UNIVERSIDAD DE CARABOBO
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE COMPUTACIÓN
CÁTEDRA DE COMPUTACIÓN I

Fundamentos de Programación

Con aplicaciones en la Ingeniería

MSc. Ing. Alejandro Bolívar

2023

FUNDAMENTOS DE PROGRAMACIÓN

Con aplicaciones en la Ingeniería

Libro de estudio de la asignatura Computación I.

Tabla de contenido

Introducción

Este libro nace con la intención de proporcionar a los estudiantes del tercer semestre de la Facultad de Ingeniería de la Universidad de Carabobo, en la asignatura Computación I del Departamento de Computación una comprensión sólida y completa de los conceptos fundamentales de la programación enfocado en la resolución de problemas mediante el computador. Partiendo desde la lógica de programación, el estudiante aprenderá a desarrollar algoritmos eficientes y efectivos para resolver problemas complejos en un lenguaje de programación. Además, se abordan temas esenciales como el control de flujo, las estructuras de datos, el diseño modular, y la depuración de código.

Este texto representa una guía detallada y accesible para desarrollar habilidades en una de las disciplinas más relevantes e influyentes de nuestro tiempo. Esperamos que este libro proporcione la base necesaria para avanzar en su formación como programador, así como también permitirle desarrollar soluciones tecnológicas innovadoras e impactantes.

Unidad 7
Programación Modular

La creación de segmentos de código divide el problema en segmentos lo más pequeño posible con el fin de resolver un problema en su totalidad. Esto facilita la revisión y expansión del código además de aprovechar su reutilización y evitar la repetición de instrucciones.

La programación modular es un paradigma de programación basado en utilizar *funciones* o subrutinas, y únicamente tres estructuras de control:

- secuencial: ejecución de una sentencia tras otra.
- selección o condicional: ejecución de una sentencia o conjunto de sentencias, según el valor de una variable booleana.
- iteración (ciclo o bucle): ejecución de una sentencia o conjunto de sentencias, mientras una variable booleana sea verdadera.

Este paradigma se fundamenta en el teorema correspondiente, que establece que toda función computable puede ser implementada en un lenguaje de programación que combine sólo estas tres estructuras lógicas o de control.

La estructura de secuencia es la que se da naturalmente en el lenguaje, ya que por defecto las sentencias son ejecutadas en el orden en que aparecen escritas en el programa.

Para las estructuras condicionales o de selección, Python dispone de la sentencia *if*, que puede combinarse con sentencias *elif* y/o *else*.

Para los ciclos o iteraciones existen las estructuras *while* y *for*.

1.1 Ventajas de la programación modular.

Entre las ventajas de la programación modular, cabe citar las siguientes:

- Los programas son más fáciles de entender, pueden ser leídos de forma secuencial y no hay necesidad de tener que rastrear saltos de líneas (GOTO) dentro de los bloques de código para intentar entender la lógica interna.
- La estructura de los programas es clara, puesto que las sentencias están más ligadas o relacionadas entre sí.
- Se optimiza el esfuerzo en las fases de pruebas y depuración. El seguimiento de los fallos o errores del programa (*debugging*), y con él su detección y corrección, se facilita enormemente.
- Se reducen los costos de mantenimiento. Análogamente a la depuración, durante la fase de mantenimiento, modificar o extender los programas resulta más fácil.

- Los programas son más sencillos y más rápidos de confeccionar.
- Se incrementa el rendimiento de los programadores.

1.2 Funciones

En capítulos anteriores hemos aprendido a utilizar funciones. Algunas de ellas están predefinidas (abs, round, etc.) mientras que otras deben importarse de módulos antes de poder ser usadas (por ejemplo, *sin* y *cos* se importan del módulo math). En la Tabla 1-1 se presenta algunas de estas funciones.

Tabla 1-1. Funciones incorporadas en Python

Función interna	Devuelve
`type`	tipo de dato
`id`	Identidad o ubicación de memoria
`bin`	string del binario equivalente al entero dado
`int, float, str`	entero, real, string del valor dado
`input`	string del texto leído del teclado
`print`	valores a imprimir en pantalla
`abs`	valor absoluto de un número
`round`	redondea un real a los decimales especificados

Módulo math	
`pi`	valor de π (no es función)
`sqrt`	raíz cuadrada de un número

Módulo random	
`randint`	número entero aleatorio en el rango dado

La lista de funciones internas de la versión 3 de Python se puede consultar en la web de *Python Software foundation* (https://docs.python.org/3/library/functions.html).

En este tema aprenderemos a definir nuestras propias funciones. Definiendo nuevas funciones escribiremos instrucciones en Python para hacer cálculos que inicialmente no están incluidos en el lenguaje y, en cierto modo, adaptando el lenguaje de programación al tipo de problema a resolver, enriqueciéndolo para que el programador pueda ejecutar acciones complejas de un modo sencillo: llamando a funciones desde su programa.

Ya el lector ha utilizado módulos, es decir, archivos que contienen funciones y variables de valor predefinido que puede importar a los programas. En este capítulo aprenderemos a crear nuestros propios módulos, de manera que reutilizar nuestras funciones en varios programas resultará extremadamente sencillo: bastará con importarlas.

Suponga que necesita calcular la suma de números enteros de 1 a 10, de 20 a 37 y de 35 a 49. Si crea un programa para agregar estos tres conjuntos de números, su código podría verse así:

```
sum = 0
for i in range(1, 11):
    sum += i
print("La suma desde 1 hasta 10 es ", sum)

sum = 0
for i in range(20, 38):
    sum += i
print("La suma desde 20 hasta  37 es ", sum)

sum = 0
for i in range(35, 50):
    sum += i
print("La suma desde 35 hasta  49 es ", sum)
```

Observe que el código para calcular estas sumas es muy similar, excepto que los números enteros de inicio y final de la suma son diferentes. Sería bueno poder escribir el código común y luego reutilizarlo. Puede hacer esto definiendo una función, que le permite crear un código reutilizable. Por ejemplo, el código anterior se puede simplificar mediante el uso de funciones, de la siguiente manera:

```
def sum(i1, i2):
    result = 0
    for i in range(i1, i2 + 1):
        result += i
    return result


def main():
    print("La suma desde 1 hasta  10 es ", sum(1, 10))
    print("La suma desde 20 hasta  37 es ", sum(20, 37))
    print("La suma desde 35 hasta  49 es ", sum(35, 49))

main()  # llamada de la función principal
```

1.3 Funciones definidas por el usuario.

Una función, es la forma de agrupar expresiones y sentencias que realicen determinadas acciones, pero que éstas, solo se ejecuten cuando son llamadas. Es decir que, al escribir las instrucciones dentro de una función, al ejecutar el programa, las instrucciones contenidas en la función no serán ejecutadas si no se ha hecho una referencia a la función que lo contiene.

1.3.1 Definiendo funciones

En Python, la definición de funciones se realiza mediante la instrucción ***def*** más un nombre de función descriptivo -para el cuál, aplican las mismas reglas que para el nombre de las variables- seguido de paréntesis de apertura y cierre entre los cuales se escriben los parámetros que recibe la función. Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (:) y el código que la conforma, se escribe con sangría de 4 espacios:

```
def nombre_de_la_función(param1, param2, ….):
    ''' DOCSTRING '''
    # aquí se escribe el código de la función
    return valor_devuelto  # opcional
```

Por ejemplo, la siguiente función convierte el valor ingresado de pulgadas a centímetros:

```
def pulg_cm(pulg):
    ''' Función que realiza la conversión de pulgadas a centímetros '''
    cm = pulg * 2.54
    return cm
```

1.3.2 Usando funciones

El modo general es:

```
>>> Nombre(parámetro)
```

Ejemplos:

Usando la función *pulg_cm* (función que convierte de pulgadas a centímetros):

```
>>> print(pulg_cm(5))
```

```
12.7
```

Todas las funciones devuelven un valor. Las que aparentemente no devuelven nada, están devolviendo *None*:

```
def guarda_lista(lista, nombre):
    ''' Función para guardar una lista de datos en un archivo (nombre) '''
    fh = open(nombre,'w')

    for x in lista:
        fh.write('%s\n'%x)

    fh.close()
    return None
```

Uso de la función:

```
>>> guarda_lista([1,2,3],'algo.txt')
```

Note que retorna la función anterior.

