ANALISTA
DE DATOS

Autor de contenido
John Freddy Parra Peña

EMPEZAR

Tabla de Contenido

Presentación

En la sociedad actual se ha desarrollado una economía basada en datos e información, a tal punto que en este momento existen empresas líderes en mercados tradicionales, sin tener la infraestructura requerida para participar en dichos mercados tales como los conocíamos, por ejemplo hoy es fácil encontrar que la empresa líder en servicios de transporte individual de pasajeros, es una compañía que no cuenta con vehículos, así como en el mercado hotelero podemos ver que este es liderado por una empresa que no cuenta con camas ni cuartos de hotel. Estos nuevos participantes solo cuentan con datos e información y la infraestructura necesaria para conectar a quienes sí tienen disponible esa infraestructura con quienes requieren estos servicios.

En este curso se busca formar al estudiante en competencias relacionadas con el análisis de datos usando herramientas tecnológicas tales como: Hojas de cálculo, bases de datos, y el lenguaje de programación Python. Dentro de las temáticas a abordar se puede encontrar desde el procesamiento de datos usando hojas de cálculo hasta el uso del aprendizaje automático para la extracción de conocimiento a partir de conjuntos de datos.

Al finalizar este curso, los participantes deben estar en capacidad de apoyar procesos de análisis de datos que agreguen valor a los procesos de las empresas donde vayan a desempeñarse en este rol.

Objetivos del curso (competencias)

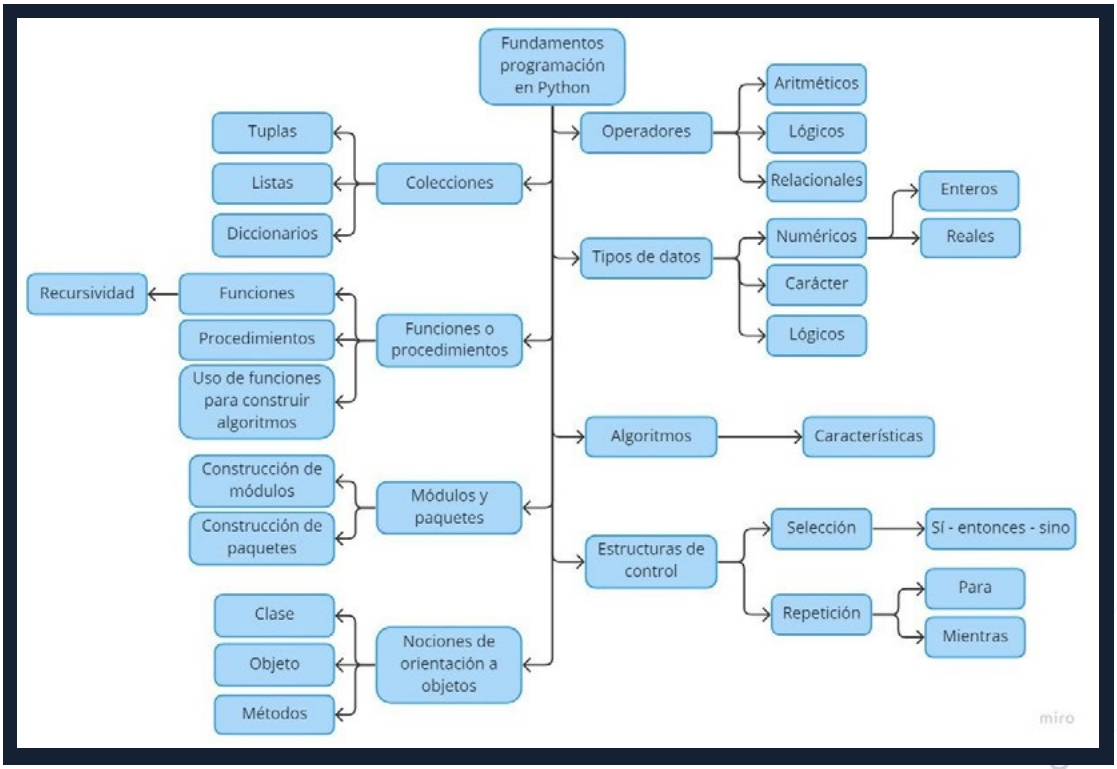
Objetivo general

Capacitar a los estudiantes en las diferentes partes del proceso de análisis de datos, iniciando desde la preparación y la limpieza de estos hasta la presentación de informes resultantes del procesamiento y representación de los datos.

Objetivo específico

- Presentar al estudiante las herramientas de uso común que pueden ser utilizadas para el procesamiento y análisis de los datos.
- Capacitar al estudiante en las técnicas para obtener y preparar los datos que van a ser analizados posteriormente
- Enseñar al estudiante a usar las herramientas para procesar y presentar los datos de forma visual, extrayéndoles valor
- Enseñar técnicas de aprendizaje automático para que estén en capacidad de predecir escenarios futuros a partir de información pasada y actual.

Mapa de contenido de la unidad



Módulo 4

Fundamentos de programación en Python

Con la cada vez más evidente dependencia que tienen las personas a las tecnologías de la información, tareas que antes eran exclusivas de ingenieros informáticos y desarrolladores se han ido tornando masivas, este es el caso de la programación de computadores; a tal punto que se pueden encontrar lenguajes de programación de propósito específico para automatizar procesos relacionados con diferentes áreas del conocimiento.

