Análisis numérico

Facultad de Ciencias, UNAM
Notas Python

Jorge Zavaleta Sánchez

Semestre 2022-1

Índice

Introducción

Python es un lenguaje de programación de *uso general*, *interpretado*, *interactivo*, *orientado a objetos* y de *alto nivel*. Fue creado por *Guido van Rossum* entre 1985 y 1990. El código fuente de *Python* está disponible bajo la *Licencia Pública General GNU* (GPL). *Python* debe su nombre al programa de televisión llamado "*Circo volador de Monty Python*".

Python 3.0 fue lanzado en 2008. Aunque se supone que esta versión es incompatible con versiones anteriores, después muchas de sus características importantes se han actualizado para que sean compatibles con la versión 2.7.

Algunas características importantes de *Python* son:

- **Fácil de aprender, leer y mantener**: *Python* está diseñado para ser muy *legible*. Mientras que otros lenguajes de programación usan puntuación, *Python* utiliza con frecuencia *palabras clave* (*keywords*), y tiene menos construcciones sintácticas que otros lenguajes.
- **Una biblioteca estándar amplia**: La mayor parte de la biblioteca de *Python* es muy portátil y compatible con varias plataformas en *UNIX*, *Windows* y *Macintosh*.
- **Modo interactivo**: *Python* admite un modo interactivo que permite realizar pruebas interactivas y depurar fragmentos de código.
- **Portátil**: *Python* puede ejecutarse en una amplia variedad de plataformas de hardware y tiene la misma interfaz en todas las plataformas.
- **Ampliable**: Se puede agregar módulos de bajo nivel al intérprete de *Python*. Estos módulos permiten a los programadores agregar o personalizar sus herramientas para que sean más eficientes.
- **Bases de datos**: *Python* proporciona interfaces a las principales bases de datos comerciales.

- **Programación GUI**: *Python* admite aplicaciones *GUI* (*Graphic User Interface-Interfaz gráfica de usuario*) que se pueden crear y portar a varios *llamados del sistema*, *bibliotecas* y *sistemas de ventanas*, como *Windows MFC*, *Macintosh* y el sistema *X Window* de *Unix*.
- Adicionalmente:
  - Admite métodos de programación funcionales y estructurados, así como programación orientada a objetos (*POO*).
  - Se puede utilizar como lenguaje de programación o se puede compilar en *bytecode* para crear aplicaciones de gran tamaño.
  - Proporciona tipos de datos dinámicos de muy alto nivel y admite la verificación dinámica de tipos.
  - Es compatible con la recolección automática de basura.
  - Se puede integrar fácilmente con otros lenguajes de programación como *C*, *C++*, *Java*, entre otros.

Notas sobre Jupyter Notebook

Jupyter Notebook es un entorno informático interactivo basado en la web para crear documentos de *Jupyter notebook*. Un documento de *Jupyter Notebook* contiene una lista ordenada de celdas de entrada/salida que pueden contener código, texto, matemáticas, gráficos y texto enriquecidos, generalmente guardados con la extensión "*.ipynb*".

Un *Jupyter Notebook* se puede convertir a varios formatos de salida estándar abierto (HTML, diapositiva de presentación, LaTeX, PDF, ReStructuredText , Markdown, Python) a través de "descargar como" en la interfaz web. *Jupyter Notebook* puede conectarse a muchos núcleos (*kernels*) para permitir la programación en muchos lenguajes de programación. Por defecto, *Jupyter Notebook* se conecta con el núcleo *IPython*, que nos permite tener un entorno interactivo para *Python*.

1. Se usarán dos tipos de celdas:
   - `Code`: Donde se escribirá el código de *Python* y que nos permitirá ejecutarlos de forma interactiva dentro de la libreta.
   - `Markdown`: Donde se pueden incorporar comentarios con formato a través del lenguaje *Markdown*.
2. Para evaluar una celda usamos la combinación `shift+enter` o `ctrl+enter`.
3. El símbolo `#` sirve para poner títulos en una celda `Markdown`. Para consultar una referencia rápida de cómo escribir un *Markdown* y *LaTeX* (texto matemático) en las libretas de Jupyter véase la siguiente liga.

Tipos de datos

Dentro de los tipos de datos básicos que maneja *Python* se encuentran:

- Enteros (`int`)
- Flotantes o decimales (`float`)
- Cadenas (`str`)
- Booleanos (`bool`)
- Complejos (`complex`)

Comentarios y funciones básicas en Python

1. Para poner una línea como comentario es necesario utilizar el símbolo `#` al inicio de una línea. Sí se quiere tener un bloque de comentarios, es necesario que cada línea inicie con `#`. Es posible también poner comentarios después de una instrucción con el mismo procedimiento.
2. La función `type` sirve para mostrar el tipo de dato que estamos poniendo como argumento.
3. La función `print` sirve para desplegar el valor que tenga una variable o un tipo de dato.

[1]:
```
# Comentario en una linea
print("Hola mundo") # Comentario despues de una instruccion
```

```
Hola mundo
```

Ejemplos de los tipos de datos

Ahora se mostrará, a través de algunos ejemplos, la sintaxis que se debe utilizar para definir los diferentes tipos de datos a través de *literales*. En programación, un **literal** es una notación para representar un valor fijo en el *código fuente*.

[2]:
```
# Enteros
type(10)   # Notacion decimal
type(0b10) # Notacion binaria
type(0o10) # Notacion octal
type(0x10) # Notacion hexadecimal
```

[2]:
```
int
```

[3]:
```
# Numeros de punto flotante (reales)
type(10.0)  # Notacion con punto decimal
type(1.0e1) # Notacion cientifica 1.0 x 10^1
```

[3]:
```
float
```

[4]:
```
# Cadenas (strings)
type('Cadena') # Con comillas simples o apostrofe
type("Cadena") # Con comillas
type("""Cad
ena""") # Tres pares de comillas
```

[4]:
```
str
```

Nota: Los tres pares de comillas permiten generar comentarios en bloque, ya que *Python* ignorará los literales de cadena que no están asignados a una variable.

[5]:
```
"""Esto me da la
posibilidad de escribir
comentario en un bloque"""
```

[5]:
```
'Esto me da la\nposibilidad de escribir\ncomentario en un bloque'
```

[6]:
```
# Booleanos
type(True) # Verdadero
type(False) # Falso
```

[6]:
```
bool
```

[7]:
```
# Numeros complejos
type(3 + 6j) # La parte imaginaria corresponde al caracter acompañado de j
```

[7]:
```
complex
```

Operadores aritméticos

Los operadores aritméticos disponibles se dan a continuación:

Operador	Operación	Uso
`+`	Suma	`a + b`
`-`	Resta	`a - b`
`*`	Multiplicación	`a * b`
`/`	División	`a / b`
`%`	Módulo (residuo)	`a % b`
`//`	División entera (piso)	`a // b`
`**`	Exponenciación	`a ** b`

La jerarquía de las operadores, es decir, el orden en que se evalúan las operaciones es la siguiente:

- ***Primero***. *Paréntesis*. Aunque no es propiamente un operador, permite cambiar el orden en que se evalúan las operaciones. Para paréntesis anidados, el que está más al interior es el que se evalúa primero.
- ***Segundo***. *Potencias*.
- ***Tercero***. *Multiplicación y división*. (El módulo y la división entera se encuentran también en este nivel)
- ***Cuarto***. *Suma y resta*.

[8]:
```
# Suma
3+4
```

[8]:
```
7
```

[9]:
```
# Resta
4-5
```

[9]:
```
-1
```

[10]:
```
# Multiplicacion
3*7
```

[10]:
```
21
```

[11]:
```
# Division
5/6
```

[11]:
```
0.8333333333333334
```

[12]:
```
# Potencia
3.3**2
```

[12]:
```
10.889999999999999
```

[13]:
```
# Modulo
5%2
```

[13]:
```
1
```

[14]:
```
# Division entera
11//3
```

[14]:
```
3
```

[15]:
```
# Jerarquia
print((3+4)*2)
print(3+4*2)
```

```
14
11
```

Los operadores aritméticos aplicados a tipos de datos numéricos de un mismo tipo pueden producir un tipo de dato numérico diferente, como es el caso de la división entre números enteros, cuyo resultado es de tipo `float`. Para otros tipos de datos también se pueden usar los operadores aritméticos pero en un sentido diferente, por ejemplo, en el caso de la suma de cadenas esta produce concatenación de cadenas.

[16]:
```
# Suma de cadenas = concatenacion de cadenas
"Hola" + " Mundo"
```

[16]:
```
'Hola Mundo'
```

También se pueden hacer operaciones entre diferentes tipos de datos, pero en algunas ocasiones aparecen errores `TypeError`, que ocurren cuando la operación no está definida entre los tipos de datos involucrados. Por ejemplo, sumar una cadena con un entero produce un error de tipo `TypeError`, sin embargo, se puede hacer la multiplicación de cadenas con enteros, cuyo resultado es la repetición de la cadena tantas veces como el valor del entero.

[17]:
```
# Multiplicacion de una cadena por un entero
4*"Hola "
```

[17]:
```
'Hola Hola Hola Hola '
```

Variables

Las variables en *Python* no necesitan una declaración explícita para reservar espacio en la memoria. La declaración ocurre automáticamente cuando se asigna un valor a una variable, ya sea a través de una literal o mediante otra variable previamente declarada. Para declarar una variable se utiliza el operador de *asignación* `=`. La sintaxis a usar es:

```
nombre_variable = valor_variable
```

esto es, el operando a la izquierda del operador `=` es el **nombre de la variable** y el operando a la derecha del operador `=` es el **valor almacenado** en la variable.

Dado que *Python* es un lenguaje interpretado, la asignación de variables es dinámica, por lo que el tipo de dato asignado a una variable se puede cambiar en cualquier momento. Esto requiere poner atención en el manejo de las variables para evitar errores de tipo `TypeError`.

Nota: El nombre de las variables debe comenzar con una letra, de la `A` a la `Z` o de la `a` a la `z` o un guión bajo (`_`) seguido de cero o más letras, guiones bajos y *dígitos* (`0` a `9`). *Python* no permite caracteres de puntuación o caracteres como `@`, `$` y `%` dentro de los nombres. También, al ser un lenguaje de programación sensible a mayúsculas y minúsculas se debe prestar atención en los nombres ya que, por ejemplo, `Variable` y `variable`, hacen referencia a dos identificadores de variables diferentes. En la siguiente liga se puede encontrar las palabras reservadas (*keywords*) que no se pueden usar como nombres de variables.

[18]:
```
# Ejemplo cadenas
x = "Hola mundo"
print(x)
```

```
Hola mundo
```

[19]:
```
# Ejemplo enteros
y = 3 + 4
print(y)
```

```
7
```

[20]:
```
# Ejemplo flotantes, cadenas y salida con formato
z = 3.45*2.87
y = 3*z
s = "El resultado de la operacion es "
print(s,y)
```

```
El resultado de la operacion es  29.704500000000003
```

[21]:
```
# Ejemplo nombre de variable con _
_xz = 4
```

```
print(_xz)
```

```
4
```

Asignación múltiple

Python permite asignar un solo valor a varias variables de manera simultánea. También es posible declarar múltiples variables con su respectivo valor en una sola línea.

[22]:
```
# Asginacion de un solo valor a varias variables
a = b = c = 2
print(a,b,c)
```

```
2 2 2
```

[23]:
```
# Declaracion de multiples variables en una sola linea
d,e,f = 3.45,5,"Cadena"
print(d,e,f)
```

```
3.45 5 Cadena
```

Eliminación de variables

Para eliminar la referencia a una o varias variables se utiliza el comando `del`. La sintaxis a usar es:

```
del var1[,var2[,var3[....,varN]]]
```

[24]:
```
#Ejemplo de eliminacion de variables
print(a,b,c,d,e,f)
del a # Borrar una sola variable
del b,c,d,e,f # Borrar multiples variables
```

```
2 2 2 3.45 5 Cadena
```

Nota: Una vez eliminada una variable ya no será posible acceder a ella a través de su nombre (identificador). Si queremos hacer referencia a una variable que no este declarada o la cual haya sido eliminada, se obtendrá un error de tipo `NameError`.

Variables, operadores booleanos (lógicos) y relacionales

Las *variables booleanas* sólo pueden tomar los valores `True` o `False` y los *operadores booleanos o lógicos* usan operandos de tipo `bool` regresan alguno de estos dos valores aunque los operandos pueden ser de diferentes tipos. La siguiente tabla muestra los *operadores booleanos*:

Operador	Nombre
`and`	y lógico
`or`	o lógico
`not`	Negación
`==`	Igual
`!=`	Diferentes

Operador	Nombre
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Primero se mostrará el uso de los operadores lógicos se Como ejemplo se declararán dos variables booleanas `cero` con el valor `False` y `uno` con el valor `True` para construir las tablas de verdad para los operadores *binarios* `and` y `or`, y para el operador *unario* `not`, con el objetivo de mostrar todos los posibles resultados de aplicar los operadores a las variables booleanas.

[25]:
```
# Declaracion de variables booleanas
cero,uno = False,True
print(cero) # Falso
print(uno) # Verdadero
```

```
False
True
```

[26]:
```
# Tabla de verdad para el operador binario and
print(cero and cero)
print(cero and uno)
print(uno and cero)
print(uno and uno)
```

```
False
False
False
True
```

[27]:
```
# Tabla de verdad para el operador binario or
print(cero or cero)
print(cero or uno)
print(uno or cero)
print(uno or uno)
```

```
False
True
True
True
```

[28]:
```
# Tabla de verdad para el operador unario not
print(not cero)
print(not uno)
```

```
True
False
```

Para literales o variables de tipo numérico se pueden utilizar los operadores de comparación para determinar si se cumple o no una relación de orden, por lo que el resultado de una comparación devuelve un valor de tipo `bool`.

[29]:
```
# Comparacion
a,b = 5,7.3
print(a,"==",b,"es: ",a==b)
print(a,"!=",b,"es: ",a!=b)
print(a,">",b,"es: ",a>b)
print(a,">=",b,"es: ",a>=b)
print(a,"<",b,"es: ",a<b)
print(a,"<=",b,"es: ",a<=b)
```

```
5 == 7.3 es:  False
5 != 7.3 es:  True
5 > 7.3 es:  False
5 >= 7.3 es:  False
5 < 7.3 es:  True
5 <= 7.3 es:  True
```

Para el caso de cadenas (`str`), es posible comparar si dos cadenas son iguales mediante el operador `==` o diferentes con `!=`. Pero incluso podemos hacer comparaciones entre cadenas ya que el orden está establecido a través del orden alfabético.

[30]:
```
# Ejemplo del uso de la comparacion entre cadenas
palabra1 = "Anaconda"
palabra2 = "Raton"

print(palabra1 < palabra2)
print(palabra1 > palabra2)
print(palabra1 == palabra2)
```

```
True
False
False
```

La manera que se hace esto es a través de los valores en la codificación de los caracteres en la computadora. *Python* utiliza preferentemente la codificación UTF-8. Así, se van comparando carácter por carácter a través de sus valores enteros y de esta manera se determina cual está primero alfabéticamente.

Nota: Podemos usar la función `ord` para recuperar el valor entero de un carácter. Y podemos hacer lo contrario usando la función `chr` para una entrada entera.

[31]:
```
# Uso de la funcion ord
print(ord("A"))
print(ord("a"))
print(ord("Á"))
print(ord("á"))
```

```
65
97
193
225
```

[32]:
```
# Uso de la funcion chr
print(chr(ord("A")+0))
print(chr(ord("A")+1))
print(chr(ord("A")+2))
print(chr(ord("A")+3))
```

```
A
B
C
D
```

[33]:
```
# Tambien es posible utilizar su valor hexadecimal dada por la codificacion
print(chr(0x706B))
```

```
火
```

Funciones integradas (*Built-in functions*)

Python cuenta con funciones integradas que se pueden utilizar para los diferentes tipos de datos. Aunque ya hemos utilizado algunas de ellas, en esta parte trataremos de introducir otras que serán de utilidad. Una referencia para consultar dichas funciones puede ser encontrada en la siguiente liga.

Función print

La función `print` nos permite mostrar un mensaje en pantalla.

[34]:
```
# Usando una cadena
print("Hola mundo")
```

```
Hola mundo
```

También vamos a tener la posibilidad de mandar texto con formato. Por ejemplo,

[35]:
```
# Usando una cadena y un entero. Salida con formato
print("Mi numero de la suerte es",7)
```

```
Mi numero de la suerte es 7
```

Sin embargo, se pueden utilizar los *literales de cadena formateados* (*formatted string literal*) o ***f-string***, para facilitar la lectura desde el código. Los *f-string* son un literal de cadena que se prefija con `'f'` o `'F'`, y son cadenas que pueden contener campos de reemplazo, las cuales son expresiones delimitadas por llaves `{}`. El siguiente ejemplo muestra su uso.