Los valores deberían retornarse solo vía *"return"*, para cumplir con la llamada "integridad referencial" (esto es, que una función no modifique el resto del programa). Python no obliga al programador a cumplir con dicha propiedad ya que es posible alterar un dato mutable dentro de una función y esta modificación puede ser vista desde fuera de la misma.

Cuando la función, haga un **retorno de datos**, éstos, pueden ser asignados a una variable:

```
def función():
    return 'Hola Mundo'

frase = función()
print (frase)
```

1.3.3 Ámbito de una función

Los valores declarados en una función son reconocidos solamente dentro de la función. Cuando un valor no es encontrado en el ámbito donde se la invoca, se busca en el ámbito inmediatamente anterior.

Se puede usar **global** para declarar variables globales dentro de funciones, aunque **su uso no es recomendable**. Estas variables globales cambian el contenido de las variables del módulo que contiene a la función.

```
def test():
    z = 10
    print ('Valor de z: %s' % z)
    return None

def test2():
    global z
    z = 10
    print ('Valor de: %s' % z)
    return None
```

Probando el ámbito de las variables:

```
>>> z = 50
>>> test()  # se hace el llamado de la primera función
Valor de z: 10
>>> z
50
>>> test2()  # se hace el llamado de la segunda función
Valor de z: 10
>>> z
10
```

La función *test* no modifica el valor externo de la variable z, originalmente 50, la segunda función *test2* si modifica el valor de *z* ya que cambia de 50 a 10.

1.4 Los parámetros

Un parámetro es un valor que la función espera recibir cuando sea llamada (invocada), a fin de ejecutar acciones en base al mismo. Una función puede esperar uno o más parámetros (que irán separados por una coma) o ninguno [1].

```
def función(par1, par2, ...):
    # código de la función
```

Los parámetros, **se indican** entre los paréntesis, **a modo de variables**, a fin de poder utilizarlos como tales, dentro de la misma función. Los parámetros que una función espera, serán utilizados por ésta, dentro de su algoritmo, a modo de **variables de ámbito local**. Es decir, que los parámetros serán variables locales, a las cuáles solo la función podrá acceder:

```
def mi_función(nombre, apellido):
    nombre_completo = nombre, apellido
    print(nombre_completo)
```

Si quisiéramos acceder a esas variables locales, fuera de la función, se obtiene un error:

```
def mi_función(nombre, apellido):
    nombre_completo = nombre, apellido
```

```
    print(nombre_completo)

print(nombre)  # Retornará el error: NameError: name 'nombre' is not defined
```

Al llamar a una función, siempre se le deben pasar sus argumentos en el mismo orden en el que los espera. Pero esto puede evitarse, haciendo uso del paso de argumentos como *keywords* (ver más abajo: "Keywords como parámetros").

1.4.1 Parámetros por omisión

En Python, también es posible, asignar valores por defecto a los parámetros de las funciones. Esto significa, que la función podrá ser llamada con menos argumentos de los que espera:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre

saludar('Mundo') # Imprime: Hola Mundo
```

PEP 8: Funciones A la definición de una función la deben anteceder dos líneas en blanco.

Al asignar parámetros por omisión, no debe dejarse espacios en blanco ni antes ni después del signo `=`.

1.4.2 Keywords como parámetros

En Python, también es posible llamar a una función, pasándole los argumentos esperados, como pares de `claves=valor`:

```
def saludar(nombre, mensaje='Hola'):
    print mensaje, nombre

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

En este caso no importa el orden de los parámetros durante el llamado de la función.

1.4.3 Parámetros arbitrarios

Al igual que en otros lenguajes de alto nivel, es posible que una función, espere recibir un número arbitrario -desconocido- de argumentos. Estos argumentos, llegarán a la función en forma de ***tupla***.

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

```
def recorrer_parámetros_arbitrarios(parámetro_fijo, *arbitrarios):
    print (parámetro_fijo)

    # Los parámetros arbitrarios se corren como tuplas
    for argumento in arbitrarios:
        print (argumento)

recorrer_parámetros_arbitrarios('Fixed', 'arbitrario 1', 'arbitrario
2', 'arbitrario 3')
```

Por ejemplo,

```
>>> def menú(*platos):
...     print('Hoy tenemos: ', end='')
...     for plato in platos:
...         print(plato, end=', ')
...     return
...
>>> menú('pasta', 'pizza', 'ensalada')
Hoy tenemos: pasta, pizza, ensalada,
```

Si una función espera recibir parámetros fijos y arbitrarios, **los arbitrarios siempre deben suceder a los fijos**.

Es posible también, enviar parámetros arbitrarios como pares de clave=valor. En estos casos, al nombre del parámetro deben precederlo dos asteriscos (**):

```
def recorrer_parametros_arbitrarios(parametro_fijo, *arbitrarios,
**kwords):
    print (parametro_fijo)
    for argumento in arbitrarios:
              print (argumento)

          # Los argumentos arbitrarios tipo clave, se recorren como
          # los diccionarios
          for clave in kwords:
                print ('El valor de ', clave, ' es ', kwords[clave])

          recorrer_parametros_arbitrarios('Fixed', 'arbitrario 1',
'arbitrario 2', 'arbitrario 3', clave1='valor uno', clave2='valor
dos')
```

Por ejemplo,

```
>>> def contacto(**info):
...     print('Datos del contacto)
...     for clave, valor in info.items():
...         print(clave, ":", valor)
...     return
...
>>> contacto(Nombre = "Alf", Email = "asalber@ceu.es")
Datos del contacto
```

```
Nombre : Alf
Email : asalber@ceu.es
>>> contacto(Nombre = "Alf", Email = "asalber@ceu.es", Dirección =
"Madrid")
Datos del contacto
Nombre : Alf
Email : asalber@ceu.es
Dirección : Madrid
```

1.4.4 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:

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

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

El mismo caso puede darse cuando los valores a ser pasados como parámetros a una función, se encuentren disponibles en un diccionario. Aquí, deberán pasarse a la función, precedidos de dos asteriscos (**):

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

datos = {'descuento': 10, 'importe': 1500}
print (calcular(**datos))
```

1.4.5 Parámetros por Valor o por Referencia

Un último factor a tener en cuenta en los parámetros a funciones en cualquier lenguaje de programación es su modificación en tiempo de ejecución. Tradicionalmente los pasos por parámetro funcionan por valor o por referencia. En el primer caso los argumentos a una función no se modifican al salir de ella (en caso de que el código interno los altere). Técnicamente, se debe al hecho de que realmente no pasamos a la función la variable en cuestión, sino una copia local a la función que es eliminada al acabar su ejecución. En el caso del paso por referencia se pasa a la función un puntero al objeto (en el caso de lenguaje C) o simplemente una referencia (en lenguajes de alto nivel) que permite su indirección. De este

modo, las modificaciones a los parámetros que se hacen dentro de la función se ven reflejadas en el exterior una vez la función termina. En Python los pasos por parámetros son en general por referencia, la excepción la conforman los tipos de datos básicos o inmutables (enteros, flotantes, ..., tuplas), que se pasan por valor. A continuación, se muestra un ejemplo de este hecho (Ejemplo 1-1).

Ejemplo 1-1. Ejemplo de paso por valor y referencia en Python

```
# Ejemplos de definiciones y llamadas de funciones
# función para verificar la modificabilidad de los parámetros

def persistenciaParametros(parametro1, parametro2):
    parametro1 = parametro1 + 5
    parametro2[1] = 6
    print (parametro1)
    print (parametro2)

numeros = [1 ,2 ,3 ,4]
valor_inmutable = 2
persistenciaParametros(valor_inmutable, numeros)
print(números)
print(valor_inmutable)
```

Como se puede observar a partir de la ejecución del ejemplo, la lista *numeros* es modificada en el código de la función, mientras que el valor entero de la variable *valor_inmutable* modificado dentro de la función, pierde esta modificación al devolver el control al programa principal.

1.5 Sobre la finalidad de las funciones

Una función, puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos y, utilizar cualquiera de las características vistas hasta ahora. No obstante, **una buena práctica, indica que la finalidad de una función, debe ser realizar una única acción, reutilizable y, por lo tanto, tan genérica como sea posible.**

1.6 Consideraciones prácticas.

Uso de programa principal o no en los códigos siguientes los resultados son idénticos:

```
def suma(a, b):
    return a+b

res = suma(3, 5)
print(res)
```

```
def suma(a, b):
    return a+b

if __name__ == "__main__":
    res = suma(3, 5)
    print(res)
```

Si se guarda la función en un archivo (por ejemplo suma.py) ***el código a la derecha permite importar el archivo suma.py como un módulo (sin que se ejecute el programa principal)***

1.7 Documentación de funciones

Una práctica muy recomendable cuando se define una función es describir lo que la función hace en un comentario.

En Python esto se hace con un **docstring** que es un tipo de comentario especial se hace en la línea siguiente al encabezado de la función entre tres comillas simples ''' o dobles """.

Después se puede acceder a la documentación de la función con la función `help(<nombre-función>)`.

