprimir el resultado en la consola. La suma no se realizará directamente sobre el print().
15. Definir por asignación una lista de enteros en el bloque principal del programa. Elaborar tres funciones, la primera recibe la lista y retorna la suma de todos sus elementos, la segunda recibe la lista retornando el mayor valor y la última recibe la lista retornando el menor valor.
16. Programar una función que permita determinar si un número es primo o no.
17. Desarrollar la función recursiva Fibonacci.

18. Crear una función recursiva para el cálculo del factorial de un determinado número.
19. Haciendo uso de argumentos arbitrarios cree una función que concatene todos los argumentos arbitrarios con el separador que se le envíe como parámetro.
20. Haciendo uso del desempaquetado de parámetros hallar el mayor y el menor de 5 números.
21. Realizar una función que permita la carga de n alumnos. Por cada alumno se deberá preguntar el nombre completo (nombres y apellidos) y permita el ingreso de 3 notas. Las notas deben estar comprendidas entre 0 y 20. Devolver el listado de alumnos.
22. Desarrollar un programa que permita adivinar el número en el rango de 0 a 99, para lo cual debe crear la función adivinar, la consola debe mostrar mensajes: demasiado pequeño, muy grande o ha ganado cuando adivina el número.
23. Al ejercicio anterior añada los parámetros mínimo y máximo para controlar el número a adivinar. Utilice las funciones que considere.
24. Programe una función que genere la tabla de multiplicar del 1 al 10 de un determinado número.
25. Desarrollar una función que permita sumar todos los dígitos de un número.

VI

CUESTIONARIO

1. Desarrollar los ejercicios del codingbat - http://codingbat.com/python
2. ¿Qué es una función en Python?
3. ¿Cuál es la sintaxis básica para definir una función en Python?
4. ¿Qué es el valor de retorno de una función?
5. ¿Cómo se llama el valor que se pasa a una función cuando se la llama?
6. ¿Cuál es la diferencia entre parámetros y argumentos de una función?
7. ¿Qué es el ámbito (scope) de una variable en Python?
8. ¿Qué es una función lambda y cuál es su uso común?
9. ¿Cómo se pueden documentar funciones en Python?
10. ¿Qué es la recursión en programación y cómo se utiliza en funciones?

VII

BIBLIOGRAFIA Y REFERENCIAS

[1] Luz Nolasco Valenzuela, Jorge Nolasco Valenzuela, Javier Gamboa Cruzado, Fundamentos de Programación con Python 3, Lima: Empresa Editora Macro EIRL, 2020.

[2] S. CHAZALLET, Python 3 Los fundamentos del lenguaje (3ª edición), Éditions ENI, 2020.

[3] Moreno, A. Córcoles, Python Práctico, RAMA, 2019.