[36]:
```
a,b,c = 29,7,18
print("Los números ganadores son:", a,",",b,"y",c)
print(f"Los números ganadores son: {a}, {b} y {c}")
```

```
Los números ganadores son: 29 , 7 y 18
Los números ganadores son: 29, 7 y 18
```

También se pueden insertar expresiones de *Python* entre las llaves. Las expresiones se reemplazan con su resultado en la cadena, pero se recomienda que las expresiones sean simples. Por ejemplo,

[37]:
```
a,b = 8.5,9.73
print(f"{a} x {b} = {a*b}")
```

```
8.5 x 9.73 = 82.705
```

Las *f-strings* sólo están disponibles en la versión de *Python* 3.6 en adelante. En las versiones previas y actuales de *Python*, el método `.format()` se puede usar para obtener los mismos resultados

[38]:
```
a,b = -15.2,79.43
print("{} x {} = {}".format(a,b,a*b))
```

```
-15.2 x 79.43 = -1207.336
```

En este caso, cada uno de las llaves vacías corresponde a cada una de las variables o expresiones en la parte del método `.format()`. En ambos casos es posible añadir un especificador de formato utilizando `:` para controlar como se muestran los valores de las variables en la cadena final.

[39]:
```
pi = 3.14159265359
print(f"Diferentes aproximaciones de pi son: {pi:3.2f}, {pi:<15.4f}, o {pi:2.
 ↪11F}")
print("Diferentes aproximaciones de pi son: {0:3.2f}, {0:=+15.4f}, o {0:2.11f}".
 ↪format(pi))
print(f"{1000000000.00:,.2f}")
```

```
Diferentes aproximaciones de pi son: 3.14, 3.1416          , o 3.14159265359
Diferentes aproximaciones de pi son: 3.14, +        3.1416, o 3.14159265359
1,000,000,000.00
```

En el caso de las *f-strings* es necesario poner la variable y seguido de los dos puntos el especificador de formato que se le aplicará a dicha variable. Si se utiliza el método `.format()` es necesario especificar la posición dentro de este (observe que aquí el primero está indicado por 0) y nuevamente después de los dos puntos el especificador de formato. La sintaxis general para el especificador de formato es:

```
{:[bandera][ancho][.precision]tipo}
```

Una lista de los símbolos disponibles para las banderas es:

Bandera	Descripción
<	Alineación a la izq.
>	Alineación a la der.

Bandera	Descripción
`^`	Alineación centrada
`=`	Pone el signo en la posición más a la izq.
`+`	Usa el signo más para indicar si el resultado es positivo o negativo
`-`	Usa el signo menos solamente para valores negativos
`,`	Separa cantidades por miles

Una lista para los tipos es:

Tipo	Descripción
`b`	Formato binario
`c`	Convierte el valor a su correspondiente caracter unicode
`d`	Formato decimal
`e`	Formato en notación científica, con e minúscula
`E`	Formato en notación científica, con E mayúscula
`f`	Formato de punto fijo
`F`	Formato de punto fijo (`inf` y `nan` se muestran en mayúsculas)
`g`	Formato general
`G`	Formato general (con E para notación científica)
`o`	Formato octal
`x`	Formato hexadecimal, minúsculas
`x`	Formato hexadecimal, mayúsculas
`n`	Formato de número
`%`	Formato de porcentaje

Otras opciones para salida con formato en *Python* se pueden consultar en la siguiente liga.

Función input

Cuando se realiza un *programa* es necesario recibir información o valores para declarar las variables, es decir, asignarles un valor. Por lo general, esto se hace a través de pedir esta información al usuario. La función `input` permite desplegar mensajes y recibir información con la siguiente sintaxis:

```
var_salida = input("Mensaje como cadena de texto ")
```

Una cosa que hay que tener en cuenta es que el argumento de salida (`var_salida`) de `input` es una cadena (tipo `str`), por lo que dependiendo del uso hay que hacer una conversión (*casting*) en el tipo de dato. Esto se hace a través de las funciones

- `int(x)` Convierte el argumento x a uno de tipo `int`.
- `float(x)` Convierte el argumento x a uno de tipo `float`.
- `str(x)` Convierte el argumento x a uno de tipo *str*.

[40]:
```
# Programa para calcular el cuadrado de un numero
n = input("Dame un numero para calcular su cuadrado: ")
print(type(n))
print(n + "^2 = " + str(float(n)**2))
```

```
Dame un numero para calcular su cuadrado: 5
<class 'str'>
5^2 = 25.0
```

La función `eval` analiza la expresión (dada como una cadena) pasada a esta y ejecuta la expresión (código) de *Python* dentro del programa. De esta manera, usada junto con `input`, podemos dar cualquier expresión valida de *Python* para declarar variables de los diferentes tipos de datos sin la necesidad de usar las otras funciones para conversión de tipos.

[41]:
```
t = eval(input("Dame una entrada : "))
print(t)
print(type(t))
```

```
Dame una entrada : 4/8
0.5
<class 'float'>
```

Funciones matemáticas

Para poder usar las funciones matemáticas usuales que encontramos por ejemplo en una calculadora científica es necesario importar un módulo. Dado que Python es un lenguaje de propósito general, varias funciones para propósitos específicos vienen empaquetadas en módulos que es necesario importarlas primero para poder utilizarlas.

Para utilizar una biblioteca o módulo de funciones adicionales utilizamos el comando *import*. Este comando sólo es necesario ejecutarlo una vez al principio.

Módulo `math`

Las funciones matemáticas usuales como funciones trigonométricas, raíz cuadrada, redondeos, etc, además de constantes como π o e vienen definidas en el módulo *math*. Para poder usar dichas funciones o constantes usamos la instrucción *import math* y después hay que utilizarlas a través del espacio de nombres *math*. Por ejemplo, para usar la función seno, es necesario usar el comando *math.sin()*, para indicar que usaremos la función seno definida en el módulo *math*. Esto se hace así debido a que una función puede estar definida en múltiples módulos (como veremos más adelante) y es necesario indicar a que función de que módulo nos referimos y evitar ambigüedades. La documentación del módulo *math* la podemos consultar en la siguiente liga.

[42]:
```
# Importamos el paquete math
import math
```

[43]:
```
# Usamos la funcion coseno de math
math.cos(0)
```

[43]: 1.0

[44]:
```
# Mostramos el valor de la constante pi de math
math.pi
```

[44]:
```
3.141592653589793
```

[45]:
```
# Usamos la funcion raiz cuadrada de math
math.sqrt(2)
```

[45]:
```
1.4142135623730951
```

[46]:
```
# Usamos la funcion techo de math
math.ceil(3.45)
```

[46]:
```
4
```

Ejemplos

Ejemplo 1

1. Evalúe la siguiente expresión y muestre el resultado.

$$(-3.5)^3 + \frac{e^6}{\ln 524} + 206^{1/3}$$

Solución: Para calcular lo que se pide, utilizamos la función exponencial `math.exp()` para evaluar el término e^6 y la función `math.log()` para calcular el logaritmo natural.

[47]:
```
# import math #Es necesario este modulo para cargar las funciones. Descomentar
 ↪si no se ha cargado previamente.
print(f"El resultado es: {(-3.5)**3 + math.exp(6)/math.log(524) + 206**(1/3):2.
 ↪4f}")
```

```
El resultado es: 27.4611
```

2. Evalúe la siguiente expresión y muestre el resultado.

$$\frac{2.5^3\left(16 - \frac{216}{22}\right)}{1.7^4 + 14} + \sqrt[4]{2050}.$$

Solución: Es importante tener en cuenta que $\sqrt[4]{x} = x^{\frac{1}{4}}$. En este caso en lugar del operador `**`, se uso la función `math.pow()` para calcular este término.

[48]:
```
# import math #Es necesario este modulo para cargar las funciones. Descomentar
 ↪si no se ha cargado previamente.
print(f"El resultado es: {2.5**3*(16-216/22) / (1.7**4+14)+ math.pow(2050,1/4):2.
 ↪4f}")
```

```
El resultado es: 11.0501
```

3. Evalúe la siguiente expresión y muestre el resultado.

$$\frac{\tan 64^\circ}{\cos^2 14^\circ} - \frac{3\sin 80^\circ}{\sqrt[3]{0.9}} + \frac{\cos 55^\circ}{\sin 11^\circ}.$$

Solución: Como la expresión contempla los ángulos medidos en grados, es necesario convertirlos a radianes, ya que las funciones trigonométricas dentro del módulo `math` reciben los argumentos en radianes. Para ello se puede usar la función `math.radians()` para hacer la conversión de grados a radianes. También existe la función `math.degrees()` para el proceso inverso. Dado que la expresión es muy larga, para partirla en múltiples líneas se puede usar el símbolo \ al final de cada línea. Esto le índica a *Python* que la instrucción continua en la siguiente línea, y de este modo la instrucción es legible.

[49]:
```
# import math #Es necesario este modulo para cargar las funciones.  Descomentar
 ↪si no se ha cargado previamente.
aux = math.tan(math.radians(64))/math.cos(math.radians(14))**2 \
               - 3*math.sin(math.radians(80))/math.pow(0.9,1/3) \
               + math.cos(math.radians(55))/math.sin(math.radians(11))
print(f"El resultado es: {aux:2.4f}")
```

```
El resultado es: 2.1238
```

4. Defina las variables a, b, c y d como: $a = 12$, $b = 5.6$, $c = \dfrac{3a}{b^2}$, y $d = \dfrac{(a-b)^c}{c}$, evalúe la siguiente expresión y muestre el resultado

$$\exp\left(\frac{d-c}{a-2b}\right) + \ln\left(\left|c - d + \frac{b}{a}\right|\right)$$

Solución: Aquí solo hay tener cuidado en la asignación de los valores a las variables y usar la función `math.fabs()` para calcular el valor absoluto.

[50]:
```
# import math #Es necesario este modulo para cargar las funciones.  Descomentar
 ↪si no se ha cargado previamente.
a,b = 12,5.6
c = 3*a/b**2
d = (a-b)**c/c
print(f"El resultado es: {math.exp((d-c)/(a-2*b)) + math.log(math.fabs(c - d + b/
 ↪a)):2.4f}")
```

```
El resultado es: 2292.5402
```

Arreglos

Para empaquetar conjuntos de datos, en este caso lo podemos hacer en *Python* mediante:

- **Listas** (`list`). Puede cambiar de forma dinámica su contenido mediante los métodos definidos para las listas, además de que puede contener diferentes tipos de datos.
- **Tuplas** (`tuple`). Es equivalente a una lista, pero con la diferencia de que no puede cambiar de tamaño.

Notas:

- Tanto las listas como las tuplas son el análogo a un *arreglo*, pero con la diferencia que pueden tener diferentes tipos de datos.

- La numeración de los indices comienza en 0 (cero).
- El acceso a los elementos se hace a través de corchetes `[]`.
- Se pueden generar arreglos n-dimensionales.

Listas

Se generan mediante corchetes `[]` y los elementos se separan mediante comas `,`. Para generar listas n-dimensionales es necesario anidar listas, es decir, hacer listas de listas.

[51]:
```
# Listas simple con el mismo tipo de dato
L = [1,2,3,4,5,6]
type(L)
```

[51]:
```
list
```

[52]:
```
# Listas simple con diferentes tipos de dato
L2 = [1,"Algo",True]
print(L2)
```

```
[1, 'Algo', True]
```

[53]:
```
# Acceso al segundo elemento de la lista
L2[1]
```

[53]:
```
'Algo'
```

[54]:
```
# Lista bidimensional, analogo a poner directamente␣
↪[[1,"Algo",True],[1,2,3,4,5,6]]
L3 = [L2,L]
print(L3)
```

```
[[1, 'Algo', True], [1, 2, 3, 4, 5, 6]]
```

[55]:
```
# Acceso al segundo elemento de la primera lista
L3[0][1]
```

[55]:
```
'Algo'
```

[56]:
```
# Lista tridimensional
L4 = [L3,L3]
print(L4)
```

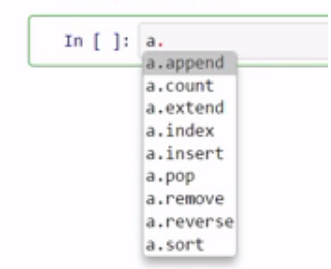
```
[[[1, 'Algo', True], [1, 2, 3, 4, 5, 6]], [[1, 'Algo', True], [1, 2, 3, 4, 5,
6]]]
```

[57]:
```
# Acceso al cuarto elemento de la segunda lista de la primera lista
L4[0][1][3]
```

[57]:
```
4
```

Se puede modificar el tamaño de la lista a través de los métodos definidos para listas. Recuérdese que *Python* es un lenguaje de programación orientado a objetos, en el cual cada tipo de dato es un *objeto* definido mediante sus *atributos*, que es lo que define el estado de cada objeto, y este objeto cuenta con diferentes *métodos*, los cuales pueden ser pensados como funciones que actúan sobre los atributos.

Para acceder a los métodos es necesario poner `.` después del objeto y escribir el nombre del método, que incluso puede tener argumentos adicionales. En *Jupyter* y en *Spyder* es posible mostrar los métodos disponibles para un objeto usando la tecla del **tabulador**. Para ello después de poner `.` apretamos la tecla `Tab` y aparecerá un menú, como se muestra en la siguiente imagen



En este curso no se hará énfasis en la parte orientada a objetos pero se usarán los métodos cuando sea conveniente

Algunos ejemplos para agregar nuevos elementos a una lista se muestra a continuación.

[58]:
```
# Agregamos el numero 7 al final de la lista L con el metodo append
L.append(7)
print(L)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

[59]:
```
# Agregamos la dcadena 'Hola' en la posicion 3 de la lista L con el metodo insert
L.insert(3,'Hola')
print(L)
```

```
[1, 2, 3, 'Hola', 4, 5, 6, 7]
```

Tuplas

Se generan mediante paréntesis `()` y los elementos se separan mediante comas `,`. Para generar tuplas n-dimensionales es necesario anidar tuplas, es decir, hacer tuplas de tuplas.

[60]:
```
# Tupla
T = (1,2,3,4)
type(T)
```

[60]:
```
tuple
```

[61]:
```
# Acceso al segundo elemento en la tupla T
T[1]
```

[61]: 2

[62]:
```
# Tupla bidimensional y acceso mediante 2 indices
T2 = (T,T)
print(T2)
T2[0][0]
```

```
((1, 2, 3, 4), (1, 2, 3, 4))
```

[62]: 1

[63]:
```
# Podemos combinar tuplas con listas y viceversa
T3 = (T,L)
print(T3)
```

```
((1, 2, 3, 4), [1, 2, 3, 'Hola', 4, 5, 6, 7])
```

[64]:
```
# Mostramos el tipo al que corresponde
print(type(T3)) # tupla
print(type(T3[0])) # tupla
print(type(T3[1])) # lista
T3[1][3] # Acceso usual mientras el objeto lo permita
```

```
<class 'tuple'>
<class 'tuple'>
<class 'list'>
```

[64]: 'Hola'

Acceso a porciones de un arreglo (*slices*)

Podemos acceder de forma dinámica a los elementos de un arreglo mediante `:`.

Nota: El acceso está dado de la siguiente manera: `inicio : final : incremento`, que se conocen como *slices*. Una cosa importante que se debe considerar en este caso es que el valor final no se toma en cuenta, es decir, los valores son tomados menores estrictos que el valor final. Por ejemplo, si lo consideramos como un intervalo, los valores están en $[inicio, final)$.

[65]:
```
# Los valores de L del indice 3 hasta el 5 (6 menos el incremento de 1 que esta␣
↪implicito)
L[3:6]
```

[65]: ['Hola', 4, 5]

[66]:
```
# Todos los elementos de la lista
L[:]
```

[66]:
```
[1, 2, 3, 'Hola', 4, 5, 6, 7]
```

[67]:
```
# Del indice 3 hasta el final de la lista
L[3:]
```

[67]:
```
['Hola', 4, 5, 6, 7]
```

[68]:
```
# Del inicio de la lista hasta el indice 2 (3 menos el incremento de 1 que esta␣
↪implicito)
L[:3]
```

[68]:
```
[1, 2, 3]
```

[69]:
```
# Del indice 2 hasta el indice 5 con incrementos de 2
L[1:6:2]
```

[69]:
```
[2, 'Hola', 5]
```

Nota: Si se usan valores negativos, el arreglo se recorre del final al inicio, donde el último elemento tendría el índice `-1`.

[70]:
```
# Ultimo elemento de L
L[-1]
```

[70]:
```
7
```

[71]:
```
# Recorriendo de atras hacia adelante la lista
L[-1:-len(L)-1:-1]
```

[71]:
```
[7, 6, 5, 4, 'Hola', 3, 2, 1]
```

Nota: La función `len` nos permite determinar la longitud de una cadena, lista, tupla, etc.

[72]:
```
print(f"La longitud de la cadena 'Cadena' es {len('Cadena')}")
print(f"La longitud de la lista {L} es {len(L)}")
print(f"La longitud de la tupla {T} es {len(T)}")
```