```
>>> def area_triangulo(base, altura):
...     """Función que calcula el área de un triángulo.
...
...     Parámetros:
...         - base: Un número real con la base del triángulo.
...         - altura: Un número real con la altura del triángulo.
...     Salida:
...         Un número real con el área del triángulo de base y altura
especificadas.
...     """
...
...     return base * altura / 2
...
>>> help(area_triangulo)
area_triangulo(base, altura)
    Función que calcula el área de un triángulo.

    Parámetros:
        - base: Un número real con la base del triángulo.
        - altura: Un número real con la altura del triángulo.
    Salida:
        Un número real con el área del triángulo de base y altura
especificadas.
```

1.8 Uso de DOCTEST para probar funciones.

Existe en Python el módulo *doctest* que incluye la función *testmod*. Esta función busca fragmentos de texto en el *docstring* como las sesiones interactivas de Python, y luego ejecuta esas sesiones para verificar que funcionan exactamente como se muestra [2]. Por

ejemplo, la función *ganancia_dB()* se puede ejecutar varias veces en la consola de Python, o se puede prever qué valores debe dar:

```
>>> ganancia_dB(1,1000)
60.0
>>> ganancia_dB(10,1000)
40.0
>>> ganancia_dB(1,1)
0.0
>>> ganancia_dB(10,1)
-20.0
```

La función incluye el string de prueba (*doctest*) en el *docstring* y la llamada a la función **testmod**.

```
from math import log10

def ganancia_dB(x, y):
    """Calcula ganancia en dB: 20log(y/x)
    >>> ganancia_dB(1,1000)
    60.0
    >>> ganancia_dB(10,1000)
    40.0
    >>> ganancia_dB(1,1)
    0.0
    >>> ganancia_dB(10,1)
    -20.0
    """
    return 20*log10(y/x)


import doctest
doctest.testmod(verbose=True)
```

Ejecutando el programa se escribe el resultado de evaluar la función con la sesión interactiva incluida en el *docstring*:

```
Trying:
ganancia_dB(1,1000)
Expecting:
60.0
ok
Trying:
ganancia_dB(10,1000)
Expecting:
40.0
ok
Trying:
ganancia_dB(1,1)
Expecting:
0.0
ok
Trying:
ganancia_dB(10,1)
Expecting:
-20.0
ok
1 items had no tests:
```

```
__main__
1 items passed all tests:
4 tests in __main__.ganancia_dB
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```

Ejemplo de función con error en el diseño (- en lugar de +):

```
from math import log10


def ganancia_dB(x, y):
    """Calcula ganancia en dB: 20log(y/x)
    >>> ganancia_dB(1,1000)
    60.0
    >>> ganancia_dB(10,1000)
    40.0
    >>> ganancia_dB(1,1)
    0.0
    >>> ganancia_dB(10,1)
    -20.0
    """
    return 20*log10(y/x)


import doctest
doctest.testmod(verbose=True)

-1
**********************************************************************
**
File "J:/pruebaDoctest2.py", line 12, in __main__.suma
Failed example:
suma(1, 2)
Expected:
3
Got:
-1
**********************************************************************
**
File "J:/pruebaDoctest2.py", line 14, in __main__.suma
Failed example:
suma(10, 10)
Expected:
20
Got:
0
**********************************************************************
**
1 items had failures:
2 of 2 in __main__.suma
***Test Failed*** 2 failures.
>>>
```

1.9 Recomendaciones de Escritura del Código.

1. Al inicio de cada función, agregar brevemente un comentario que explique el comportamiento general de la función.
2. Definir el nombre de funciones de acuerdo a la acción que realiza, por lo general es un verbo.
3. Definir los nombres de las variables y constantes locales de acuerdo al contenido que almacenarán.
4. Comentar cuando sea justo y necesario, usar los comentarios dentro de las funciones para describir las variables (sólo cuando su utilidad sea potencialmente dudosa) y cuando existan bloques de código difíciles de entender a primera vista; el exceso de comentarios vuelve ilegible el código.
5. Definir variables locales al inicio de la implementación de cada función, como un bloque de código bien separado del bloque que contenga las instrucciones ejecutables, esta separación puede consistir en una línea en blanco, o bien un comentario que denote la utilidad de cada bloque.
6. Las secciones deseadas en el programa son:
    1. Encabezado con datos del equipo, colegio y breve descripción del programa (no más de 5 líneas), versionado del programa.
    2. Definición de bibliotecas utilizadas
    3. Declaración de constantes
    4. Declaración de variables globales
    5. Declaración de funciones
    6. Declaración bloque de inicialización si existiera
    7. Declaración de bloque principal
    8. Cierre del programa al salir del bloque principal si existiera

1.10 Ejercicios Resueltos

Ejercicio 1-1: Cálculo del IMC.

```
def imc():

    print("CÁLCULO DEL ÍNDICE DE MASA CORPORAL (IMC)")
    peso = float(input("¿Cuánto kg pesa? "))
    altura = float(input("¿Cuánto mide en metros? "))
    imc = peso / altura**2
    print(f"Su IMC es {round(imc, 1)}")
    print(
        "Un IMC muy alto indica obesidad. Los valores normales de imc están
entre 20 y 25,"
    )
    print(
        "pero esos límites dependen de la edad, del sexo, de la constitución
física, etc."
    )
```

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

Ejercicio 1-2: Calculadora básica.

```
      def add(x,y):
    return x+y


def subtract(x,y):
    return x-y


def multiply(x,y):
    return x*y


def divide(x,y):
    return x/y


x = 8
y = 4

print ('%d + %d = %d' % (x, y, add(x, y)))
print ('%d - %d = %d' % (x, y, subtract(x, y)))
print ('%d * %d = %d' % (x, y, multiply(x, y)))
print ('%d / %d = %d' % (x, y, divide(x, y)))
```

Ejercicio 1-3: Operaciones con Fracciones.

El departamento de matemáticas pretende corregir un examen en línea de operaciones con fracciones, que se le aplicará a un grupo de estudiantes del curso introductorio. Cuando un estudiante realiza la evaluación del tema, se crea el archivo *"examen.txt"*, en él se almacena la siguiente información: en la primera línea del archivo se encuentra la identificación del estudiante (Nombre y cédula) y en líneas separadas se guarda, Tipo de operación a realizar (1 = Suma, 2 = Resta, 3 = Multiplicación y 4 = división), las dos fracciones a las que se les va a realizar la operación y la respuesta al ejercicio dada por el estudiante, en la forma de fracción mixta. Las fracciones que se van a utilizar se generan aleatoriamente. Además, la evaluación consiste en resolver cinco (5) operaciones, con un puntaje de 4 puntos c/u.

Elabore un programa que dado el archivo *"examen.txt"*, el cual contiene la evaluación realizada por un estudiante, procese la información con el fin que determine e imprima por pantalla, para cada operación evaluada: la representación de la operación, la respuesta correcta y un mensaje que indique respondió bien o no el estudiante,

Además, al final se debe mostrar el puntaje total obtenido por el estudiante en el examen presentado, tomando en cuenta lo indicado anteriormente.

Requerimientos

• Desarrolle un subprograma que reciba dos valores enteros A y B; y retorne el cociente entero y el residuo de la división de A entre B. Ejemplo, Si A=7 y B=3 => Cociente entero es 2 y el residuo entero es 1.

• Desarrolle un subprograma que reciba una fracción de la forma (a/b) y retorne tres valores (c, r, b) que representan una fracción mixta de la forma (c a/b), donde:

sí a <= b, c es Cero (0) y r es el valor original de A y

sí a > b, c es el cociente entero de A/B y R es el residuo entero de a/b

• Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la suma de las dos fracciones, representando la solución en forma de fracción, sabiendo que la fracción resultante es:

num / den = a*d + b*c / (b*d)

• Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la multiplicación de la primera fracción por la segunda, sabiendo que la fracción resultante es:

num / den = a*c / (b*d)

• Desarrolle un subprograma que reciba dos fracciones la forma (a/b) y (c/d), retorne la división de la primera fracción entre la segunda, sabiendo que la fracción resultante es:

num / den = a*d / (b*c)

• Desarrolle un subprograma que reciba dos fracciones mixtas y devuelva si las fracciones son iguales o no.