```
La longitud de la cadena 'Cadena' es 6
La longitud de la lista [1, 2, 3, 'Hola', 4, 5, 6, 7] es 8
La longitud de la tupla (1, 2, 3, 4) es 4
```

Módulo `numpy`

El módulo `numpy` es una biblioteca para cómputo científico que nos permite hacer cálculos numéricos y tener acceso a rutinas numéricas que se pueden encontrar en *Matlab/Octave*. De esta manera,

dentro de este módulo se cuenta con las funciones y constantes matemáticas que encontramos en el módulo `math`.

[73]:
```
# Cargamos el modulo
import numpy as np
```

[74]:
```
# Funcion coseno en numpy
np.cos(np.pi)
```

[74]: -1.0

[75]:
```
# Funcion coseno en math
math.cos(math.pi)
```

[75]: -1.0

Arreglos en numpy

Además, `numpy` permite tener arreglos n-dimensionales (`ndarray`, vea la documentación para mayor información) que se comportan de manera similar a los arreglos que podemos encontrar en *Matlab/Octave*. En particular permite crear *vectores* (arreglos unidimensionales) de tamaño n

$$\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix},$$

y *matrices* (arreglos bidimensionales) de tamaño $m \times n$ (m renglones y n columnas)

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \dots & a_{m,n} \end{pmatrix}.$$

[76]:
```
# Generacion de un vector a través de una lista. Tambien se puede usar una tupla
v = np.array([1,2,3,4])
print(v)
type(v)
```

```
[1 2 3 4]
```

[76]: numpy.ndarray

[77]:
```
# Generacion de una matriz a través de una lista bidimensional. Tambien se puede␣
↪usar una tupla
M = np.array([[1,2,3],[4,5,6]])
print(M)
type(M)
```

```
[[1 2 3]
 [4 5 6]]
```

[77]:
```
numpy.ndarray
```

Nota: El acceso a los elementos de los arreglos de `numpy` se puede hacer exactamente igual que para las listas y tuplas. Con ello se puede modificar sus valores mediante la referencia a un elemento o a un subconjunto de elementos usando el operador de asignación (=).

[78]:
```
# Referencia a los elementos del vector
print(v[0]) # Referencia al primer elemento
print(v[1:3]) # Referencia a los elementos de la posición 1 a la 2
print(v[:]) # Referencia a todos los elementos
```

```
1
[2 3]
[1 2 3 4]
```

[79]:
```
# Referencia a los elementos de la matriz
print(M[0][0]) # Elemento en el primer renglon y primera columna M
print(M[1][:]) # Segundo renglon de M
```

```
1
[4 5 6]
```

[80]:
```
# Tipos de datos
print(type(M[0][0]))
print(type(v[0:2]))
```

```
<class 'numpy.int64'>
<class 'numpy.ndarray'>
```

[81]:
```
# Modificamos el valor del primer renglon tercera columna
M[0][2] = -1
print(M)
```

```
[[ 1  2 -1]
 [ 4  5  6]]
```

[82]:
```
# Modificamos los valores de la primera fila, y de la primera y tercera columna
M[0][:3:2] = -5
print(M)
```

```
[[-5  2 -5]
 [ 4  5  6]]
```

Nota: Sin embargo para tener acceso a submatrices como en el caso de *Matlab/Octave*, la clase `array` permite el manejo de los índices a través de un solo par de corchetes, y los índices correspondientes a cada dimensión se separan por comas, de forma análoga a *Matlab/Octave*.

[83]:
```
# Intento de obtener la submatriz de todas las filas de M, y de la segunda y␣
↪tercera columna de M
M[:][1:3]
```

[83]:
```
array([[4, 5, 6]])
```

[84]:
```
# Submatriz de M de todas las filas de M, y de la segunda y tercera columna de M
M[:,1:3]
```

[84]:
```
array([[ 2, -5],
       [ 5,  6]])
```

[85]:
```
# Se modifican todos los valores en la submatriz con un arreglo del mismo tamaño
M[:,1:3] = np.array([[1,2],[3,4]])
M
```

[85]:
```
array([[-5,  1,  2],
       [ 4,  3,  4]])
```

Funciones para generar arreglos y métodos sobre arreglos de `numpy` En el caso de las funciones para generar arreglos de *numpy* contamos con una gran variedad que nos permiten crear matrices con formas particulares como en el caso de *Matlab/Octave*. También es posible acceder a métodos para hacer manipulaciones sobre los elementos del arreglo o consultar algunos de los atributos. Una lista completa de las funciones puede ser consultada en la siguiente liga y los métodos disponibles se puede encontrar en la siguiente liga.

La función `linspace` dentro de `numpy` sirve para crear vectores con elementos espaciados de manera uniforme. La sintaxis es la siguiente

`v = np.linspace(xi,xf,n)`

- La variable `xi` indica el valor del primer elemento del vector.
- La variable `xf` indica el valor del último elemento del vector.
- La variable `n` indica el numero de elementos. Con la sintaxis `v = linspace(xi,xf)` la variable `n` toma el valor de 50.

Así, cada entrada del vector `v` estará dada por:

`v[k] = xi + kh`

para `k` de `0` hasta `n-1` y

`h = (xf-xi)/(n-1)`

[86]:
```
# Ejemplo de linspace
x = np.linspace(-1,2,10)
print(x)
```

```
[-1.         -0.66666667 -0.33333333  0.          0.33333333  0.66666667
  1.          1.33333333  1.66666667  2.        ]
```

La función `arange` dentro de `numpy` también sirve para crear vectores con elementos espaciados de manera uniforme. La sintaxis es la siguiente

`v = np.arange(xi,xf,h)`

- La variable `xi` indica el valor del primer elemento del vector.
- La variable `xf` indica la cota superior valor del último elemento del vector.
- La variable `h` indica el incremento. Con la sintaxis `v = arange(xi,xf)` la variable `h` toma el valor de 1.

Así, cada entrada del vector `v` estará dada por:

`v[k] = xi + kh`

para `k` $\geq$ `0` de tal manera que `xi` $\leq$ `v[k]` $<$ `xf` (`v[k]`$\in$ [`xi`,`xf`)).

[87]:
```
# Para tener el analogo al operador : de Octave se usa arange de numpy
y = np.arange(-1,10,0.5)
print(y)
```

```
[-1.  -0.5  0.   0.5  1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5
  6.   6.5  7.   7.5  8.   8.5  9.   9.5]
```

En los siguientes ejemplos se muestran algunas funciones para crear matrices con formas particulares:

- `np.ones(shape)`. Permite crear un arreglo n-dimensional con todas sus entradas `1.` del tamaño definido por `shape`. Para crear una matriz de tamaño $m \times n$, el argumento `shape` puede ser una lista o una tupla que contenga el número de filas y columnas si tiene dos entradas `(m,n)` o un vector con m entradas si sólo contiene una `(m)` o `(m,)`.
- `np.zeros(shape)`. Permite crear un arreglo n-dimensional con todas sus entradas `0.` del tamaño definido por `shape`. Para crear una matriz de tamaño $m \times n$, el argumento `shape` puede ser una lista o una tupla que contenga el número de filas y columnas si tiene dos entradas `(m,n)` o un vector con m entradas si sólo contiene una `(m)` o `(m,)`.
- `np.eye(m[,n])`. Permite crear una matriz de tamaño $m \times n$ donde los elementos son cero, excepto en la diagonal principal donde son 1. Si sólo se da el argumento `m` se obtendrá una matriz cuadrada de tamaño $m \times m$. El segundo argumento `n` es opcional y si se especifica se tendrá una matriz rectangular de tamaño $m \times n$.

[88]:
```
# Funcion ones
O = np.ones((4,5))
print(O)
```

```
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

[89]:
```
# Funcion zeros
u = np.zeros((9))
print(u)
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

[90]:
```
# Funcion eye
Q = np.eye(5)
print(Q)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Hay diversos métodos definidos para `ndarray`, estos son sólo algunos ejemplos. * `.diagonal()`. Extrae la diagonal principal. * `.sum()`. Permite calcular la suma de todos los elementos en el arreglo. Si es un arreglo n-dimensional, permite hacer la suma sobre alguna dimensión específica obteniendo un arreglo $(n-1)$-dimensional. * `.prod()`. Permite calcular el producto de todos los elementos en el arreglo. Si es un arreglo n-dimensional, permite hacer el producto sobre alguna dimensión específica obteniendo un arreglo $(n-1)$-dimensional.

También se tiene acceso a ciertos atributos que nos dan información sobre el arreglo. Por ejemplo, * `.shape`. Permite saber la forma de un arreglo. Si el arreglo es una matriz arroja una tupla con el número de filas y columnas y si es un vector, regresa una tupla con el número de elementos. * `.size`. Permite saber el número de elementos que hay en el arreglo.

[91]:
```
# Metodo diagonal
O.diagonal()
```

[91]:
```
array([1., 1., 1., 1.])
```

[92]:
```
# Suma de los elementos de un vector
u.sum()
```

[92]:
```
0.0
```

[93]:
```
# Suma de los elementos de una matriz
print(O.sum()) #suma de todos los elementos
print(O.sum(0)) #suma por filas
print(O.sum(1)) #suma por columnas
```

```
20.0
[4. 4. 4. 4. 4.]
[5. 5. 5. 5.]
```

[94]:
```
# Producto de los elementos de un vector
v.prod()
```

[94]:
```
24
```

[95]:
```
# Producto de los elementos de una matriz
print(M.prod()) #producto de todos los elementos
print(M.prod(0)) #producto por filas
```

```
print(M.prod(1)) #producto por columnas
```

```
-480
[-20   3   8]
[-10  48]
```

[96]:
```
# Tamaño de un arreglo
print(u.shape) # Vector
print(O.shape) # Matriz
```

```
(9,)
(4, 5)
```

[97]:
```
# Longitud del arreglo
print(u.size) # Vector
print(O.size) # Matriz
```

```
9
20
```

Operaciones y funciones sobre arreglos de `numpy` En el caso de las operaciones, al aplicar cualquier operador aritmético estas se realizan componente a componente. Esto es, si se tienen dos arreglos de `numpy` `A` y `B` del mismo tamaño, entonces `C` $=$ `A` $\odot$ `B` para $\odot \in \{$ `+`, `-`, `/`, `*`, `%`, `//` $\}$ implica que

`C[idx1[,idx2[,idx3[....,idxN]]]]` $=$

`A[idx1[,idx2[,idx3[....,idxN]]]]` $\odot$ `B[idx1[,idx2[,idx3[....,idxN]]]]`

En particular, para vectores esto implica que:

`C[i]` $=$ `A[i]` $\odot$ `B[i]` para `i` de `0` hasta `n-1`

y para matrices

`C[i,j]` $=$ `A[i,j]` $\odot$ `B[i,j]` para `i` de `0` hasta `m-1` y `j` de `0` hasta `n-1`.

Para el caso de las potencias, tener la instrucción `A**p` implica que cada entrada del arreglo `A` será elevado a la potencia `p`, esto es `A[i]**p` para vectores y `A[i,j]**p` para matrices.

[98]:
```
# Suma de vectores y multiplicacion por escalar
print(v)
print(v + 2*v)
```

```
[1 2 3 4]
[ 3  6  9 12]
```

[99]:
```
# Multiplicacion de vectores elemento a elemento
print(v*v)
```

```
[ 1  4  9 16]
```

[100]:
```
# Division de vectores elemento a elemento
print(v/(2*v))
```

```
[0.5 0.5 0.5 0.5]
```

[101]:
```
# Elevar cada elemento a una potencia
print(v**3)
```

```
[ 1  8 27 64]
```

Nota: También está permitido hacer otras operaciones con escalares, lo cual tendrá el efecto de aplicar a cada entrada del arreglo esa operación con el escalar como segundo operando.

[102]:
```
# Operaciones con escalares
print(v+3) # Suma
print(v-10) # Restar
print(v%2) # Módulo
print(v/3) # División
```

```
[4 5 6 7]
[-9 -8 -7 -6]
[1 0 1 0]
[0.33333333 0.66666667 1.         1.33333333]
```

Dado que la suma y resta de matrices se define como una operación elemento a elemento, con las implementaciones de esos operadores en el módulo `numpy` se tiene cubiertas esas operaciones. Sin embargo, la multiplicación de matrices para A de tamaño $m \times n$ y B de tamaño $n \times r$ da como resultado es una matriz $C = AB$ de tamaño $m \times r$ con entradas

$$C_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j} \quad i = 1, \ldots, m,\ j = 1, \ldots, r.$$

Nota: Observe que el número de columnas de A debe coincidir con el número de renglones de B.

Entonces cuando se quiere realizar la multiplicación de matrices, este debe ser realizado mediante el método *dot* o el operador *@*.

[103]:
```
# Multiplicacion usual de matrices mediante dot
v.dot(O)
```

[103]:
```
array([10., 10., 10., 10., 10.])
```

[104]:
```
# Multiplicacion usual de matrices mediante la funcion dot
np.dot(v,O)
```

[104]:
```
array([10., 10., 10., 10., 10.])
```

[105]:
```
# Multiplicacion usual de matrices mediante @
v @ O
```

[105]:
```
array([10., 10., 10., 10., 10.])
```

Otra característica es que las funciones matemáticas dentro de este módulo se pueden aplicar directamente a los arreglos, teniendo como resultado un arreglo del mismo tamaño a donde cada entrada se le aplica la función. Por ejemplo, dada la matriz `A`, entonces si se le aplica la función `np.cos()`, tendremos que si `B = np.cos(A)`, entonces `B` tiene entradas `B[i,j] = np.cos(A[i,j])`.

[106]:
```
# Aplicación de funciones a vectores y matrices
print("Aplicamos coseno a un vector\n")
print(np.cos(v))
print("\nAplicamos la raiz cuadrada a una matriz\n")
print(np.sqrt(abs(M)))
```

```
Aplicamos coseno a un vector

[ 0.54030231 -0.41614684 -0.9899925  -0.65364362]

Aplicamos la raiz cuadrada a una matriz

[[2.23606798 1.         1.41421356]
 [2.         1.73205081 2.        ]]
```

Nota: La función `abs` sirve para calcular el valor absoluto de un número. Esta función pertenece a las funciones integradas de *Python*, por lo que se puede aplicar a los tipos de datos numéricos. Aunque también es posible usarlas con los tipos de datos del módulo `numpy`, como se muestra en el ejemplo anterior, también se puede usar la función `np.abs()` dentro de este módulo.

Ejemplos

Ejemplo 2

Cree un vector `vec` que tenga 12 elementos de los cuales el primero sea 5, el incremento sea de 4 y el último elemento sea 49.

Solución: Dado que se da el incremento se puede utilizar la función `np.arange()`. Teniendo en cuenta que esta función no toma el valor final, es necesario aumentar el final en 1 para que 49 aparezca en el arreglo.

[107]:
```
vec = np.arange(5,50,4)
print("vec:",vec)
```

```
vec: [ 5  9 13 17 21 25 29 33 37 41 45 49]
```

A partir de `vec` cree los siguientes vectores:

1. Un vector (`vecA`) que tenga 7 elementos. Los primeros 4 elementos deben ser los primeros 4 elementos del vector `vec`, y los últimos 3 deben ser los últimos 3 elementos del vector `vec`.

Solución: Aquí podemos utilizar la función `np.append()` para agregar al primer vector `vec[0:4]` el segundo vector `vec[vec.size-3:]` y así construir `vecA`.

[108]:
```
vecA = np.append(vec[0:4],vec[vec.size-3:])
print("vecA:",vecA)
```

```
vecA: [ 5  9 13 17 41 45 49]
```

2. Un vector (nombrelo `vecB`) que contenga todos los elementos con índice impar de `vec`.

Solución: Los índices impares son 1,3,...,11. Para determinar los elementos con índice impar de `vec` esto se puede hacer a través de los slices indicando el inicio en el índice 1, dejando sin especificar el final para recorrer hasta el final y haciendo incrementos de 2.

[109]:
```
vecB = vec[1::2]
print("vecB:",vecB)
```

```
vecB: [ 9 17 25 33 41 49]
```

3. Un vector (nombrelo `vecC`) que contenga todos los elementos con índice par de `vec`.

Solución: Los índices impares son 0,2,...,10. Para determinar los elementos con índice impar de `vec` esto se puede hacer a través de los slices donde el inicio no se especifica (puede ser 0) para empezar desde el inicio, dejando sin especificar el final para recorrer hasta el final y haciendo incrementos de 2.

[110]:
```
vecC = vec[::2]
print("vecC:",vecC)
```

```
vecC: [ 5 13 21 29 37 45]
```

Ejemplo 3

Muestre que la suma de la serie infinita

$$\sqrt{12}\sum_{n=0}^{\infty}\frac{(-3)^{-n}}{2n+1}$$

converge a π. Haga esto calculando las sumas:

1. $\sqrt{12}\sum_{n=0}^{10}\frac{(-3)^{-n}}{2n+1}$.
2. $\sqrt{12}\sum_{n=0}^{20}\frac{(-3)^{-n}}{2n+1}$.
3. $\sqrt{12}\sum_{n=0}^{50}\frac{(-3)^{-n}}{2n+1}$.

Para cada parte cree un vector n en el cual el primer elemento sea 0, el incremento sea 1 y el último término sea 10, 20 ó 50. Luego, use operaciones elemento a elemento para crear un vector cuyos elementos sean $\frac{(-3)^{-n}}{2n+1}$. Finalmente, use el método `.sum()` para sumar los términos de la serie y multiplique el resultado por $\sqrt{12}$. Compare los valores obtenidos en los incisos 1., 2. y 3. con el valor de π del módulo `numpy`.

Solución: Para calcular la suma es necesario calcular cada uno de los términos. Luego como n toma valores enteros se usa `np.arange(n+1)` para generar un vector con los enteros del `0` hasta `n` que

se guardan en `t_n`. Así, para calcular el término n-esimo de la suma sólo es necesario calcular la expresión $\dfrac{(-3)^{-n}}{2n+1}$ a cada entrada de `t_n`, que se hace usando las operaciones elemento a elemento con el vector. Una vez teniendo los términos calculados sólo se hace la suma de todas las entradas con el método `.sum()` y se multiplica por $\sqrt{12}$. Cambiando los valores de `n` por `10`, `20` y `50`, se obtienen los resultados para los incisos 1., 2. y 3., respectivamente. Conforme más términos se calculen, veremos que el error será menor.

[111]:
```
n = 10
t_n = np.arange(n+1)
t_n = (-3.0)**(-t_n) / (2.0*t_n + 1)
pi_approx = np.sqrt(12)*t_n.sum()
print(f"La aproximacion de {np.pi} con {n+1} terminos de la suma es:␣
 ↪{pi_approx}")
print(f"El error absoluto es: {np.fabs(pi_approx - np.pi):2.2e}")
```

```
La aproximacion de 3.141592653589793 con 11 terminos de la suma es:
3.141593304503081
El error absoluto es: 6.51e-07
```

Ejemplo 4

Considere las siguientes matrices

$$A = \begin{pmatrix} 2 & 4 & -1 \\ 3 & 1 & -5 \\ 0 & 1 & 4 \end{pmatrix} \quad B = \begin{pmatrix} -2 & 5 & 0 \\ -3 & 2 & 7 \\ -1 & 6 & 9 \end{pmatrix} \quad C = \begin{pmatrix} 0 & 3 & 5 \\ 2 & 1 & 0 \\ 4 & 6 & -3 \end{pmatrix}$$

1. ¿Se cumple que $AB = BA$? 2. ¿Se cumple que $A(BC) = (AB)C$? 3. ¿Se cumple que $(AB)^\intercal = B^\intercal A^\intercal$? 4. ¿Se cumple que $(A+B)^\intercal = A^\intercal + B^\intercal$?

Solución: Aquí se definen las matrices, se usa el operador `@` para realizar la multiplicación de matrices y el atributo `.T` para calcular la transpuesta de una matriz ($A^\intercal \equiv$`A.T`).

[112]:
```
A = np.array([[2,4,-1],[3,1,-5],[0,1,4]])
B = np.array([[-2,5,0],[-3,2,7],[-1,6,9]])
C = np.array([[0,3,5],[2,1,0],[4,6,-3]])
print(f"La pregunta 1 no se cumple ya que" \
      f"\n\n AB = \n{A@B}\n\n {chr(0x2260)} \n\nBA = \n{B@A}")
print(f"\nLa pregunta 2 si se cumple ya que" \
      f"\n\n A(BC) = \n{A@(B@C)}\n\n {chr(0x003D)} \n\nA(BC) = \n{(A@B)@C}")
print(f"\nLa pregunta 3 si se cumple ya que" \
      f"\n\n (AB)^T = \n{(A@B).T}\n\n {chr(0x003D)} \n\nB^T(A^T) = \n{B.T@A.T}")
print(f"\nLa pregunta 4 si se cumple ya que" \
      f"\n\n (A+B)^T = \n{(A+B).T}\n\n {chr(0x003D)} \n\nA^T + B^T = \n{A.T+B.
 ↪T}")
```

```
La pregunta 1 no se cumple ya que

 AB =
```

```
[[-15  12  19]
 [ -4 -13 -38]
 [ -7  26  43]]

!=

BA =
[[ 11  -3 -23]
 [  0  -3  21]
 [ 16  11   7]]

La pregunta 2 si se cumple ya que

 A(BC) =
[[ 100   81 -132]
 [-178 -253   94]
 [ 224  263 -164]]

 =

A(BC) =
[[ 100   81 -132]
 [-178 -253   94]
 [ 224  263 -164]]

La pregunta 3 si se cumple ya que

 (AB)^T =
[[-15  -4  -7]
 [ 12 -13  26]
 [ 19 -38  43]]

 =

B^T(A^T) =
[[-15  -4  -7]
 [ 12 -13  26]
 [ 19 -38  43]]

La pregunta 4 si se cumple ya que

 (A+B)^T =
[[ 0  0 -1]
 [ 9  3  7]
 [-1  2 13]]

 =
```

```
A^T + B^T =
[[ 0  0 -1]
 [ 9  3  7]
 [-1  2 13]]
```

Programación

El concepto más fundamental de las *ciencias de la computación* es el concepto de algoritmo. Informalmente, un **algoritmo** es un conjunto de pasos que define cómo hay que realizar una tarea. Para que una máquina como una computadora pueda llevar a cabo una tarea, es preciso diseñar y representar un algoritmo de realización de dicha tarea y en una forma que sea compatible con la máquina. A la representación de un algoritmo se la denomina **programa**. El proceso de desarrollo de un programa, de codificarlo en un formato compatible con la máquina y de introducirlo en una máquina se denomina **programación**.

El estudio de los algoritmos comenzó siendo un tema del campo de las matemáticas. De hecho, la búsqueda de algoritmos fue una actividad de gran importancia para los matemáticos mucho antes del desarrollo de las computadoras actuales. El objetivo era determinar un único conjunto de instrucciones que describiera cómo resolver todos los problemas de un tipo concreto. Uno de los ejemplos mejor conocidos de estas investigaciones pioneras es el algoritmo de división para el cálculo del cociente de dos números de varios dígitos. Otro ejemplo es el *algoritmo de Euclides* descubierto por este matemático de la antigua Grecia que permite determinar el máximo común divisor de dos números enteros positivos.

Una vez que se ha encontrado un algoritmo para llevar a cabo una determinada tarea, la realización de esta ya no requiere comprender los principios en los que el algoritmo está basado. En lugar de ello, la realización de la tarea se reduce al proceso de seguir simplemente las instrucciones proporcionadas. Por ejemplo, podemos emplear el algoritmo de división para calcular un cociente o el algoritmo de Euclides para hallar el máximo común divisor sin necesidad de entender por qué funciona el algoritmo. En cierto sentido, la inteligencia requerida para resolver ese problema está codificada dentro del algoritmo.

Estructuras de control

Se han definido tres estructuras o constructores para un programa o algoritmo estructurado. La idea es que un programa debe estar hecho de una combinación de solo estas tres estructuras: **secuencia**, **decisión** y **repetición**. Se ha demostrado que no es necesario ningún otra estructura. Usar solo estas construcciones hace que un programa o un algoritmo sea fácil de *comprender*, *depurar* o *cambiar*.

- *Secuencia.* Un algoritmo, y eventualmente un programa, es una secuencia de instrucciones, que puede ser una instrucción simple o cualquiera de las otras dos estructuras.
- *Decisión.* Algunos problemas no se pueden resolver con solo una secuencia de instrucciones simples. Algunas veces se necesita probar una condición. Si el resultado de la prueba es verdadero, se sigue una secuencia de instrucciones: si es falso, se sigue una secuencia diferente de instrucciones.
- *Repetición.* En algunos problemas, se debe repetir la misma secuencia de instrucciones. Manejamos esto con la repetición o ciclo (bucle).

Estructuras de decisión (condicionales) [`if`, `else`, `elif`]

Al igual que en otros lenguajes de programación, en *Python* se pueden utilizar las estructuras de control que nos permiten escribir un programa que no sea sólo una secuencia de instrucciones. Como se menciono antes, las estructuras de decisión o condicionales permiten ejecutar una secuencia específica de instrucciones cuando se cumplen ciertas condiciones. En particular, las palabras reservadas para las estructuras de decisión en *Python* son: `if`, `else` y `elif`.

La estructura más básica consiste en usar sólo `if`, y esto se da cuando sólo se quiere evaluar una condición. La sintaxis es la siguiente.

```
if condicion:
    # Secuencia de instrucciones
```

En el siguiente ejemplo se muestra el uso de `if`, al desplegar el mensaje `"La contraseña es correcta"` sólo cuando las cadenas son iguales.

[113]:
```
password = "pass"
intento = input("Dame la constraseña: ")
if password == intento:
    print("La contraseña es correcta")
```

```
Dame la constraseña: pass
La contraseña es correcta
```

La segunda forma es usar `if` junto con `else`, mediante la siguiente sintaxis,

```
if condicion:
    # Secuencia de instrucciones
else:
    # Secuencia de instrucciones
```

donde la secuencia de instrucciones después de `else` se ejecutara automáticamente cuando la condición evaluada en `if` no se cumpla. Continuando con el ejemplo de la contraseña, se puede mandar un mensaje al usuario para que sepa cuando se ha equivocado

[114]:
```
password = "pass"
intento = input("Dame la constraseña: ")
if password == intento:
    print("La contraseña es correcta")
else:
    print("La contraseña no es correcta. Intentalo nuevamente")
```

```
Dame la constraseña: passs
La contraseña no es correcta. Intentalo nuevamente
```

Cuando es necesario evaluar múltiples condiciones es necesario utilizar las estructuras anteriores junto con `elif` mediante la sintaxis

```
if condicion1:
    # Secuencia de instrucciones
elif condicion2:
    # Secuencia de instrucciones
```

```
#...
elif condicionN:
    # Secuencia de instrucciones
else:
    # Secuencia de instrucciones
```

Por ejemplo, se quiere crear un programa que dado un número x diga si este es positivo, negativo o cero. Dado que hay tres casos posibles hay que tener tres condiciones: $x < 0$, $x = 0$ ó $x > 0$. Utilizando la ley de la tricotomía, podemos omitir una y ponerla después de `else` cuando no se cumpla las otras 2. En este caso se omitirá la condición $x = 0$.

[115]:
```
x = float(input("Dame un número: "))
if x > 0:
    print(f"{x} es positivo")
elif x < 0:
    print(f"{x} es negativo")
else:
    print("Es cero")
```

```
Dame un número: 5.6
5.6 es positivo
```

Notas: 1. **Es muy importante la indentación para indicar donde está actuando la estructura de control, ya que en *Python* no se usa ningún carácter o delimitador para este fin**. 2. Las condiciones deben ser un valor booleano (tipo `bool`), el cual se puede obtener mediante operaciones booleanas y relacionales o mediante alguna función o método que regrese valores booleanos. 3. Es posible anidar las estructuras de control u omitir la parte de `else`. Esto dependerá del problema a resolver y el algoritmo implementado.

Ejemplos

Ejemplo 5 (calculo de raíces de una ecuación cuadrática mediante fórmula general) Considérese la ecuación de segundo grado

$$ax^2 + bx + c = 0$$

para la cual se quieren calcular sus raíces reales mediante el uso de la formula general

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Por lo tanto se debe clasificar las raíces a partir del discriminante $D = b^2 - 4ac$ en los siguientes casos:

- $D > 0$, se tienen dos raíces reales distintas.
- $D = 0$, se tienen una raíz real (de multiplicidad 2).
- $D < 0$, no hay raíces reales

Para ello se construirá un programa que permita calcular las raíces, considerando el caso adicional cuando $a = 0$, es decir, cuando se tenga una ecuación lineal.

Programa para clasificar las raíces Se crea un programa que pide valores de los coeficientes de la ecuación cuadrática a, b y c, se calcula el discriminante y a partir de este se hace la clasificación para mostrar el resultado al usuario

[116]:
```
# import math #Es necesario este modulo para cargar las funciones. Descomentar
 ↪si no se ha cargado previamente.
# Se piden los coeficientes
a = float(input('Dame el valor del coeficiente del termino cuadratico '))
b = float(input('Dame el valor del coeficiente del termino lineal '))
c = float(input('Dame el valor del coeficiente del termino constante '))

# << Procedimiento >>
# Se calculan, clasifican y muestran las raices en cada caso.
if a == 0:
    x1 = -c/b
    print(f'La ecuacion es lineal y tiene la raiz {x1}')
else:
    D = b**2 - 4*a*c
    if D > 0:
        x1,x2 = (-b + math.sqrt(D))/(2*a),(-b - math.sqrt(D))/(2*a)
        print(f'La ecuación tiene dos raíces reales x_1 ={x1} y x_2 ={x2}')
    elif D == 0:
        x1 = -b/(2*a)
        print(f'La ecuacion tiene las raices repetidas {x1}')
    else:
        print('No hay raices reales')
```

```
Dame el valor del coeficiente del termino cuadratico 1
Dame el valor del coeficiente del termino lineal -4
Dame el valor del coeficiente del termino constante -21
La ecuación tiene dos raíces reales x_1 =7.0 y x_2 =-3.0
```

Ejemplos

- **Caso lineal**. Consideramos el ejemplo

$$6x - 12 = 0$$

  cuya raíz es x = 2.

- **Raíces reales diferentes**. Consideremos el ejemplo

$$x^2 - 4x - 21 = 0$$

  cuyas raíces son x = 7 y x = -3.

- **Raíces reales repetidas** Consideremos el ejemplo

$$4x^2 - 12x + 9 = 0$$

  cuya raíz es x = 1.5.

- **Raíces complejas** Consideremos el ejemplo

$$3x^2 + 1 = 0$$

que no tiene raíces reales.

Estructuras de repetición (ciclos o bucles) [`for`,`while`]

Un ciclo o bucle, en programación, es una estructura de control que ejecuta repetidas veces una secuencia de instrucciones, hasta que la condición asignada a dicho ciclo deje de cumplirse.

Ciclo `for` El ciclo `for` sirve para ejecutar un número predeterminado de veces una instrucción o un conjunto de instrucciones, es decir, ya se sabe de antemano el número exacto de veces que se tiene que repetir esa secuencia. La sintaxis para el ciclo `for` en *Python* es:

```
for variable in objeto:
    #Codigo
```

donde `objeto` puede ser una lista o una tupla o cualquier objeto que sea *iterable*. Es común usar la función predefinida `range()` en conjunto con los ciclos `for` para generar el objeto sobre el cual se *itera*. De forma análoga a la función `arange` del módulo `numpy`, se puede usar de tres formas

- `range(inicio,fin,incremento)`
- `range(inicio,fin)` La variable *incremento* toma el valor de 1.
- `range(fin)` La variable *inicio* toma el valor de 0 e *incremento* 1.

Los valores de los argumento de entrada deben ser todos enteros. En todos los casos genera un objeto de tipo `range` que se puede interpretar como una lista con los valores comenzando en `inicio` y terminando antes de `fin` ya que el valor `fin` no se considera. Por ejemplo, los siguientes programas muestran los valores generados por `range` en sus distintas formas.

[117]:
```
for i in range(3):
    print(i)
```

```
0
1
2
```

[118]:
```
for i in range(1,3):
    print(i)
```

```
1
2
```

[119]:
```
for i in range(7,1,-2):
    print(i)
```

```
7
5
3
```

Como se dijo antes, también es posible recorrer los elementos en una lista no importando el tipo de dato contenido en esta, ya que la variable de iteración tomará cada uno de los valores en la lista.