• Un subprograma que imprima una fracción por pantalla de la forma (a/b).

Ejemplo de los archivos de entrada y salida por pantalla:

1, 8, 2, 3, 4, 4, 6, 8
4, 3, 9, 6, 4, 1, 2, 9
3, 5, 9, 1, 1, 0, 5, 9
2, 6, 3, 4, 8, 1, 12, 24
4, 10, 5, 4, 3, 1, 10, 20

Pantalla de salida

(8/ 2) + (3/ 4) = 4(6/ 8) Correcto
(3/ 9) / (6/ 4) = 0(12/54) Incorrecta
(5/ 9) * (1/ 1) = 0(5/ 9) Correcto
(6/ 3) – (4/ 8) = 1(12/24) Correcto
(10/ 5) / (4/ 3) = 1(10/20) Correcto

Puntaje obtenido = 16 puntos.

Código 1-1. Fracciones.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Apr 19 08:19:29 2022
Solución del ejercicio de operaciones con fracciones
@author: Prof. Alejandro Bolívar
Fecha: 19-04-2022
"""
# subprograma de lectura de los datos de una vuelta
def leer(registro):
    linea = registro.split(',')
    op = int(linea[0])
    num1 = int(linea[1])
    den1 = int(linea[2])
    num2 = int(linea[3])
    den2 = int(linea[4])
    cocr = int(linea[5])
    numr = int(linea[6])
    denr = int(linea[7])
    return op, num1, den1, num2, den2, cocr, numr, denr

# Procedimiento que divide dos números y regresa el cociente y residuo de la
división
def divisionentera(a, b):
    coc = a // b
    res = a % b
    return coc, res

# Procedimiento que forma una fracción mixta a partir de una fracción
def fraccionmixta(a, b):
    if a <= b:  # fracción propia
        c = 0
        r = a
    else:       # fracción impropia
        c, r = divisionentera(a, b)
    return r, c, b

# Procedimiento que suma dos fracciones
def sumafraccion(a, b, c, d):
    num = a * d + b * c
    den = b * d
    return num, den

# Procedimiento que Resta dos fracciones
def restafraccion(a, b, c, d):
    num = a * d - b * c
    den = b * d
    return num, den
```

```
# Procedimiento que Multiplica dos fracciones
def multiplicafraccion(a,b,c,d):
    num = a * c
    den = b * d
    return num, den

# Procedimiento que Divide dos fracciones
def dividafraccion(a, b, c, d):
    num = a * d
    den = b * c
    return num, den

# Función que compara dos fracciones mixtas y devuelve si son iguales o no
def iguales(c1, r1, b1, c2, r2, b2):
    return (c1 == c2) and (r1 == r2) and (b1 == b2)

# procedimiento que imprime por pantalla una fracción
def imprimirfraccion(num, den):
    print(" (%d / %d) " % (num, den), end="")


def main():

    # Que tengo
    num1: int
    den1: int  # Fracción 1
    num2: int
    den2: int  # Fracción 2
    op: int
    cocr: int  # Fracción mixta resultante por el estudiante
    numr: int
    denr: int
    # Que Quiero
    punt: int = 0  # Puntaje obtenido, Acumulador
    # Variables auxiliares
    coc: int
    num: int
    den: int  # Fracción mixta correcta
    i: int

    arch = open("examen.txt", "r")
    punt = 0
    # Ciclo de lectura
    for registro in arch:
        op, num1, den1, num2, den2, cocr, numr, denr = leer(registro)

        # Determinación del resultado correcto a la operación a realizar e
        # Impresión del resultado
        imprimirfraccion(num1, den1)

        if op == 1:
            num, den = sumafraccion(num1, den1, num2, den2)
```

```
            print("+", end="")
        elif op == 2:
            num, den = restafraccion(num1, den1, num2, den2)
            print("-", end="")
        elif op == 3:
            num, den = multiplicafraccion(num1, den1, num2, den2)
            print("*", end="")
        else:
            num, den = dividafraccion(num1, den1, num2, den2)
            print("/", end="")

        imprimirfraccion(num2, den2)

        # Determinación de la fracción mixta resultante
        num, coc, den = fraccionmixta(num, den)
        print("= %d" % coc, end="")
        imprimirfraccion(num, den)

        if iguales(cocr, numr, denr, coc, num, den):
            print(" Correcto")
            punt += 4
        else:
            print(" Incorrecta")

    # Impresión del puntaje obtenido
    print ("Puntaje Obtenido = %d Ptos" % punt)
    arch.close()

    # Mensaje al usuario
    input("Pulse una tecla para finalizar")

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

1.11 Ejercicios propuestos de funciones

1) Valide y devuelva un valor x entre a y b
2) Compruebe que tres valores (x, y, z) son diferentes.
3) Devolver el valor intermedio de tres valores (x, y, z) diferentes.
4) Comprobar si un valor x es par, impar o cero.
5) Compruebe sí un número x es múltiplo de un valor k.
6) Devuelva la cantidad de dígitos de x.
7) Cuántos dígitos impares tiene x
8) Dados tres valores (x, y, z) verifique sí están ordenados ascendentemente, descendentemente o desordenados.
9) Cuál es el mayor de tres valores (x, y, z).
10) Determine sí dos valores (a, b) son iguales sí |a-b|<=tolerancia.

11) Leer por teclado n dígitos (0..9) y devolver el número formado (los dígitos van de izquierda a derecha).
12) Dado un número entero devuelva el digito ubicado en la n-ésima posición a partir de la derecha.
13) Dados tres valores (x, y, z) devolverlos ordenados.
14) Leer n valores enteros desde un archivo e indique cuáles valores faltan en un rango comprendido desde el menor hasta el máximo valor.
15) Dados dos archivos "a.txt" y "b.txt" que contienen números enteros, devuelva un archivo "unión.txt" con los números de ambos archivos, otro archivo "intersección.txt" con los números comunes de ambos archivos y otro "diferencia_simetrica.txt" con los números no comunes en ambos archivos.
16) Dado un archivo "datos.txt" que contiene una lista de números enteros, verifica si tienen un orden ascendente, descendente o desordenado.
17) Los números perfectos y números amigos
    a. Escribir una función que devuelva la suma de todos los divisores de un número n, sin incluirlo.
    b. Usando la función anterior, escribir una función que imprima los primeros m números tales que la suma de sus divisores sea igual a sí mismo (es decir los primeros m números *perfectos*).
    c. Usando la primera función, escribir una función que imprima las primeras m parejas de números (a,b), tales que la suma de los divisores de a es igual a b y la suma de los divisores de b es igual a a (es decir las primeras m parejas de números *amigos*).
    d. Proponer optimizaciones a las funciones anteriores para disminuir el tiempo de ejecución.
18) Escribir una función que reciba dos números como parámetros, y devuelva cuántos múltiplos del primero hay, que sean menores que el segundo.
    a. Implementarla utilizando un ciclo `for`, desde el primer número hasta el segundo.
    b. Implementarla utilizando un ciclo `while`, que multiplique el primer número hasta que sea mayor que el segundo.
    c. Comparar ambas implementaciones: ¿Cuál es más clara? ¿Cuál realiza menos operaciones?
19) **Bisiesto**. Escribe un programa que incluya la función bisiesto(n), que indique si un año es o no es un año bisiesto (devuelve valor Booleano). Un año bisiesto tiene 366

días. Después de la reforma gregoriana, los años bisiestos son los múltiples de cuatro que no terminan con dos ceros, y también los años que terminan con dos ceros que, después de eliminar estos dos ceros, son divisibles por cuatro. Así, 1800 y 1900, aunque eran múltiplos de cuatro, no eran años bisiestos; Por el contrario, 2000 fue un año bisiesto. Dicho en formalismo lógico: un año bisiesto es divisible por 400 o bien divisible por 4 exceptuando los divisibles por 100. Ejemplo:

```
>>> bisiesto(2000)
True
>>> bisiesto(1900)
False
```

20) **Coordenadas cartesianas.** Diseña una función cartesianas(mod, ang) que reciba el valor del módulo y de su ángulo en grados y devuelva sus coordenadas cartesianas (x, y) ajustados a 4 decimales usando la función interna round(value; digits). Ejemplo:

```
>>> from math import sin, cos, pi
>>> cartesianas(1,pi/3)
(0.5, 0.866)
>>> cartesianas(1.4142,pi/4)
(1.0, 1.0)
```

21) **Impuestos**. Una empresa desea calcular la estimación del importe de impuestos que los empleados deben pagar. Los ingresos inferiores a 8.000 euros no están sujetos a impuestos; los comprendidos entre 8.000 euros y 20.000 euros, lo están al 18 %; los comprendidos entre 20.000 euros y 35.000 euros, están sujetos al 27% y los superiores a 35.000 euros, lo están al 38%. Diseña una función impuestos(x) que calcule los impuestos correspondientes a los ingresos x.