[120]:
```
LL = [1,"Persona1",True,2,"Persona2",True,3,"Persona3",False]
for i in LL:
    print(i)
```

```
1
Persona1
True
2
Persona2
True
3
Persona3
False
```

Ejemplos

Ejemplo 6 (suma de todos los elementos en un arreglo) El ejemplo consiste en sumar todos los elementos en un arreglo. Considérese el arreglo `x` de tamaño `n` con componentes `x[k]` (`k` desde `0` hasta `n-1`), y se quiere calcular la suma de todos sus elementos, esto es,

```
s = x[0] + x[1] + ... + x[n-1]
```

Dado que la suma es una operación binaria, se necesita ir sumando de dos en dos y acumular los resultados parciales en `s` para calcular la suma de todos los elementos, esto es

```
s = x[0] + x[1]
s = s + x[2]
s = s + x[3]
...
s = s + x[n-1]
```

Si se considera al inicio `s = 0`, se puede cambiar la instrucción `s = x[0] + x[1]` por

```
s = 0
s = s + x[0]
s = s + x[1]
```

que de forma general, sin contar la instrucción `s = 0` se pueden reescribir de la siguiente manera

```
s = s + x[k]
```

para `k` desde `0` hasta `n-1`. Es por esto que un ciclo `for` es una buena opción para abordar este problema, ya que se conoce exactamente cuantos elementos sumar y que operación debemos repetir.

[121]:
```
# La lista debe ser ingresada con la sintaxis de Python. por ejemplo, [1,2,3,4,5]
x = eval(input("Dame una lista de elementos: "))
s = 0
for k in range(len(x)):
    s = s + x[k]
```

```
print(f"La suma de los elementos de la lista es: {s}")
```

```
Dame una lista de elementos: [1,2,3,4,5]
La suma de los elementos de la lista es: 15
```

La forma análoga usando el recorrido de cada uno de los elementos en el arreglo y sin el uso de índices.

[122]:
```
# La lista debe ser ingresada con la sintaxis de Python. por ejemplo, [1,2,3,4,5]
x = eval(input("Dame una lista de elementos: "))
s = 0
for elem in x:
    s = s + elem
print(f"La suma de los elementos de la lista es: {s}")
```

```
Dame una lista de elementos: [1,2,3,4,5]
La suma de los elementos de la lista es: 15
```

Nota: Como ya se vio anteriormente, esto está implementado en el método `.sum()` para los arreglos de tipo `ndarray` de `numpy`. Pero también está implementado para listas y tuplas mediante la función `sum`. Por ejemplo,

[123]:
```
print(sum([1,2,3,4,5]))
print(sum((7,8,9,10)))
```

```
15
34
```

Ciclo `while` Contrario al ciclo `for`, el ciclo `while` se usa en situaciones en las cuales no se sabe con precisión el número de veces que una instrucción o una secuencia de instrucciones debe ser ejecutado. Por lo tanto, es necesario establecer una condición que permita repetir una secuencia de instrucciones hasta que la condición se deje de cumplir. La sintaxis para el ciclo `while` en *Python*

```
while condicion:
    #Codigo
```

Es posible usar operadores aritméticos mezclados con el operador de asignación para cuando se hacen actualizaciones del valor de una variable mediante operaciones aritméticas. Esto sirve para acortar las expresiones. Un ejemplo de su uso es cuando en los ciclos `while` se quiere incrementar el valor de la variable de iteración, por lo general se usa, `var_iter = var_iter + incremento`, esto se puede representar de manera corta como `var_iter += incremento`. A continuación se da una lista de estos operadores y su significado

Operador	Uso	Expresión análoga
+=	a+=b	a = a + b
-=	a-=b	a = a - b
*=	a*=b	a = a * b
/=	a/=b	a = a / b
%=	a%=b	a = a% b
//=	a//=b	a = a // b

Operador	Uso	Expresión análoga
`**=`	`a**=b`	`a = a ** b`

El siguiente ejemplo muestra el uso del ciclo `while`, en el cual se usa la variable `i` como parte de la condición. El programa sólo imprime los valores que toma la variable `i` a lo largo de la ejecución del ciclo. Observe que esta variable se inicializa con el valor `0` para que la condición `i<10` se cumpla y entre al ciclo. Posteriormente dentro del ciclo se modifica el valor de `i` para que cuando esta variable tome el valor de `10` la condición se deje de cumplir y salga del ciclo.

[124]:
```
i = 0
while i < 10:
    print(i)
    i += 1
```

```
0
1
2
3
4
5
6
7
8
9
```

El ciclo `while` es la estructura de repetición más general, por lo que es posible escribir un programa en donde se usa el ciclo `for` con `while`, pero no en todas los casos se puede hacer al revés. Tomando el ejemplo de la suma de los elementos del arreglo, este se puede reescribir como:

[125]:
```
# La lista debe ser ingresada con la sintaxis de Python. por ejemplo, [1,2,3,4,5]
x = eval(input("Dame una lista de elementos: "))
s = k = 0
while k < len(x):
    s += x[k]
    k += 1
print(f"La suma de los elementos de la lista es: {s}")
```

```
Dame una lista de elementos: [1,2,3,4,5]
La suma de los elementos de la lista es: 15
```

Nota: Cuando se usa un ciclo `while` es importante cuidar que la condición se deje de cumplir en algún momento, ya que si no pasa esto, el programa entrará en un ciclo infinito. En algunos casos se puede usar la combinación de teclas `ctrl+c` y en el caso de la libretas la opción del menú `Kernel`→`Interrupt`, para interrumpir el proceso.

Ejemplo 7 (búsqueda lineal) El problema de búsqueda consiste en que dado un arreglo `L` se determine si un elemento `x` se encuentra o no dentro del arreglo. El algoritmo más simple para implementar la búsqueda consiste en comparar cada elemento en `L` con `x` hasta encontrarlo o recorrer

toda la lista en caso contrario. Es por ello que se utiliza un ciclo `while` para implementar la búsqueda lineal.

Dado que se va a comparar con cada elemento en el arreglo, se usa un índice `i` que debe tomar los valores de `0` hasta `n-1`, con `n = len(L)` el tamaño del arreglo. Para cada `i` se debe hacer la comparación `L[i] == x` **mientras** `i < n` (ya que son los valores válidos para los índices) y en dado caso que se cumpla parar, ya que significaría que `x` está en `L`. Esto se puede implementar de la siguiente manera:

[126]:
```
# La lista debe ser ingresada con la sintaxis de python. por ejemplo, [1,2,3,4,5]
L = eval(input("Dame una lista de elementos: "))
x = eval(input("Dame el elemento que quieres buscar dentro de la lista: "))
i,n = 0,len(L)
while i < n:
    if L[i] == x:
        print(f"El elemento {x} se encuentra en el indice {i}")
        break
    i += 1
if i == n:
    print(f"El elemento {x} no se encuentra en el arreglo")
```

```
Dame una lista de elementos: [1,2,3,4,5]
Dame el elemento que quieres buscar dentro de la lista: 3
El elemento 3 se encuentra en el indice 2
```

donde `break` es una palabra reservada que permite romper (detener) un ciclo.

Sin embargo, es posible incorporar la condición del `if` dentro de la condición del ciclo `while`. Si no se cumple la condición `L[i] == x` se procede a incrementar el índice `i` en `1`, lo que es equivalente a incrementar el índice `i` en `1` **mientras** `i<n` **y** `L[i]!=x`. Esto también implica que si `L[i] == x` la condición

```
i < n and L[i] != x:
```

se deje de cumplir, por tanto es la condición que sirve para romper el ciclo tanto para el caso en que se encuentra el elemento `x` en `L`, con `i` tomando un valor menor que `n`, como en el caso contrario, donde `i` tomará el valor de `n`.

[127]:
```
# La lista debe ser ingresada con la sintaxis de python. por ejemplo, [1,2,3,4,5]
L = eval(input("Dame una lista de elementos: "))
x = eval(input("Dame el elemento que quieres buscar dentro de la lista: "))
i,n = 0,len(L)
while i < n and L[i] != x:
    i += 1

if i < n:
    print(f"El elemento {x} se encuentra en el indice {i}")
else:
    print(f"El elemento {x} no se encuentra en el arreglo")
```

```
Dame una lista de elementos: [1,2,3,4,5]
```

```
Dame el elemento que quieres buscar dentro de la lista: 8
El elemento 8 no se encuentra en el arreglo
```

Scripts y funciones

Podemos crear scripts y funciones para hacer tareas especificas. Los **scripts** son simplemente una secuencia de instrucciones que se van ejecutando línea por línea para realizar una tarea específica, es decir, esto corresponde a la definición de algoritmo. Por otro lado, las **funciones** también son secuencias de instrucciones que realizan tareas específicas, pero tienen la ventaja de que se pueden reutilizar como **subprogramas** dentro de otros scripts o funciones. En ambos casos, se guardan en archivos `.py`. Para poder utilizarlas es necesario cargar en memoria los archivos mediante el comando `run`. En una *interfaz de desarrollo integrado* (IDE) como *Spyder*, se tienen editores que permiten crear, modificar y cargar archivos `.py` de manera gráfica.

En el caso de *Jupyter*, un script o función se pueden escribir en una celda y la evaluación de esta celda permite cargar en memoria o ejecutar un script. En particular, una libreta de jupyter funciona como un script que depende de la ejecución secuencial (o no) de las celdas.

Funciones

Para definir funciones se utiliza la siguiente sintaxis,

```
def nombre_funcion(argumentos_entrada):
    # Cuerpo de la funcion
    return argumentos_salida
```

Notas

- Usamos la palabra reservada `def` para denotar que estamos definiendo una función en python.
- Los argumentos de entrada van separados por `,`.
- Para regresar uno o varios valores tenemos que usar la palabra reservada `return` seguida de las variables de salida separadas por `,`.
- Si la función no regresa nada se omite el uso de `return`.
- Los nombres de las funciones siguen las mismas reglas que para el caso de los nombres de variables.
- **Es muy importante la indentación para indicar donde finaliza la definición de la función, ya que no se usa ningún elemento para este fin**.

Argumentos de salida de una función Como se menciono antes, algo importante sobre las funciones en *Python* es que pueden regresar más de un argumento de salida y una vez que la instrucción

```
return arg1[,arg2[,arg3[....,argN]]]
```

sea alcanzada, la ejecución de la función terminará, regresando el valor o valores dados después de `return`. En este caso, los argumentos de salida, pueden ser:

- Un valor específico. Por ejemplo, podemos regresar directamente literales como `3`, `çadena"` o `False`, sin la necesidad de guardarlo en una variable.
- El valor que tenga una variable. Por ejemplo, si dentro de la función se define la variable `x`, y se pone la instrucción `return x`, la función regresará el valor que se le haya asignado a `x`.
- El resultado de la evaluación de una instrucción.

Ejemplos

Ejemplo 8 (función que calcula el cuadrado de un número) Se va a definir una función que dado un número n calcule su cuadrado, es decir, la función debe regresar n^2. Esta función se llamará `cuadrado`, cuyo argumento de entrada será n y como argumento de salida regresará n^2. La definición queda como:

[128]:
```
# Funcion cuadrado. Calcula el cuadrado de un numero.
def cuadrado(num):
    return num**2
```

Ahora se generará un pequeño *script* (o programa) que haga el **llamado** a la función `cuadrado` definida arriba.

[129]:
```
#Script para el uso de cuadrado
n = float(input("Dame un numero para calcular su cuadrado: "))
print(f"{n}^2 = {cuadrado(n)}")
```

```
Dame un numero para calcular su cuadrado: 5
5.0^2 = 25.0
```

Ejemplo 9 (función con varios argumentos de salida) Para este ejemplo se va a crear una función que dados dos números enteros positivos m y n regrese el cociente c y el residuo r que se obtienen de la división de m entre n $\left(\frac{m}{n}\right)$. Luego se utilizaran las salidas de la función para determinar si m es divisible entre n (que significa $r = 0$) o no (que $r \neq 0$)

[130]:
```
# Definicion de la funcion
def divisible(m,n):
    return m//n,m%n
```

[131]:
```
# Script
m = int(input("Dame un numero entero: "))
n = int(input("Dame otro un numero entero: "))
c,r = divisible(m,n)
if r == 0:
    print(f"\n{m} es divisible por {n}. " \
    f"Por tanto {m} es multiplo de {n} ya que {m} = {c} x {n}")
else:
    print(f"\nDebido a que {m} = {c} x {n} + {r}, " \
    f"{m} no es divisible por {n}. ")
```

```
Dame un numero entero: 13
Dame otro un numero entero: 4

Debido a que 13 = 3 x 4 + 1, 13 no es divisible por 4.
```

Definición de funciones a través módulos

El crear módulos sirve para *empaquetar* todas las funciones necesarias para una tarea y puedan ser usadas sin la necesidad de estar cargandolas en memoria una a una. Para crear un **módulo** se usa un archivo `.py` donde se ponen todas las definiciones de las funciones. De esta forma, el módulo se puede cargar a través del comando `import` utilizando el nombre del archivo como nombre del módulo.

Ejemplos

Ejemplo 10 (conversión de decimal a binario y viceversa) Lo que se hará en este ejemplo es implementar la conversión de números enteros no negativos de su representación decimal a binaria y viceversa. En este caso, se definirán las funciones a través de un módulo creado para este fin.

Para el ejemplo se usará el módulo `conversion` que se creo desde una libreta de *Jupyter*. El archivo de la libreta se llama `conversion.ipynb` y el archivo para el módulo se creo a partir de este desde el menú `File` → `Download as` → `Python (.py)`. El módulo contiene dos funciones con sus descripciones.

- `bin2dec` - *Conversión binario a decimal*. En este caso es necesario hacerlo mediante un ciclo `for` ya que cada entrada de la cadena representa el coeficiente de una potencia de 2. El código es el siguiente, los detalles están en el módulo

```
def bin2dec(bin):
    d,p = 0,len(bin)-1
    for i in bin:
        d += int(i)*2**p
        p -=1
    return d
```

- `dec2bin` - *Conversión decimal a binario*. En este caso es necesario hacerlo mediante un ciclo `while` ya que cada hay que hacer divisiones sucesivas por 2 hasta que el número se vuelva 0. El código es el siguiente, los detalles están en el módulo

```
def dec2bin(dec):
    bin = ""
    if dec <= 0:
        return "0"
    else:
        while dec > 0:
            bin = str(dec%2) + bin
            dec //= 2
        return bin
```

Los detalles de estos algoritmos los vimos en la parte de pseudocódigo.

Nota: En el caso de querer cargar un módulo propio dentro de ***Google Colab*** es necesario habilitar la importación de archivos mediante drive. Para ello es necesario evaluar las siguientes celdas antes de importar un módulo propio. Primero, es necesario montar su *drive* al entorno de ejecución para poder buscar el archivo `.py` donde se encuentra el módulo.

[132]:
```
# Descomentar las siguientes lineas sólo si se quiere trabajar en Google Colab
#from google.colab import drive
#drive.mount('/content/drive')
```

Luego, es necesario copiar el archivo al entorno de ejecución. Para ello se necesita copiar la ruta del archivo dentro del *drive*. Por ejemplo, si el archivo `conversion.py` se encuentra en la raíz del *drive*, entonces la instrucción a ocupar es:

```
#!cp /content/drive/MyDrive/conversion.py .
```

Observe que después de la ruta hay un punto (`.`), eso es importante dejarlo, para que lo copie al entorno de ejecución.

[133]:
```
# Descomentar la siguientes lineas
#!cp /content/drive/MyDrive/conversion.py .
```

Una vez hecho esto se procede de forma normal a como se haría en las libretas de *Jupyter*. En las siguientes celdas se muestra un ejemplo de las funciones de conversión dentro del módulo.

[134]:
```
# Importamos el modulo propio
import conversion as conv
```

[135]:
```
# Usamos la conversion de decimal a binario
d = int(input("Dame un numero entero no negativo en representacion decimal: "))
print(f"La representación binaria de {d} es {conv.dec2bin(d)}")
```

```
Dame un numero entero no negativo en representacion decimal: 31
La representación binaria de 31 es 11111
```

[136]:
```
# Usamos la conversion de decimal a binario
b = input("Dame un numero entero no negativo en representacion binaria: ")
print(f"La representación binaria de {b} es {conv.bin2dec(b)}")
```

```
Dame un numero entero no negativo en representacion binaria: 11111
La representación binaria de 11111 es 31
```

Funciones anónimas

En *Python* es posible usar **funciones anónimas**, también conocidas como **funciones lambda**. Una función anónima es una definición de función que no está vinculada a un identificador (*nombre de función*). Estas funciones se definen mediante la palabra reservada `lambda` mediante la siguiente sintaxis

```
lambda var_entrada: expr_salida
```

Notas:

- Las variables de entrada se deben separar por `,`.
- La expresión de salida solo es una, pero se puede usar una tupla para regresar múltiples resultados. **Usar separación por `,` es lo mismo que definirlo en una tupla, ya que estas se definen así**. Por ejemplo, `1,2,3` es equivalente a `(1,2,3)`.