Entrada: Un valor x de ingresos.

Salida: Los impuestos correspondientes a x con 2 decimales.

Ejemplos:

```
>>> impuestos(15000)
2700.0
>>> impuestos(55346)
21031.48
```

22) **IMC**. Diseña una función imc(p,h) que reciba reciba el peso p y la altura h de una persona, calcule el índice de masa corporal de una persona (IMC = peso[kg]/altura2[m]) y retorne el estado en el que se encuentra esa persona en función del valor de IMC:

- <16: criterio de ingreso hospitalario
- de 16 a 17: infrapeso
- de 17 a 18: bajo peso

- de 18 a 25: saludable
- de 25 a 30: sobrepeso
- de 30 a 35: sobrepeso crónico
- de 35 a 40: obesidad premórbida
- >40: obesidad mórbida

Entrada: Peso p y altura h.
Salida: Estado.
Ejemplos:

```
>>> imc(1.65,68)
'saludable'
```

23) **Riesgo Cardiaco.** Diseña una función que reciba el peso p en kg, la altura h en metros, el valor del colesterol de lipoproteínas de baja densidad, LDL en mg/dl y si es fumador (o no, variable booleana). La función devuelve si está en riesgo de insuficiencia cardiaca si se cumple que: el índice de masa corporal (IMC = peso[kg]/altura2[m]) es mayor que 35 y que el colesterol (LDL) está por debajo de 71 mg/dl o por encima de 300 mg/dl, o si es fumador. Ejemplo, riesgoCardio(p,h,LDL,fuma):

```
>>> riesgoCardio(95,1.64,310,False)
True
>>> riesgoCardio(90,1.64,310,False)
False
```

24) **Número perfecto.** Diseñe una función que encuentre el primer número perfecto mayor que 28 (o un número n dado). Un número es perfecto si coincide con la suma de sus divisores (excepto él mismo). Por ejemplo, 28 es perfecto ya que 28 = 1 + 2 + 4 + 7 + 14.

```
>>> perfecto(28)
496
>>> perfecto(500)
8128
```

25) ***Números armónicos***. Diseña una función harmon(n) que lea un número **n** e imprima el n-ésimo número armónico, definido como Hn = 1/1 + 1/2 + ⋯ + 1 / n.

Entrada: un número natural n.

Salida: Hn con 4 dígitos después del punto decimal.

```
>>> harmon(2)
1.5000
>>> harmon(7)
2.5929
```

26) ***Es primo?*** Diseña una función es_primo(n) que reciba un natural n y devuelva True si el número es primo o False en caso contrario. Se asume que n > 1.

```
>>> es_primo(15)
```

```
False
>>> es_primo(349)
```
True

27) **Cuántos primos.** Diseña una función cuantos_primos(n) que devuelva el número de primos que existen en el intervalo (1, n) (se excluyen). Usar la función es_primo.

```
>>> cuantos_primos(15)
6
```

28) **Seno por serie de Taylor.** Diseña una función seno_t(x, error=1e-9) que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor. La función puede tener el valor del error por omisión, que sirve para terminar la iteración cuando un término sea menor que el error.

$$seno(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \cdots$$

```
>> seno_t(45)
0.7071067811796194
```

29) **Seno por serie de Taylor v2.** Diseña una función seno_t2(x, error=1e-9) que reciba un valor en grados, lo transforme en radianes y que aproxime el valor de su seno, según su desarrollo en serie de Taylor. No utilizar ni las funciones factorial de math ni elevar xn, usar los términos anteriores para reducir el tiempo de cálculo.

1.12 Guía de estilo - PEP8

En Python existen las denominadas PEP's (*Python Enhancement Proposals*), en concreto la PEP 8 hace referencia a las convenciones de estilo de programación en Python.

Entre las convenciones principales, destacan:

- Usar 4 espacios para indentar.
- Tamaños de línea con un máximo de 79 caracteres.
- Las funciones y las clases se deben separar con dos líneas en blanco, mientras que los métodos de clase solo con uno.
- Los *import* deben de estar separados uno en cada línea. Se permite: `from math import sin, pi`
- Las sentencias import deben de estar siempre en la parte superior del archivo agrupadas de la siguiente manera:
    - Biblioteca estándar
    - Bibliotecas de terceros
    - import's de la aplicación local

- Usar espacios alrededor de los operadores aritméticos, a excepción de cuando forman parte de los argumentos de una función.
- No se deben de realizar comentarios obvios.
- No se deben comparar booleanos mediante ==

1.13 Resumen.

Una función puede tener ninguno, uno o más parámetros y que los mismos pueden o no tener valores por defecto. En el caso de tener más de uno, se separan por comas tanto en la declaración de la función como en la invocación.

Es altamente recomendable documentar cada función que se escribe, para poder saber qué parámetros recibe, qué devuelve y qué hace sin necesidad de leer el código.

Las funciones pueden imprimir mensajes para comunicarlos al usuario, y/o devolver valores. Cuando una función realice un cálculo o una operación con sus parámetros, es recomendable que devuelva el resultado en lugar de imprimirlo, permitiendo realizar otras operaciones con él.

No es posible acceder a las variables definidas dentro de una función desde el programa principal, si se quiere utilizar algún valor calculado en la función, será necesario devolverlo.

Si una función no devuelve nada, por más que se la asigne a una variable, no quedará ningún valor asociado a esa variable.

Unidad 8

La programación como técnica de resolución de problemas numéricos y/o de distintas áreas de ingeniería

En este tema se utiliza la biblioteca Sympy para la realización de cálculos simbólicos, se resuelven ejercicios de diferentes temas asociados al Cálculo Matemático.

SymPy es una biblioteca de Python desarrollada para resolver problemas de matemáticas simbólicas. Existen diversos softwares comerciales que realizan estas tareas: Maple, Mathematica, MATLAB, entre otros, pero requieren una licencia de uso que puede resultar poco accesible en algunos casos. En cambio, SymPy se distribuye bajo licencia BSD, que en resumen permite el uso libre de la misma.

2.1 Importando SymPy

Para importar SymPy y disponer de todos los módulos y funciones que le componen puede hacerse de diversas formas:

1. Forma tradicional

```
>>> import sympy
```

Es la manera más habitual, se carga toda la biblioteca y se accede a cada una de las funciones mediante la sintaxis:

```
>>> r = sympy.funcion(args)
```

2. Importando funciones seleccionadas

```
>>> from sympy import Symbol, integrate, sin, cos
```

De este modo se importan solamente las funciones que vayan a utilizarse, es recomendable cuando se utilizará un número reducido de las mismas. Proporciona cierta ventaja dado que para acceder a una función no es necesario anteponer el nombre de la biblioteca (Sympy), aunque esto mismo represente una desventaja en aquellos casos en los que existen funciones de diferentes bibliotecas con el mismo nombre.