- Se puede asignar a una variable para que esta sirva como su nombre y se utilice como en el caso de las funciones usuales.
- En *Python*, generalmente se usan como argumento para una función de orden superior (una función que toma otras funciones como argumentos). Por ejemplo, estos se utilizan junto con funciones integradas como `filter()`, `map()` y `reduce()`, etc.

Un ejemplo análogo a la función `cuadrado` mediante funciones anónimas es el siguiente:

[137]:
```
cuad = lambda num: num**2
print(cuad(3))
```

```
9
```

Un ejemplo para regresar múltiples expresiones sería:

[138]:
```
# Operadores en Python
oper = lambda n1,n2:(n1+n2,n1-n2,n1*n2,n1/n2,n1%n2,n1**n2,n1//n2) #int(n1/n2) =␣
 ↪ni//n2
print(oper(7,3))
```

```
(10, 4, 21, 2.3333333333333335, 1, 343, 2)
```

El análogo a la forma de definir la función con *def* sería:

```
def oper(n1,n2):
    return n1+n2,n1-n2,n1*n2,n1/n2,n1%n2,n1**n2,n1//n2
```

y los llamados validos correspondientes serían

```
v1,v2,v3,v4,v5,v6,v7 = oper(n1,n2)
v = oper(n1,n2)
```

donde `vi` (`i = 1,...,7`) se les asignaria el valor de la correspondiente operación y `v` sería una tupla con cada entrada la operación correspondiente.

Nota: En general se recomienda que las expresiones que se evaluan en las funciones anónimas sean simples, por lo que cuando sean muy complejas es mejor definir la funciones a través de `def`.

Funciones `filter`, `map` y `reduce` La función `filter()` se emplea para seleccionar algunos elementos particulares de una colección de elementos. La colección utilizada en esta función es un iterador como listas, tuplas, etc.

Los elementos que se pueden seleccionar se basan en alguna restricción predefinida. La sintaxis es la siguiente

```
result = filter(fun,iterador)
```

donde los 2 parámetros de entrada : * `fun`. Una función que define la restricción de filtrado. * `iterador`. Una colección (cualquier iterador como listas, tuplas, etc.)

Por ejemplo, si se quiere extraer todos los números pares dentro de una lista, se puede hacer de la siguiente manera

[139]:
```
L = [10,2,8,7,5,4,3,11,0,1]
r = filter(lambda x : x % 2 == 0,L)
print(list(r))
```

```
[10, 2, 8, 4, 0]
```

Nota. El resultado de estas funciones no es un objeto de tipo `list`, por lo cual se puede aplicar la función predefinida `list()` para convertirlo a este tipo.

La función `map()` se emplea para aplicar una operación específica a cada elemento en una colección. La sintaxis es análoga a la de `filter`, y esta dada como sigue

```
result = map(fun,iterador)
```

donde los 2 parámetros de entrada : * `fun`. Una función que define cómo se realizarán las operaciones en los elementos. * `iterador`. Una colección (cualquier iterador como listas, tuplas, etc.)

Por ejemplo, si se quiere aplicar la función `cuadrado` a todos los números pares dentro de una lista, se puede hacer de la siguiente manera

[140]:
```
L = [10,2,8,7,5,4,3,11,0,1]
r = map(cuadrado,L)
print(list(r))
```

```
[100, 4, 64, 49, 25, 16, 9, 121, 0, 1]
```

La función `reduce()`, como `map()`, se emplea para aplicar una operación a cada elemento en una colección. Sin embargo, su funcionamiento es un poco diferente de la función `map()`. Los siguientes pasos deben ser seguidos por la función `reduce()` para calcular una salida:

1. Aplicar la operación definida en los 2 primeros elementos de la colección y guardar este resultado parcial.
2. Aplicar la operación al resultado parcial y el siguiente elemento dentro de la colección y guardar este nuevo resultado parcial.
3. Repetir el paso dos hasta que no queden más elementos.

La sintaxis es análoga a las dos funciones anteriores, y esta dada como sigue

```
result = reduce(fun,iterador)
```

donde los 2 parámetros de entrada :

- `fun`. Una función que define cómo se realizarán las operaciones.
- `iterador`. Una colección (cualquier iterador como listas, tuplas, etc.)

Esta función se encuentra dentro del módulo `functools`. Por ejemplo, se puede usar está función para reescribir el programa que suma los elementos dentro de un arreglo (ejemplo 6).

[141]:
```
from functools import reduce
L = [1,2,3,4,5,6,7,8,9,10]
r = reduce(lambda u,v : u + v,L)
print(r)
```

```
55
```

Funciones con valores por defecto

Otro punto importante sobre las funciones es que se puede dar valores por defecto a los argumentos de entrada, de tal manera que si no se ingresa un valor a la función aún puede correr con dichos valores. Además esto permite definir valores especificos para ciertos argumentos de entrada utilizando su nombre y sin importar el orden. Esto da mayor flexibilidad en el llamado de las funciones. Además, dado que la asignación de variables es dinámica, entonces las variables pueden ser de cualquier tipo, incluso otras funciones.

Ejemplos

Ejemplo 11 (Calculadora científica) En el siguiente ejemplo se simulará el uso de una calculadora científica sólo para el cálculo de algunas funciones trigonométricas, logaritmos y exponenciales. En este ejemplo se usará el módulo `math` y diccionarios.

Un **diccionario** es una estructura de datos y un tipo de dato (`dict`) en *Python* con características especiales que permite almacenar cualquier tipo de dato como enteros, cadenas, listas e incluso otras funciones y que al igual que las listas es mutable. Además, permite identificar cada elemento por una clave (*Key*), la cual debe ser una cadena (`str`).

Una referencia sobre el uso básico de diccionarios y sobre sus métodos se puede consultar en la siguiente liga.

[142]:
```
def aplicarFuncion(f = lambda x:x,x = 0):
    """Realiza la evaluacion de la funcion 'f' en 'x'."""
    return f(x)

def imprimir(f = "id",x = 0.0,fx = 0.0,pre = 5):
    """Imprime la evaluacion de la funcion 'f' en 'x' ('fx') mostrando 'pre'␣
 ↪decimales'."""
    print(f"{f}({x:6.{pre}f}) = {fx:6.{pre}f}")
```

[143]:
```
import math
Fun = {"sen":math.sin, "cos":math.cos, "tan":math.tan, "exp":math.exp, "log":
 ↪math.log}
f = input("Introduce la función a aplicar (sin, cos, tan, exp, log): ")
x = float(input('Introduce el valor a evaluar en la función: '))
if f.lower() in Fun:
    imprimir(f,x,aplicarFuncion(Fun[f.lower()],x))
else:
    imprimir(x = x,fx = aplicarFuncion(x = x))
```

```
Introduce la función a aplicar (sin, cos, tan, exp, log): sin
Introduce el valor a evaluar en la función: 0
id(0.00000) = 0.00000
```

Ejemplo 12 (derivada de una función en un punto)

Dada una función $f\colon \mathbb{R} \to \mathbb{R}$, se considera el problema de calcular la derivada de la función en x. Recuérdese que la derivada de la función f en x está dada por

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

si el límite existe. Bajo ciertas hipótesis sobre f, se puede probar que

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h)$$

donde $\mathcal{O}(h)$ nos dice que este término es proporcional a h, o análogamente, si tomamos la aproximación

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

el orden del error de la aproximación es cercano al orden de h, por lo tanto, cuanto más pequeño sea h mejor será la aproximación. A la aproximación dada en la última ecuación se le conoce como **fórmula de diferencias progresivas** ó **regresivas**, de acuerdo si $h > 0$ ó $h < 0$, respectivamente. Está fórmula da una manera fácil de generar una aproximación de $f'(x)$ ya que solo bastaría usar la última ecuación para h suficientemente pequeño. El problema es como escoger h para evitar errores de redondeo.

Una manera simple es recurrir a un proceso iterativo al ir variando el valor de h y tomar una sucesión de aproximaciones de la derivada hasta que estas no varíen por más de una cota establecida, es decir, considerando en la iteración $k > 1$

$$h_k = \frac{h_{k-1}}{2} \quad \text{y} \quad f'_k = \frac{f(x+h_k) - f(x)}{h_k}$$

mientras $|f'_k - f'_{k-1}| > 10^{-e}$ para $e > 0$. Una vez que la diferencia entre dos aproximaciones consecutivas sea menor o igual que 10^{-e}, es decir, $|f'_k - f'_{k-1}| \leq 10^{-e}$, el valor de la aproximación será el valor calculado en f'_k.

Para hacer la función que calcule la derivada de una función en un punto es necesario que esta reciba como argumento una función. Para ello no es necesario hacer algo especial dentro de la función pero si en el tipo de argumentos de entrada que recibirá la función. Por el momento, para construir la función supondremos que se da una función a la que denotaremos por la variable `f` y se quiere calcular su derivada en el punto `x`. La función queda dada de la siguiente manera.

[144]:
```
# import numpy as np

def derivada(f,x,tol = 1e-12, maxiter = 500):
    d,h,it = np.inf,0.5,1
    dfx_a = f(x+1) - f(x)
    while abs(d) > tol and it < maxiter:
        dfx = (f(x+h) - f(x)) / h
        d = dfx - dfx_a
        dfx_a = dfx
        h /= 2
        it += 1
```

```
    return dfx,it
```

[145]:
```
f = eval(input("Dame la función: "))
x = eval(input("Dame el punto donde quieres evaluar la derivada: "))
df,_ = derivada(f,x,1e-16)
print(f"f'({x}) = {df}")
```

```
Dame la función: lambda x:x**2
Dame el punto donde quieres evaluar la derivada: 3
f'(3) = 6.000000059604645
```

Recursividad

La recursividad es un método para resolver un problema donde la solución depende de soluciones a instancias más pequeñas del mismo problema. Tales problemas generalmente se pueden resolver de forma iterativa, pero esto necesita identificar e indexar las instancias más pequeñas en el momento de la programación. Por el contrario, la recursividad resuelve tales problemas recursivos mediante el uso de funciones que se llaman a sí mismas desde su propio código. El enfoque puede aplicarse a muchos tipos de problemas.

La definición de una función recursiva tiene uno o más casos base, es decir, instancias para las cuales la función produce un resultado trivial (sin recursividad), y uno o más casos recursivos, es decir, entradas para las cuales el programa se llama a sí mismo. Se mostrará esto a través de algunos ejemplos.

Ejemplos

Ejemplo 13 (Cálculo del factorial) Considérese el cálculo del factorial de un entero positivo n, denotado por $n!$. El factorial se define como el producto de todos los enteros menores o iguales que n, es decir,

$$n! = 1 \cdot 2 \cdot 3 \cdot (n-1) \cdot n = \prod_{k=1}^{n} k, \tag{13.1}$$

tomando adicionalmente que $0! = 1$. De esta manera, el cálculo del factorial se pueden reescribir de manera recursiva de la siguiente manera

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n(n-1)! & \text{si } n > 0. \end{cases} \tag{13.2}$$

donde, para $n = 0$ se tiene el caso base y para $n > 0$ se hace el llamado así mismo pero para un problema más pequeño, en este caso el cálculo de $(n-1)!$. *Python* soporta la recursividad, es decir el llamado de una función a si misma. Usando la forma recursiva dada en (13.2) se tiene

[146]:
```
def factorialRec(n):
    if n <= 0:
        return 1
    else:
        return n*factorialRec(n-1)
```

Usando las ideas del ejemplo de la suma de todas las entradas de un arreglo (ejemplo 6) y la definición dada en la ecuación (13.1), la versión iterativa del problema queda como:

[147]:
```
def factorialIt(n):
    nfact = 1
    for i in range(1,n+1):
        nfact *= i
    return nfact
```

Finalmente se muestra el uso de las funciones antes definidas

[148]:
```
n = int(input("Dame el numero entero para calcular su factorial: "))
print(f"{n}! = {factorialRec(n)} (forma recursiva)")
print(f"{n}! = {factorialIt(n)} (forma iterativa)")
```

```
Dame el numero entero para calcular su factorial: 10
10! = 3628800 (forma recursiva)
10! = 3628800 (forma iterativa)
```

Ejemplo 14 (Sucesión de Fibonacci) Ahora se considera el problema de calcular el n-ésimo número de la sucesión de Fibonacci para $n \geq 0$, el cual esta dado por

$$F_n = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F_{n-1} + F_{n-2} & \text{si } n > 1 \end{cases} \tag{14.1}$$

y cuya definición es claramente recursiva. En este ejemplo hay dos casos base los cuales son para $n = 0$ se tiene que $F_0 = 0$ y para $n = 1$, $F_1 = 1$, y dos casos recursivos para $n > 1$, es decir encontrar los términos $n - 1$ y $n - 2$ de la sucesión. Siguiendo la definición dada por en (14.1), la implementación en *Python* es directa:

[149]:
```
def fibRec(n):
    if n < 1:
        return 0
    elif n == 1:
        return 1
    else:
        return fibRec(n-1) + fibRec(n-2)
```

Este algoritmo para calcular un término de la sucesión de Fibonacci es especialmente malo pues cada vez que se ejecuta la función, la función realizará dos llamadas a sí misma, cada una de las cuales hará a la vez dos llamadas más y así sucesivamente hasta que terminen en 0 o en 1. El ejemplo se denomina *recursión de árbol*, y sus requisitos de tiempo crecen de forma exponencial y los de espacio de forma lineal.

Es por esto que es mejor considerar la fórmula o una forma iterativa. Para hacer la forma iterativa, es importante notar que

$$F_k = F_{k-1} + F_{k-2} \quad \forall k > 1$$

iniciando con $F_0 = 0$ y $F_1 = 1$, es decir, solo es necesario conocer los dos números anteriores para calcular el siguiente. Teniendo en cuenta esto se obtiene el siguiente código:

[150]:
```
def fibIt(n):
    fkm2,fkm1 = 0,1
    for i in range(n):
        fkm1,fkm2 = fkm1 + fkm2,fkm1
    return fkm2
```

Se muestra el uso de las funciones definidas.

[151]:
```
n = int(input("¿Cuál es el término de la sucesión de Fibonacci que quieres␣
 ↪calcular?: "))
print(f"F_{n} = {fibRec(n)} (forma recursiva)")
print(f"F_{n} = {fibIt(n)} (forma iterativa)")
```

```
¿Cuál es el término de la sucesión de Fibonacci que quieres calcular?: 15
F_15 = 610 (forma recursiva)
F_15 = 610 (forma iterativa)
```

La diferencia en el tiempo de ejecución entre ambas implementaciones es muy grande ya que para este caso los requisitos de tiempo crecen de forma exponencial. Para corroborar esto trate de calcular $F_{40}(= 102334155)$ con ambos códigos y tome el tiempo que se tardan en obtener las soluciones, para ello puede usar el script:

[152]:
```
import time
n = 40
inicio = time.time()
fibRec(n)
fin = time.time()
print(f"La implementación recursiva tardó {fin-inicio:2.2f} seg.")

inicio = time.time()
fibIt(n)
fin = time.time()
print(f"La implementación iterativa tardó {fin-inicio:2.2e} seg.")
```

```
La implementación recursiva tardó 24.56 seg.
La implementación iterativa tardó 3.79e-05 seg.
```

Gráficas de funciones

En *Python* es posible realizar gráficas de funciones reales ($f : [a, b] \subset \mathbb{R} \to \mathbb{R}$) y funciones escalares ($f : [a, b] \times [c, d] \subset \mathbb{R}^2 \to \mathbb{R}$), entre otras. Para este fin se usará el módulo `matplolib`, en particular `pyplot` para realizar gráficas en 2D y `mplot3D` para gráficas en 3D. Las funciones y forma de usar son similares a las funciones que se ocupan en *Matlab/Octave*.

Lo primero que se necesita es cargar todos los módulos necesarios para poder crear las gráficas.

[153]:
```
# Cargamos las librerias y paquetes
from matplotlib import pyplot as plt #Graficas 2D
from mpl_toolkits.mplot3d import Axes3D #Graficas 3D
%matplotlib inline
```

Aunque `matplotlib` fue inicialmente concebida para crear gráficos en 2D, después se extendió su uso para gráficas en 3D. La instrucción `%matplolib` es lo que se conoce como una *función mágica* dentro del shell interactivo *IPython* que usa *Jupyter*. Esto permite tener un modo interactivo con las gráficas si se cambia `inline` por `notebook`. Si se omite, en las libretas queda en forma `inline`. Información adicional puede ser encontrada en la siguiente liga.

Gráficas 2D

Para crear gráficas en 2D debemos usar la función `plot` que se encuentra en `pyplot`. En este caso la sintaxis básica es la siguiente:

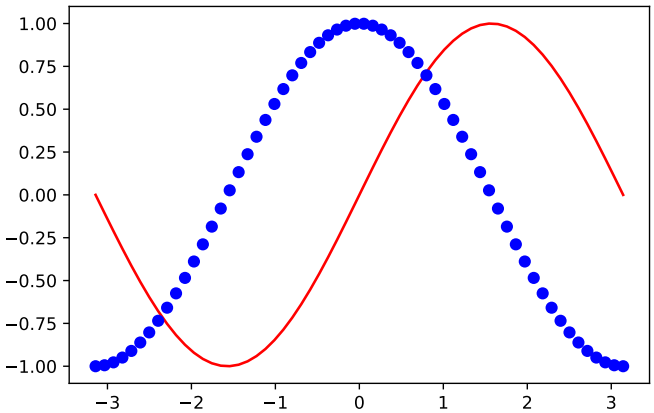
```
plt.plot(x,y,[fmt])
```

donde `x` son las abscisas, `y` son las ordenadas y `fmt` (opcional) corresponde a una cadena que define el formato básico para la apariencia de las líneas. La documentación completa de la función `plot` puede ser consultada en la siguiente liga.

Para mostrar su uso considérese los siguientes ejemplos.

[154]:
```
puntos = int(input('Dame el numero de puntos a graficar: '))
x = np.linspace(-np.pi,np.pi,puntos)
y,z = np.sin(x),np.cos(x)
plt.plot(x,y,'-r')
plt.plot(x,z,'ob')
plt.show()
```

```
Dame el numero de puntos a graficar: 60
```

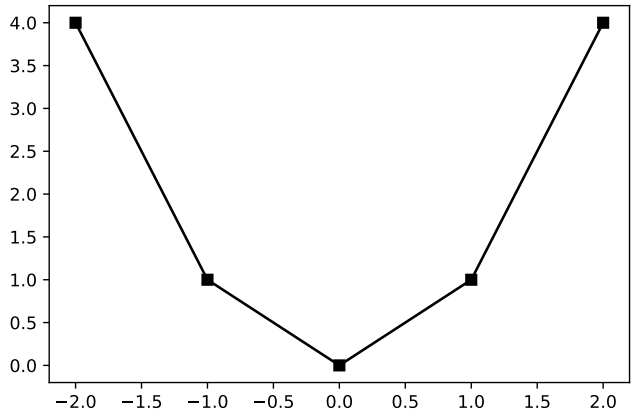


En este caso se usan funciones que se encuentran dentro del módulo `numpy` para generar el conjunto de coordenadas donde se evalúa la función. En particular, se usa `linspace` para crear las abscisas y después se evalúan estas en una función para obtener las ordenadas que definirán el par ordenado que se gráfica mediante `plot`. En este caso es importante poner la instrucción `plt.show()` para mostrar la gráfica.

Se pueden graficar varias curvas en una misma figura al ejecutar la instrucción `plot` con diferentes conjuntos de pares ordenados, como en el ejemplo anterior. Se pueden dar modificaciones en el aspecto de las curvas, como son el color, grosor, marcadores. etc. Todo esto puede ser consultado en la documentación

En los ejemplos se usarán los arreglos (`ndarray`) de `numpy` debido a las funcionalidades que estos proveen, pero es posible utilizar otros tipos de datos, como listas o tuplas, para mandar las coordenadas a la función `plot`. Por ejemplo, usando listas

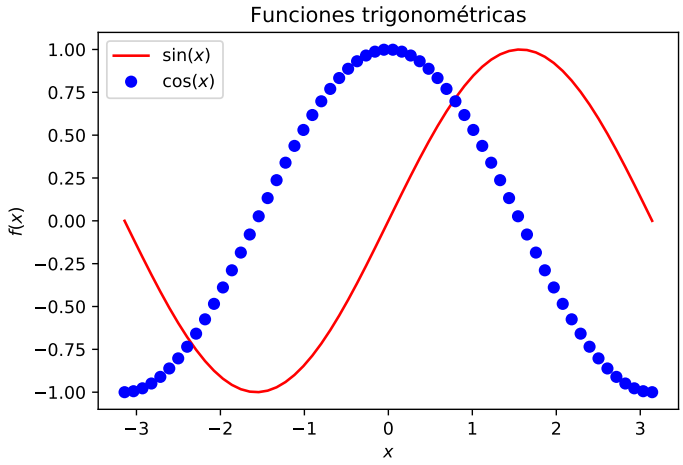
[155]:
```
u = [-2,-1,0,1,2]
v = list(map(lambda x:x**2,u))
plt.plot(u,v,'sk-')
plt.show()
```



También es posible crear etiquetas para los ejes, leyendas y títulos que soportan comandos en *LaTeX*. Estás son propiedades que pertenecen a los ejes, por lo cual es necesario obtener el control sobre los ejes actuales mediante la función `gca()`. Esto se muestra en el siguiente ejemplo

[156]:
```
x = np.linspace(-np.pi,np.pi,puntos)
y,z = np.sin(x),np.cos(x)
plt.plot(x,y,'-r',label='$\sin(x)$')
plt.plot(x,z,'ob',label='$\cos(x)$')
ax = plt.gca() # Se obtiene el control para los ejes actuales
ax.set_xlabel("$x$") # Etiqueta eje x
ax.set_ylabel("$f(x)$") # Etiqueta eje y
ax.set_title("Funciones trigonométricas") # Titulo
```

```
ax.legend(loc = 'upper left') # Leyenda a partir de las etiquetas
plt.savefig('Fig.pdf') # Guardamos la figura en el archivo Fig.pdf
```

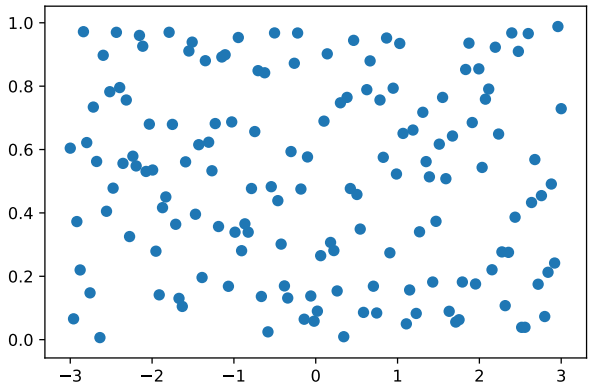


Nota: Es posible guardar la imagen en un archivo mediante la función `savefig`, la cual soporta diferentes tipos de archivos, incluyendo los más comunes como `.pdf`, `.png`, `.jpg`, `.eps`, etc. Cuando se guarda a un archivo, hay que omitir la instrucción `plt.show()`, para que la gráfica se imprima al archivo y no al visualizador de la libreta.

Es posible generar graficas donde solo aparecen puntos dispersos sin lineas mediante la función `scatter`.

[157]:
```
puntos = int(input('Dame el numero de puntos a graficar: '))
x = np.linspace(-3,3,puntos)
y = np.random.rand(puntos)
plt.scatter(x,y)
plt.show()
```

```
Dame el numero de puntos a graficar: 150
```



Gráficas 3D

Para cambiar la perspectiva de los ejes a tres dimensiones se utiliza `Axes3D`, lo cual da el efecto de profundidad. Esto se hace a través de dos instrucciones

```
fig = plt.figure()
ax = Axes3D(fig)
```

donde la primera instrucción crea una figura y la segunda crea los ejes con perspectiva en 3 dimensiones. Para mostrar su uso, se hará un ejemplo con la curva paramétrica

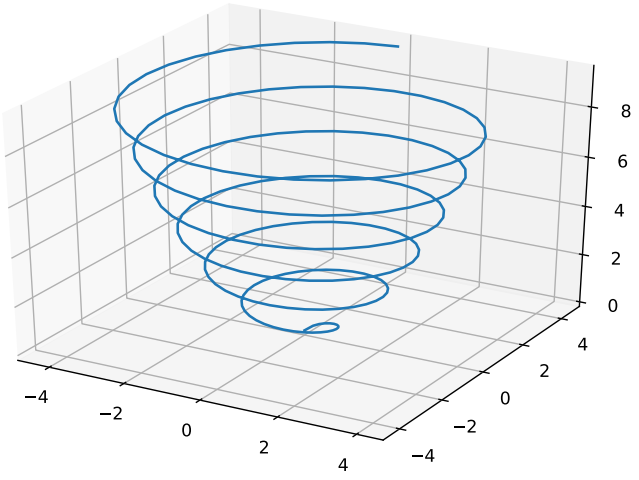
$$x = \sin(2t),$$

$$y = \cos(2t),$$

$$z = \frac{t}{2},$$

con $0 \leq t \leq 6\pi$.

[158]:
```
# Ejemplo de curva paramétrica
fig = plt.figure()
ax = Axes3D(fig)
t = np.linspace(0,6*np.pi,200)
x = np.sqrt(t)*np.sin(2*t)
y = np.sqrt(t)*np.cos(2*t)
z = 0.5*t
plt.plot(x, y, z)
plt.show()
```



La función `plot` en este caso hace la gráfica tomando en cuenta la profundidad y genera una figura en 3D. También es posible hacerlo usando la siguiente instrucción

```
ax = plt.axes(projection = '3d')
```

pero cargando el modulo `mplot3d` directamente

```
from mpl_toolkits import mplot3d
```

Con esta forma, se cambia la perspectiva directamente desde los ejes a través de la opción `projection`, aunque esto ya se cambió en la versión actual de `matplotlib` por la usada en el ejemplo anterior, y ya no se recomienda. De todas maneras, el código para generar la misma figura con este procedimiento desde cero, quedaría de la siguiente manera:

```
import numpy as np #Rutinas numericas
from matplotlib import pyplot as plt #Graficas 2D
from mpl_toolkits import mplot3d #Graficas 3D

ax = plt.axes(projection = '3d')
t = np.linspace(0,6*np.pi,200)
x = np.sqrt(t)*np.sin(2*t)
y = np.sqrt(t)*np.cos(2*t)
z = 0.5*t
ax.plot(x, y, z)
plt.show()
```

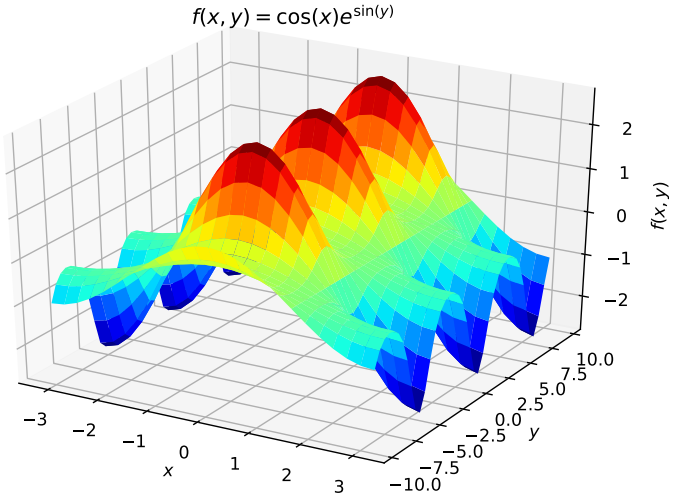
la cual pueden probar en una libreta aparte o desde un archivo *.py*.

También se pueden generar superficies a partir de funciones escalares $f(x,y)\colon \mathbb{R}^2 \to \mathbb{R}$. Para ello se necesita de una malla para tener los puntos de evaluación y generar los pares ordenados en arreglos. En *Python*, para hacer esto (en *Matlab/Octave* es de forma análoga) se puede usar la función `meshgrid` de `numpy`. El código para generar la gráfica de la función

$$f(x,y) = \cos(x)e^{\sin(y)}$$

como una superficie mediante la función `plot_surface` se da a continuación.

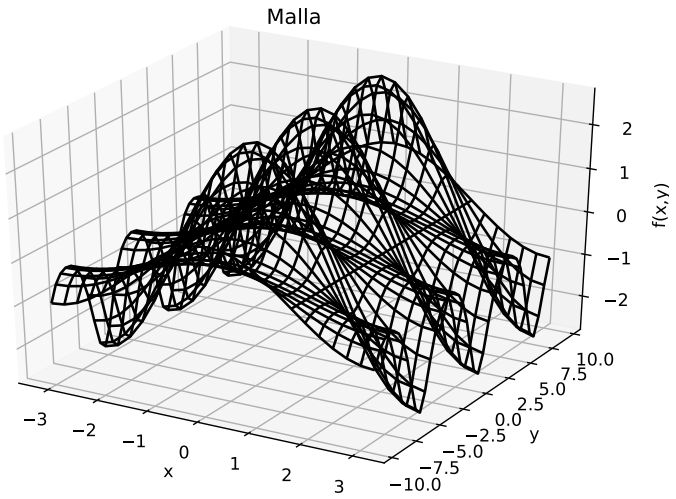
[159]:
```
fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(-np.pi,np.pi,25)
y = np.linspace(-3*np.pi,3*np.pi,40)
X,Y = np.meshgrid(x,y)
Z = np.cos(X)*np.exp(np.sin(Y))
ax.plot_surface(X, Y, Z,cmap='jet')
ax.set_xlabel("$x$")
ax.set_ylabel("$y$")
ax.set_zlabel("$f(x,y)$")
ax.set_title("$f(x,y) = \cos(x)e^{\sin(y)}$")
#plt.savefig("superficie.pdf")
plt.show()
```



El mapa de colores se cambia a través de la opción `cmap` y tienen nombres parecidos a los usados en *Matlab/Octave*. Puede consultar la siguiente liga para los mapas de colores disponibles o la documentación de `matplotlib`.

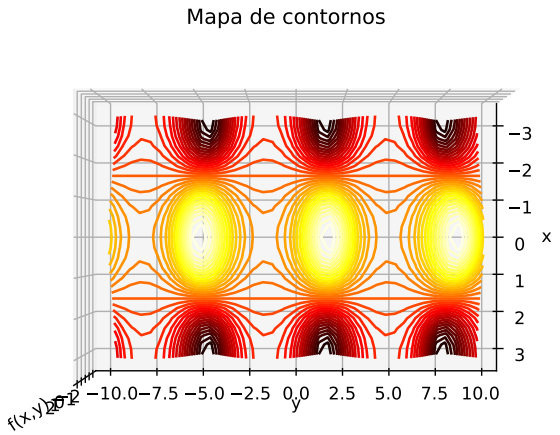
Se pueden generar gráficas sin los parches rellenos, es decir, mostrando solo la malla con el comando `plot_wireframe` donde los colores de las líneas se cambian a través de la opción `color` (valores disponibles en la siguiente liga).

[160]:
```
fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(-np.pi,np.pi,25)
y = np.linspace(-3*np.pi,3*np.pi,40)
X,Y = np.meshgrid(x,y)
Z = np.cos(X)*np.exp(np.sin(Y))
ax.plot_wireframe(X, Y, Z,color='black')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x,y)")
ax.set_title("Malla")
plt.savefig("Malla.pdf")
```



Se pueden realizar mapas de contorno mediante la función `contour3D` y ver la variación de las alturas al cambiar el mapa de colores. Para cambiar la vista inicial de la gráfica se usa `view_init` donde el primer argumento es la *elevación* y el segundo el *azimut*. El *azimut*, es la rotación horizontal alrededor del eje z, medida en grados desde el eje y negativo. Los valores positivos indican la rotación en sentido contrario a las manecillas del reloj del punto de vista. La *elevación* es la rotación vertical del punto de vista en grados. Los valores positivos de *elevación* corresponden a moverse por encima del objeto; los valores negativos corresponden a moverse debajo del objeto.