3. Utilizando un alias o seudónimo

```
>>> import sympy as sp
```

Funciona del mismo modo que para el primer caso, con la diferencia que el usuario puede asignarle un nombre más corto o bien más representativo para hacer las llamadas a

funciones. Para los ejemplos que se mostrarán en esta entrada se utilizará la segunda forma. Para visualizar la versión de sympy, se escribe:

```
>>> sympy.__version__
```

2.2 Declarando una variable simbólica

Para declarar una variable simbólica podemos utilizar la función Symbol, para ello primero importamos la función y posteriormente declaramos una variable simbólica "x":

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> x
x
>>> x + 2
x + 2
```

Como puede verse, una vez se ha declarado la variable simbólica puede utilizarle para formar expresiones algebraicas de todo tipo. Existe una forma más "simple" de declarar una variable simbólica, para ello habrá de importarse del módulo "abc" la letra correspondiente, por ejemplo:

```
>>> from sympy.abc import x
```

O bien:

```
>>> from sympy.abc import x,y,z
```

Lo anterior en el caso de que se requieran múltiples variables simbólicas.

Sympy permite hacer operaciones analíticas o con símbolos en lugar de con valores numéricos, al igual que en Python existen varios tipos de datos numéricos como enteros (int), decimales (float) o booleanos (bool:True, False, etc.), `Sympy` posee tres tipos de datos propios: Real, Rational e Integer, es decir, números reales, racionales y enteros. Esto quiere decir que `Rational(1,2)` representa 1/2, `Rational(5,2)` a 5/2, etc. en lugar de 0.5 o 2.5.

```
>>> import sympy as sp
>>> a = sp.Rational(1,2)
>>> a
1/2
>>> a * 2
1
>>> sp.Rational(2)**50 / sp.Rational(10)**50
1/888178419700125232338905334472656 25
```

2.3 Números complejos

La unidad imaginaria es denotada por `I` en Sympy.

```
>>> 1 + 1 * sp.I
1 + I
>>> sp.I**2
-1
>>> (x * sp.I + 1)**2
(I*x + 1)**2
```

También existen algunas constantes especiales, como el número e o π sin embargo, éstos se tratan con símbolos y no tienen un valor numérico determinado. Eso quiere decir que no se puede obtener un valor numérico de una operación usando el valor pi de Sympy, como (1+pi), como lo haríamos con el de Numpy, que es numérico:

```
>>> sp.pi ** 2
pi**2
>>> sp.pi.evalf()
3.14159265358979
>>> (sp.pi + sp.exp(1)).evalf()
5.85987448204884
```

como se ve, sin embargo, se puede usar el método `evalf()` para evaluar una expresión para tener un valor en punto flotante (float).

Para hacer operaciones simbólicas hay que definir explícitamente los símbolos que vamos a usar, que serán en general las variables y otros elementos de nuestras ecuaciones:

```
>>> x = sp.Symbol('x')
>>> y = sp.Symbol('y')
```

Y ahora ya se puede utilizar:

```
>>> x + y + x - y
2*x
>>> (x + y) ** 2
(x + y) ** 2
```

2.4 Manejo de expresiones algebraicas

2.4.1 Factorizar una expresión algebraica.

Para factorizar una expresión algebraica podemos utilizar la función factor, por ejemplo, suponga que se quiere factorizar la expresión (x^2+2x+1):

```
>>> from sympy import factor, Symbol
>>> x = Symbol('x')
>>> factor(x ** 2 + 2 * x + 1)
(x + 1)**2
```

2.4.2 Expandir una expresión algebraica

Enseguida se muestra un ejemplo de cómo "expandir" o multiplicar dos expresiones algebraicas.

```
>>> from sympy import Symbol, expand
>>> x = Symbol('x')
>>> expand((x + 2) * (x - 3))
x**2 - x - 6
>>> ((x + y)**2).expand()
2*x*y + x**2 + y**2
>>> expand((x+1)**2)
x**2 + 2*x + 1
>>> factor(x**2+6*x-16)
(x - 2)*(x + 8)
```

Es posible hacer una substitución de variables usando subs(viejo, nuevo):

```
>>> ((x + y)**2).subs(x, 1)
(1 + y) ** 2
>>> ((x + y)**2).subs(x, y)
4*y**2
```

2.4.3 Simplificación

Utilice *simplify* para transformar una expresión en algo más sencillo.

```
>>> from sympy import simplify
>>> simplify((x + x * y) / x)
1 + y
```

Simplificación es un término vago, es por ello que existen alternativas más precisas que simplify: powsimp (simplificación de exponentes), trigsimp (para expresiones trigonométricas), logcombine, radsimp, together.

2.5 Operaciones algebraicas

Podemos usar `apart(expr, x)` para hacer una descomposición parcial de fracciones:

```
>>> from sympy import apart
>>> 1 / ((x + 2) * (x + 1))
      1
─────────────
(2 + x)*(1 + x)
>>> apart(1 / ((x + 2) * (x + 1)), x)
  1         1
─────── - ─────
1 + x     2 + x
>>> (x + 1) / (x - 1)
-(1 + x)
```

```
________
 1 - x
>>> apart((x + 1) / (x - 1), x)
      2
1 - ------
    1 - x
```

2.6 Ecuaciones algebraicas

También se pueden resolver sistemas de ecuaciones de manera simbólica:

```
>>> # Una ecuación, resolver x
>>> from sympy import solve
>>> solve(x**4 - 1, x)
[-1, 1, -I, I]
>>> solve(x**2 - 3 * x + 2)
[1, 2]
```

```
>>> solve(x**3+5)
```

$$\left[-\sqrt[3]{5},\quad \frac{\sqrt[3]{5}}{2}-\frac{\sqrt{3}i}{2}\sqrt[3]{5},\quad \frac{\sqrt[3]{5}}{2}+\frac{\sqrt{3}i}{2}\sqrt[3]{5}\right]$$

```
>>> solve((x**2+2)>0)
```

$$\Im x = 0 \wedge -\infty < \Re x \wedge \Re x < \infty$$

```
# Sistema de dos ecuaciones. Resuelve x e y
>>>  sp.solve([x + 5*y - 2, -3*x + 6*y - 15], [x, y])
{y: 1, x: -3}
```

```
>>> solve([x+y-1,x-y])
```

$$\left\{x : \frac{1}{2},\quad y : \frac{1}{2}\right\}$$

```
>>> solve([x + y + z - 1, x + y + 2*z - 3 ])
```

$$\{x : -y - 1,\quad z : 2\}$$

2.7 Cálculo de límites

Sympy puede calcular límites usando la función `limit()` con la siguiente sintaxis: `limit(función, variable, punto)`, lo que calcularía el límite de $f(x)$ cuando variable -> punto:

```
.. code:: ipython3
```

```
>>> x = sp.Symbol("x")
```

```
>>> sp.limit(sin(x)/x, x, 0)
1
```

es posible incluso usar límites infinitos:

```
>>> sp.limit(x, x, oo)
oo
>>> sp.limit(1/x, x, oo)
0
>>> sp.limit(x**x, x, 0)
1
```

2.8 Cálculo de derivadas

La función de Sympy para calcular la derivada de cualquier función es `diff(func, var)`. Veamos algunos ejemplos:

```
>>> x = sp.Symbol('x')
>>> diff(sp.sin(x), x)
cos(x)
>>> diff(sp.sin(2*x), x)
2*cos(2*x)
>>> diff(sp.tan(x), x)
1 + tan(x)**2
```

Se puede comprobar que es correcto calculando el límite:

```
>>> dx = sp.Symbol('dx')
>>> sp.limit((sp.tan(x+dx)- sp.tan(x) )/dx, dx, 0)
1 + tan(x)**2
```

También se pueden calcular derivadas de orden superior indicando el orden de la derivada como un tercer parámetro opcional de la función `diff()`:

```
>>> sp.diff(sp.sin(2*x), x, 1)         # Derivada de orden 1
2*cos(2*x)
>>> sp.diff(sp.sin(2*x), x, 2)         # Derivada de orden 2
-4*sin(2*x)
>>> sp.diff(sp.sin(2*x), x, 3)         # Derivada de orden 3
-8*cos(2*x)
```

2.9 Expansión de series

Para la expansión de series se aplica el método `series(var, punto, orden)` a la serie que se desea expandir:

```
>>> from sympy import series
>>> sp.cos(x).series(x, 0, 10)
1 - x**2/2 + x**4/24 - x**6/720 + x**8/40320 + O(x**10)
>>> (1/sp.cos(x)).series(x, 0, 10)
1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)
```

```
e = 1/(x + y)
s = e.series(x, 0, 5)
pprint(s)
```

La función `pprint` de Sympy imprime el resultado de manera más legible:

```
     4    3    2
1   x    x    x    x
- + -- - -- + -- - -- + O(x**5)
y    5    4    3    2
    y    y    y    y
```

Si se utiliza la **qtconsole** (iniciada con `ipython qtconsole`) y se ejecuta `init_printing()`, las fórmulas se mostrarán con $LaTeX$, si está instalado.

2.10 Integración

Las integrales son unos de los conceptos básicos en la formación matemática de un ingeniero, es en términos básicos la operación inversa de la derivación. Pero, además del concepto puramente matemático, las integrales tienen múltiples interpretaciones geométricas y físicas.

En un curso ordinario de cálculo se enseñan métodos para resolver de forma analítica funciones que sean integrables. Por ejemplo, la integral de una función constante será:

$$\int a\,dx = ax + C$$

Y lo sabemos porque nos hemos aprendido reglas básicas de integración y por supuesto a identificar el tipo de función. Actualmente existen paquetes de álgebra simbólica que son capaces de realizar esta tarea: identificar el caso que se tiene y aplicar métodos computacionales, hasta cierto grado complejos, para determinar la solución.

Y claro, SymPy es uno de esos sistemas de álgebra computacional (CAS), en el que solo necesitamos escribir nuestra función a integrar, utilizar por ahí alguna rutina y obtener un resultado rápidamente. Pero claro, para ello debemos aprender mínimamente la sintaxis y eso es justo lo que veremos enseguida.

2.10.1 Integrales simples

Vamos a ver cómo resolver integrales simples indefinidas, si, de esas que vemos en un primer curso. Para resolverlas se utiliza la función `integrate`. Por ejemplo, dada la siguiente función $f(x)=x^2-3x+2$

Como primer paso se importa las funciones necesarias del paquete SymPy:

```
import sympy as sp
from sympy import integrate, init_printing
from sympy.abc import x
init_printing()
```

Del módulo `abc` se importa la variable simbólica `x` e `integrate` para resolver la integral. Ahora, se *guardar* la función a integrar en una variable o bien se pasa directamente como argumento:

```
f = x**2 - 3*x + 2
sp.integrate(f)
```

$$\frac{x^3}{3} - \frac{3x^2}{2} + 2x$$

En este caso no hemos tenido inconvenientes, porque en la expresión a integrar sólo existe una variable simbólica, pero si la expresión tuviese más de una, habría que especificar de manera explícita la variable respecto a la cual se integra, de lo contrario Python *mostrará* un error:

```
from sympy.abc import a,b,c
f = a*x**2+b*x+c
sp.integrate(f)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-3-476839d3c49d> in <module>()
      1 from sympy.abc import a,b,c
      2 f = a*x**2+b*x+c
----> 3 integrate(f)

~\Anaconda3\lib\site-packages\sympy\integrals\integrals.py             in
integrate(*args, **kwargs)
   1289     risch = kwargs.pop('risch', None)
   1290     manual = kwargs.pop('manual', None)
-> 1291     integral = Integral(*args, **kwargs)
   1292
   1293     if isinstance(integral, Integral):

~\Anaconda3\lib\site-packages\sympy\integrals\integrals.py in __new__(cls,
function, *symbols, **assumptions)
     73             return function._eval_Integral(*symbols, **assumptions)
     74
---> 75             obj = AddWithLimits.__new__(cls, function, *symbols,
**assumptions)
     76         return obj
     77
~\Anaconda3\lib\site-packages\sympy\concrete\expr_with_limits.py       in
__new__(cls, function, *symbols, **assumptions)
    375                         " more than one free symbol, an integration
variable should"
    376                         " be supplied explicitly e.g., integrate(f(x,
y), x)"
--> 377                         % function)
    378             limits, orientation = [Tuple(s) for s in free], 1
```

```
    379

      ValueError:  specify dummy variables for a*x**2 + b*x + c. If the
integrand contains more than one free symbol, an integration variable should
be supplied explicitly e.g., integrate(f(x, y), x)
```

Pues eso, si intentamos integrar la función $f(x)=ax^2 + bx + c$ sin especificar la variable de integración, Python nos mandará un error que es bastante sugerente al respecto. Así, lo correcto sería:

```
sp.integrate(f, x)
```

$$\frac{ax^3}{3}+\frac{bx^2}{2}+cx$$

Veamos algunos ejemplos:

```
>>> sp.integrate(6 * x**5, x)
x**6
>>> sp.integrate(sp.sin(x), x)
-cos(x)
>>> sp.integrate(sp.log(x), x)
-x + x*log(x)
>>> sp.integrate(2 * x + sp.sinh(x), x)
cosh(x) + x**2
```

También con funciones especiales:

```
>>> sp.integrate(exp(-x**2) * erf(x), x)
pi**(1/2)*erf(x)**2/4
```

2.10.2 Integrales definidas

Una integral definida usualmente se utiliza para calcular el área bajo la curva de una función en un intervalo finito. En SymPy, para calcular una integral definida se utiliza la función `integrate`, considerando el hecho que deben adicionarse los límites de evaluación mediante la sintaxis:

sp.`integrate(fun, (var, a, b))`

Donde `fun` es la función, `var` la variable respecto a la cual se integra, `a` el límite inferior y `b` el límite superior.

Para ejemplificar vamos a resolver la siguiente integral definida:

$$\int_0^{\frac{x}{2}} \cos x \, dx$$

```
from sympy import cos,pi

sp.integrate(cos(x), (x,0,pi/2.0))
1
```

Otro ejemplo:

$$\int_0^5 x\,dx$$

```
sp.integrate(x, (x,0,5))
25
──
2

sp.integrate(x**3, (x, -1, 1))
0
sp.integrate(sin(x), (x, 0, pi/2))
1
sp.integrate(cos(x), (x, -pi/2, pi/2))
2
```

Y también integrales impropias:

```
sp.integrate(exp(-x), (x, 0, oo))
1
sp.integrate(log(x), (x, 0, 1))
-1
```

Algunas integrales definidas complejas es necesario definirlas como objeto Integral() y luego evaluarlas con el método evalf():

```
from sympy import Integral, evalf
integ = sp.Integral(sin(x)**2/x**2, (x, 0, oo))
integ.evalf()
1.6
```

2.10.3 Integrales múltiples

Ahora vamos a resolver integrales dobles (la sintaxis/metodología de resolución que se revisará aplica para cualquier integral múltiple). Por ejemplo, vamos a resolver la integral dada por:

$$\int_a^b \int_c^d dydx$$

Recuerde que este tipo de integrales múltiples se resuelven de forma *iterada*, yendo *de dentro hacia afuera*, es decir, para la integral anterior se procedería:

$$I1 = \int_c^d dy \;\rightarrow\; I = \int_a^b I_1 dx$$

En Python/SymPy se hace exactamente lo mismo:

```
from sympy.abc import x, y, z, a, b, c, d
from sympy import simplify
I1 = sp.integrate(1, (y, c, d))
sp.simplify(sp.integrate(I1, (x, a, b) ) )
```

$(a-b)(c-d)$

```
>>> fxy = cos(x)+sin(y)
>>> integrate(fxy,(x,0,pi/2),(y,0,pi))
```

$$2\pi$$

```
>>> fxy2 = x**2 + y**2
>>> integrate(fxy2,(y,0,2),(x,0,1))
```

$$\frac{10}{3}$$

2.11 Ecuaciones diferenciales

SymPy es capaz de resolver (algunas) ecuaciones diferenciales ordinarias. sympy.ode.dsolve funciona de la siguiente forma

```
In [4]: f(x).diff(x, x) + f(x)
Out[4]:
   2
  d
─────(f(x)) + f(x)
dx dx

In [5]: dsolve(f(x).diff(x, x) + f(x), f(x))
Out[5]: C₁*sin(x) + C₂*cos(x)
```

Se pueden usar argumentos en las keywords para ayudar a encontrar el mejor sistema de resolución posible. Por ejemplo, si a priori conoces que estás tratando con ecuaciones separables, puedes usar la palabra clave (keyword) hint=’separable’ para forzar a dsolve a que lo resuelva como una ecuación separable.

```
In [6]: dsolve(sin(x)*cos(f(x)) + cos(x)*sin(f(x))*f(x).diff(x),
f(x), hint='separable')
Out[6]: -log(1 - sin(f(x))**2)/2 == C1 + log(1 - sin(x)**2)/2
```

Otro ejemplo:

```
>>> ec_dif = f(x).diff(x) + k*f(x)   # Ecuación diferencial
>>> dsolve(ec_dif, f(x))  # Resolver respecto a f(x)
```

$$f(x) = C_1 e^{-kx}$$

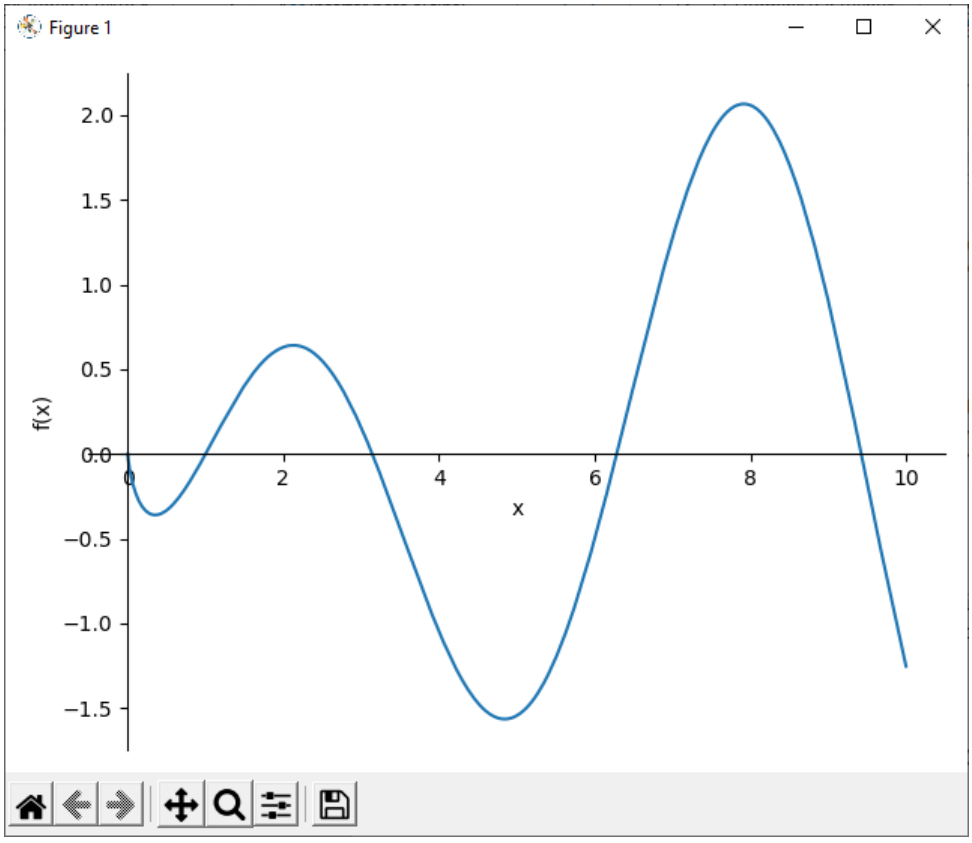
```
>>> ec_dif2 = f(x).diff(x,2) + f(x)
>>> dsolve(ec_dif2, f(x))  # Resolver respecto a f(x)
```

$$f(x) = C_1 \sin(x) + C_2 \cos(x)$$

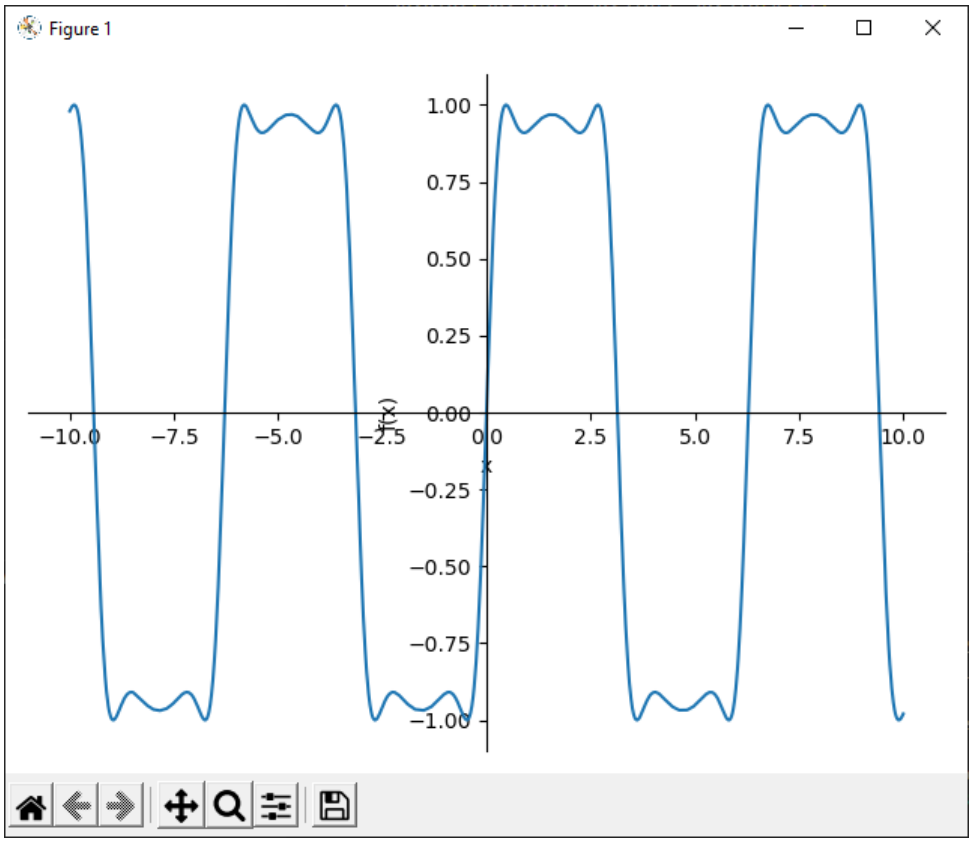
2.12 Gráficos con Sympy

Sympy trae su propio módulo para hacer gráficos, que en realidad **utiliza matplotlib**, pero con la ventaja de que no es necesario darle valores numéricos para representar las funciones, si bien se pueden indicar límites.

```
import sympy as sp
x = sp.Symbol('x')
sp.plot(sp.sin(x)*sp.log(x))
```




```
sp.plot(sp.sin(2*sp.sin(2*sp.sin(x))))
```



2.13 Exportando a LATEX

Cuando hemos terminado los cálculos, podemos pasar a $LaTeX$ la ecuación que queremos:

```
In [5]: int1 = sp.Integral(sp.sin(x)**3 /(2*x), x)
In [6]: sp.latex(int1)
Out[6]: '\\int \\frac{\\sin^{3}{\\left (x \\right )}}{2 x}\\, dx'
```

2.14 Ejemplos prácticos.

Ley de Gases Ideales.

```
'''
Ley de Gas Ideal

'''
import sympy as sym

P, V, n, R, T = sym.symbols('P, V, n, R, T')

# Gas constont
R = 8.314  # J/K/gmoL
```

```
R *= 1000  # J/K/kgmoL

# Moles of air
mAir = 1  # kg
mwAir = 28.97  # kg/kg-moL
n = mAir/mwAir

# Temperatura
T = 298

# Equation
eqn = sym.Eq(P*V, n*R*T)

# Solve for P
f = sym.solve(eqn, P)
print(f[0])

# Use the sympy pLot function to plot
sym.plot(f[0], (V, 1, 10), xlabel='Volume m**3', ylabel='Pressure Pa')
```

Referencias

[1] E. Bahit, Python para Principiantes, Buenos Aires, 2012.

[2] P. Gomis, Fundamentos de Programación en Python, Catalunya: Universitat Politècnica de Catalunya, 2018.

[3] A. Soto Suarez y M. Arriagada Benítez, Introducción a la Programación con Python, Pontificia Universidad Catolica de Chile, 2015.

[4] M. J. Mathieu, Introducción a la programación, PRIMERA EDICIÓN EBOOK ed., México: Grupo Editorial Patria, S.A. de C.V, 2014.

[5] A. Marzal Varó, I. Gracia Luengo y P. García Sevilla, Introducción a la Programación con python 3, Castellón de la Plana: Publicacions de la Universitat Jaume I, 2014.

[6] L. Joyanes Aguilar, Fundamentos de programación: Algoritmos, estructura de datos y objetos, 4a. ed., Madrid: McGraw-Hill, 2008.

[7] A. Dasso y A. Funes, Introducción a la Programación, Universidad Nacional de San Luis, 2014.

[8] E. Bahit, Introduccion al Lenguaje Python, Buenos Aires, 2018.

[9] J. M. R. Torres, Manual básico, iniciación a Python 3, 2020.