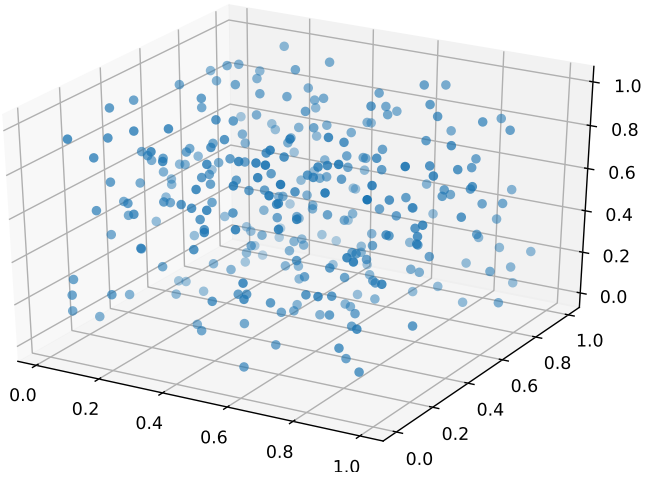
[161]:
```
fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(-np.pi,np.pi,25)
y = np.linspace(-3*np.pi,3*np.pi,40)
X,Y = np.meshgrid(x,y)
Z = np.cos(X)*np.exp(np.sin(Y))
ax.contour3D(X, Y, Z,50,cmap='hot')
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_zlabel("f(x,y)")
ax.set_title("Mapa de contornos")
ax.view_init(90,0) #Vista desde arriba
plt.show()
```



Se pueden crear graficas de nubes de puntos con `scatter3D`

[162]:
```
puntos = int(input('Dame el numero de puntos a graficar: '))
fig = plt.figure()
ax = Axes3D(fig)
x = np.random.rand(puntos)
y = np.random.rand(puntos)
z = np.random.rand(puntos)
ax.scatter3D(x,y,z)
plt.show()
```

```
Dame el numero de puntos a graficar: 300
```



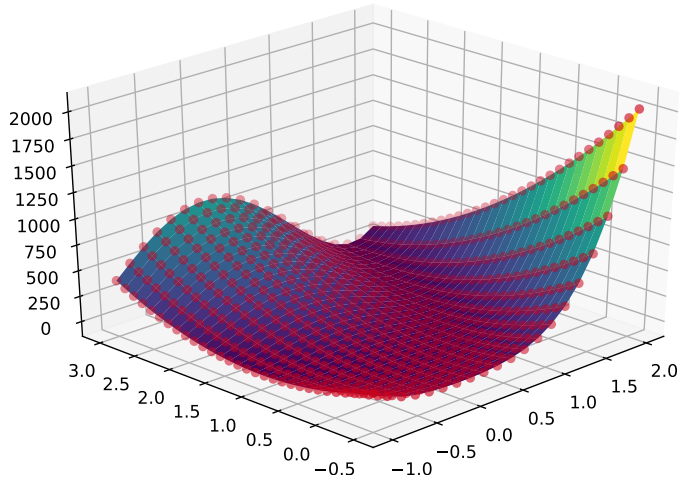
O también es posible combinar varios tipos de gráficas en una sola figura aplicando varias funciones

en un mismo eje. Esto se mostrará para la función de Rosenbrock

$$f(x,y) = (a-x^2)^2 + b(y-x^2)^2,$$

con $a = 1$ y $b = 100$.

[163]:
```
a,b = 1,100
fig = plt.figure()
ax = Axes3D(fig)
x = np.linspace(-1,2,20)
y = np.linspace(-0.5,3,30)
X,Y = np.meshgrid(x,y)
Z = (a - X)**2 + b*(Y-X**2)**2
ax.plot_surface(X,Y,Z,cmap='viridis')
ax.scatter3D(X,Y,Z,color = (0.8,0,0.1,0.7))
ax.view_init(30,-135)
plt.savefig('Rosenbrock.pdf')
```

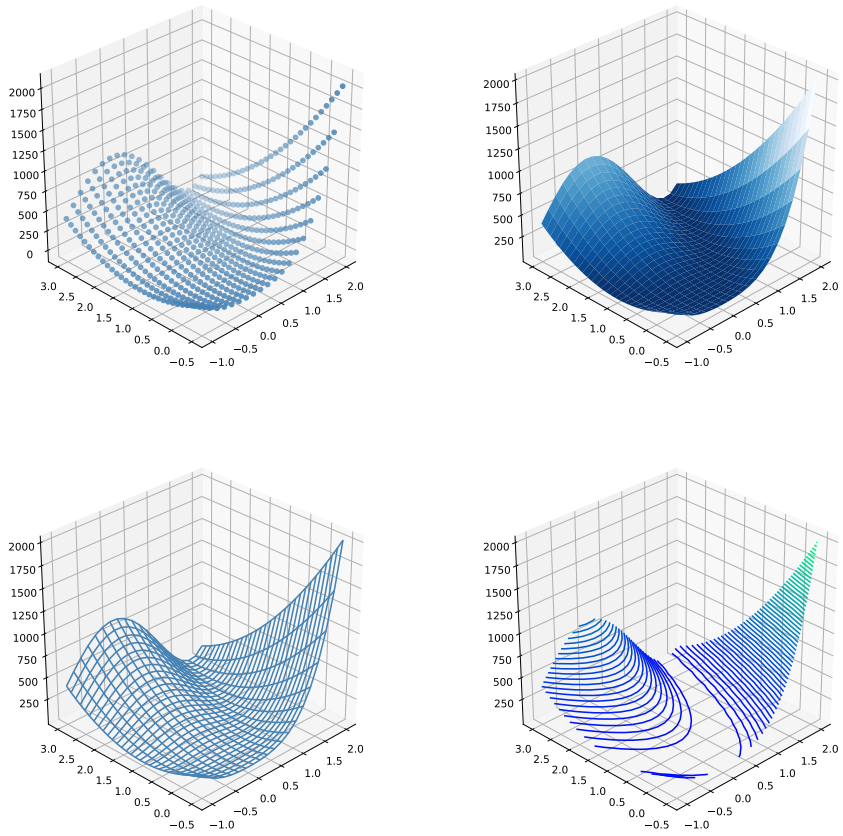


Aquí para los puntos se usa la opción `color` tomando como valor una tupla con cuatro valores entre 0 y 1. Las tres primeras corresponden al color dado en RGB estandarizado (primera rojo, segunda verde y tercera azul) y una cuarta componente para el *canal alfa* (transparencia).

Por último, es posible generar varias gráficas en una misma figura con la función `subplot`. Esto se muestra con el siguiente ejemplo:

[164]:
```
fig = plt.figure(figsize=(15,15)) # Dimensiones de la figura en pulgadas anchura␣
↪y altura.

ax1 = plt.subplot(2,2,1, projection='3d')
ax1.scatter3D(X,Y,Z,color = (.275, .51, .706,1))
ax1.view_init(30,-135)

ax2 = plt.subplot(2,2,2, projection='3d')
```

```
ax2.plot_surface(X,Y,Z,cmap='Blues_r')
ax2.view_init(30,-135)

ax3 = plt.subplot(2,2,3, projection='3d')
ax3.plot_wireframe(X, Y, Z,color='steelblue')
ax3.view_init(30,-135)

ax4 = plt.subplot(2,2,4, projection='3d')
ax4.contour3D(X,Y,Z,50,cmap='winter')
ax4.view_init(30,-135)

plt.show()
```



En este caso, para manejar las diferentes subgráficas se usa la idea de una tabla donde el primer argumento es el numero de renglones, el segundo el número de columnas y el tercero es un índice que índica la celda de la subgráfica actual. Este índice comienza en `1` que corresponde a la esquina superior izquierda y se van recorriendo de izquierda a derecha por renglones. Del ejemplo anterior, la distribución de los índices que se tiene es

1	2
3	4

La distribución de las subgráficas puede ser más compleja, para más ejemplos e información revise esta liga o la documentación de `matplotlib`.

Ejemplos

Ejemplo 15 (Coordenadas de una circunferencia) En este ejemplo se definirá una función para obtener las coordenadas cartesianas de una circunferencia con radio `r` y centro `c`. Para ello usaremos la parametrización de la circunferencia, donde el parámetro será el ángulo α, medido en radianes, entre el semieje positivo x y el segmento que une el origen con el punto sobre la circunferencia, como se muestra en la siguiente figura.

Recordando que en un triángulo rectángulo, como el que se muestra en la figura

$$\sin(\alpha) = \frac{\text{cateto opuesto}}{\text{hipotenusa}} = \frac{y}{h} \qquad \cos(\alpha) = \frac{\text{cateto adyacente}}{\text{hipotenusa}} = \frac{x}{h},$$

se puede calcular las coordenadas sobre la circunferencia al despejar x y y de las ecuaciones anteriores. Esta definición sólo se puede usar para $0 \leq \alpha \leq \pi/2$, pero como determinar x y y sigue siendo cierto para $0 \leq \alpha \leq 2\pi$. Esto se debe a las definiciones de las funciones trigonométricas seno y coseno, ya que sus valores coinciden con las coordenadas de un círculo de radio 1 centrado en el origen.

De esta manera, si $0 \leq \alpha \leq 2\pi$, se tiene que

$$x = r\cos(\alpha)$$

y

$$y = r\sin(\alpha).$$

Si se quiere variar el centro, lo único que hay que hacer es una traslación de las coordenadas del circulo, por lo que las coordenadas para un círculo con centro $c = (c_x, c_y)$ están dadas por:

$$x = r\cos(\alpha) + c_x$$

y

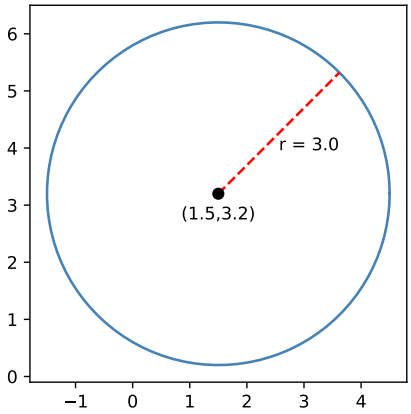
$$y = r\sin(\alpha) + c_y.$$

[165]:
```
#from matplotlib import pyplot as plt
#import numpy as np

def circunferencia(c = [0,0],r = 1,p = 100):
    """Calcula las coordenadas cartesianas de un circulo de radio r y centro c.
```

```
    c - Arreglo de tamaño 2. Coordenas del centro.
    r - Real. Radio.
    p - Entero. Numero de puntos para construir la curva.
    """
    t = np.linspace(0,2*np.pi,p)
    x = r*np.cos(t) + c[0]
    y = r*np.sin(t) + c[1]
    return x,y

def dibujaCirc(c,r):
    """Dibuja un circulo de radio r y centro c.
    c - Arreglo de tamaño 2. Coordenas del centro.
    r - Real. Radio.
    """
    x,y = circunferencia(c,r)
    rx,ry = r*np.cos(np.pi/4)+c[0],r*np.sin(np.pi/4)+c[1]
    plt.plot(x,y,'steelblue',[c[0],rx],[c[1],ry],'--r',c[0],c[1],'ko')
    plt.text((c[0]+rx)/2,(c[1]+ry)/2-r*0.1,f"r = {r:2.1f}")
    plt.text(c[0],c[1]-r*0.15,f"({c[0]:2.1f},{c[1]:2.1f})",horizontalalignment =␣
↪'center')
    ax = plt.gca()
    ax.set_aspect(1)
    plt.show()
```

[166]:
```
c = eval(input("Dame el centro (como una lista): "))
r = float(input("Dame el radio: "))
dibujaCirc(c,r)
```

```
Dame el centro (como una lista): 1.5,3.2
Dame el radio: 3
```



Manejo de excepciones

Incluso si una declaración o expresión es sintácticamente correcta, puede generar un error cuando se intenta ejecutar. Los errores detectados durante la ejecución se llaman excepciones, informando al usuario de que existe algún problema. Si la excepción no se captura el flujo de ejecución se interrumpe y se muestra la información asociada a la excepción en la consola de forma que el programador pueda solucionar el problema.

En *Python* se utiliza una construcción `try` - `except` para capturar y tratar las excepciones. El bloque `try` (*intentar*) define el fragmento de código en el que creemos que podría producirse una excepción. El bloque `except` (*excepción*) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción. Muchas veces nuestro tratamiento de la excepción consistirá simplemente en imprimir un mensaje más amigable para el usuario, otras veces nos interesará registrar los errores y de vez en cuando podremos establecer una estrategia de resolución del problema.

Por ejemplo, cuando tratamos de ingresar un valor por teclado y se da un valor diferente al esperado.

[167]:
```
try:
    w = int(input("Dame un numero entero: "))
    print(w)
except:
    print("Error en los argumentos de entrada")
```

```
Dame un numero entero: 4.5
Error en los argumentos de entrada
```

Python permite utilizar varios `except` para un solo bloque `try`, de forma que podamos dar un tratamiento distinto a la excepción dependiendo del tipo de excepción de la que se trate. Esto es una buena práctica, y solo es necesario indicar el nombre del tipo a continuación de `except`.

[168]:
```
try:
    w = float(input("Dame un numero: "))
    v = float(input("Dame otro numero: "))
    print(f"{w}/{v} = {w/v}")
except ValueError:
    print("Error en los argumentos de entrada, estos deben ser numéricos")
except ZeroDivisionError:
    print("El valor del divisor debe ser distinto de cero")
```

```
Dame un numero: 4
Dame otro numero: 0
El valor del divisor debe ser distinto de cero
```

Además podemos hacer que un mismo except sirva para tratar más de una excepción usando una tupla para listar los tipos de error que queremos que trate el bloque:

[169]:
```
try:
    num = eval(input("Dame una expresión: "))
    print(num)
```

```
except (NameError,TypeError,SyntaxError):
    print("Ocurrio un error de cierto tipo")
except:
    print("Sucedio un error imprevisto")
```

```
Dame una expresión: "Hola'
Ocurrio un error de cierto tipo
```

Una lista de las excepciones por defecto en *Python* se puede consultar en la siguiente liga.

Manejo de archivos

Ahora veremos la manipulación de archivos mediante Python. Es importante saber como importar y exportar datos desde un archivo ya que cuando se manejan volúmenes de información grandes es complicado pedir los datos al usuario a través de *input*.

Las funciones para manipular archivos en python son las siguientes:

Función / método	Sintaxis	Descripción
`open`	`f = open(archivo,modo)`	Da el acceso a un archivo con nombre *archivo* con permisos definidos por *mode*.
`close`	`f.close()`	Cierra el acceso al documento.
`write`	`f.write(cadena)`	Escribe el texto dado en *cadena* en el archivo.
`read`	`f.read(valor)`	Lee todo el contenido de un archivo a menos que se especifique el número de caracteres (*valor*) a leer
`readline`	`f.readline()`	Lee una línea del archivo
`readlines`	`f.readlines()`	Lee todas las líneas y las devuelve en una lista

Todos los argumentos de las funciones son cadenas excepto `valor` el cual debe ser `int`. El argumento `archivo` puede contener la ruta del archivo. Cuando queremos leer todo el archivo con `read` se usa la sintaxis `f.read()`. Los modos o permisos de acceso a un archivo definidos por `mode` son los siguientes:

Modo	Significado	Descripción
`'r'`	Lectura	*Valor por defecto.* Abre el archivo para lectura, marca error si el archivo no existe
`'a'`	Agregar	Abre un archivo para agregar, crea el archivo si no existe
`'w'`	Escritura	Abre un archivo para escritura, crea el archivo si no existe
`'x'`	Crear	Crea el archivo especificado, devuelve un error si el archivo existe

Ejemplos

Ejemplo 16 (Agenda)

Para ejemplificar su uso, se mostrara como implementar una agenda con tres datos agregados por el usuario, que son el nombre del contacto, el número de teléfono y correo electrónico del contacto que quiera agregar a la agenda. La base de datos se hará a través de un archivo *csv* (valores separado por comas), que es un formato de archivo simple para guardar información. Cuando se ejecuta el programa, se carga en memoria el archivo con la base de datos y cuando se sale de el se guardan los cambios hechos directo al archivo.

[170]:
```
def crearContacto(nombre,telefono,email,L):
    L.append(nombre+","+telefono+","+email)

def guardarAgenda(L):
    fileID = open("Agenda.csv",'w')
    for elem in L:
        fileID.write(elem+"\n")
    fileID.close()

def imprimirAgenda(L):
    for elem in L:
        M = elem.split(",")
        print(f"Nombre: {M[0]}\nTelefono: {M[1]}\nCorreo electrónico: {M[2]} \n")

def LeerAgenda():
    try:
        fileID = open("Agenda.csv",'r')
        L = list(map(lambda elem:elem.strip('\n'),fileID.readlines()))
        #LL = fileID.readlines()
        #L = []
        #for elem in LL:
        #    L.append(elem.rstrip('\n'))
        #LL.clear()
        fileID.close()
    except (FileNotFoundError):
        open("Agenda.csv",'x')
        L = []
    return L
```

[171]:
```
flag = True
agenda = LeerAgenda()
while flag:
    opt = input("""<<<Bienvenido a la agenda>>>\n
    1.- Crear nuevo contacto
    2.- Ver contactos
    3.- Salir\n
    Elige una opción:> """)
    if opt == "1":
```

```
        print("Escribe los siguientes datos")
        nom = input("Nombre: ")
        tel = input("Telefono: ")
        email = input("Correo electronico: ")
        crearContacto(nom,tel,email,agenda)
        print("El contacto se creo con exito")
    elif opt == "2":
        if len(agenda) == 0:
            print("La agenda no tiene contactos")
        else:
            imprimirAgenda(agenda)
    elif opt == "3":
        flag = False
        guardarAgenda(agenda)
    else:
        print("Opcion no disponible. Intenta con otra opción")
    if flag:
        input("Presione una tecla para continuar")
```

```
<<<Bienvenido a la agenda>>>

    1.- Crear nuevo contacto
    2.- Ver contactos
    3.- Salir

    Elige una opción:> 3
```