También la importancia de adquirir competencias en esta área se ve claramente en la inclusión de contenidos de programación en los currículos básicos de las escuelas en los países desarrollados, donde los niños van adquiriendo habilidades en programación a la par que aprenden matemáticas y otras ciencias básicas.

La evolución de las tecnologías de la información no solo ha tocado a los productos de software que se usan a diario sino también a los procesos de desarrollo y a los lenguajes de programación empleados. Anteriormente, dichos lenguajes constaban de una

sintaxis compleja, que resultaba de difícil acceso para quienes no tenían una formación básica en informática.
En la actualidad, existen numerosos lenguajes de programación con los que se ha buscado facilitar la forma como se escribe el código, logrando que estas herramientas estén al alcance del público en general; siendo Python uno de los lenguajes con mayor adopción.
El lenguaje de alto nivel Python fue creado en los años 90 por Guido Van Rossum, con la filosofía de ser un lenguaje fácil de aprender a escribir y de entender el código ya escrito. En este módulo se abordarán las generalidades de este poderoso lenguaje que se ha convertido en una de las herramientas más usadas para el análisis de datos.

4.1 Operadores y tipos de datos

Programar es enseñarle a la máquina a resolver una tarea de forma automatizada, para aprender a hacerlo es necesario entender las características y limitaciones de la máquina, como por ejemplo el hecho de que la memoria principal, por más amplia que sea no será suficiente para almacenar cualquier número. Lo anterior suena un poco extraño, sobre todo al ver que las máquinas hoy en día manejan cifras significativamente grandes.
Para aclarar esta idea se puede pensar en una operación aritmética sencilla, como por ejemplo dividir al número 1 entre 3, lo cual tendría como resultado el número 0,33333 con el 3 periódico, es decir que conceptualmente tendríamos una secuencia infinita del número 3 siguiendo a la coma, cuando se pasa del concepto a la representación en máquina, se encuentra que dicha secuencia infinita de 3 es imposible de almacenar en una memoria RAM que por más grande que sea resulta siendo siempre finita.

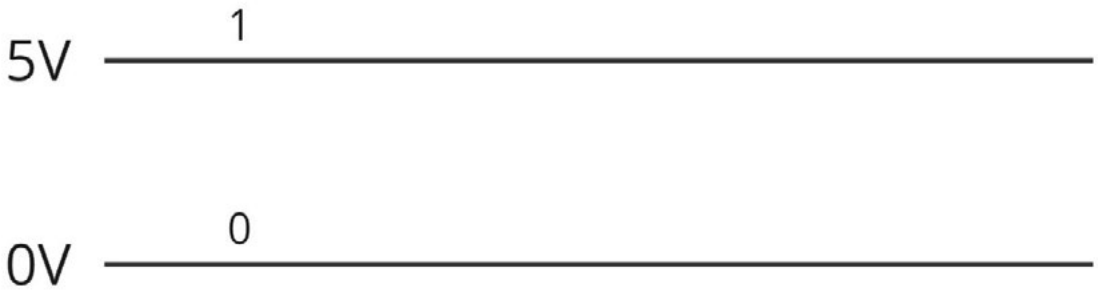
Si el caso fuera almacenar un número entero, se sabe que siempre a cualquier número se le podría sumar 1 de tal manera que nunca habría memoria suficiente para almacenar “el número más grande”; esta situación hace pertinente entender cómo se puede representar número en la máquina.

4.1.1 Números en base 2

Desde que existen las computadoras se ha escuchado que ellas trabajan usando números binarios, pero en la mayoría de los casos no se tiene clara la razón por la cual usan este sistema numérico y no el sistema en base 10 que es el que usan normalmente las personas.

Representar números en base 10, resulta muy fácil para el ser humano, dado que se cuenta con 10 dedos que permiten codificar los diferentes valores del sistema numéri-

co y se puede asimilar al 0 como la ausencia total de dedos, en el caso de las máquinas, el sistema decimal resulta muy difícil de adoptar dado que no se cuenta con 10 estados fácilmente diferenciables como es el caso de los dedos, sino que se cuenta con dos estados que no admiten confusión: encendido y apagado. Por ejemplo, si se observa un bombillo y se le pregunta a alguien si este se encuentra encendido o apagado, la respuesta sería fácil de dar, mientras que si la pregunta es qué tanto se encuentra encendido ya dependería de la interpretación de quien responde. Para el caso de los diferentes estados en la máquina, no se dependería de una subjetividad como en el caso de la pregunta al observador, sino que dependería de la influencia de dos fenómenos físicos que afectan la transmisión de electricidad, el primero de ellos es la resistencia que se opone al flujo de electricidad y el segundo la inductancia que como su nombre lo dice induce electricidad.



Teniendo en cuenta lo anterior, los estados de encendido y apagado no son exactos, se maneja un umbral que permite asociar valores cercanos a 5 voltios con un 1 y valores cercanos a 0v con un 0.

Números Naturales

El convertir números naturales de base 10 a base 2 es una tarea sencilla y que se basa en divisiones sucesivas entre 2 así:

- Se toma el número en base 10
- Se divide sucesivamente entre 2 calculando los cocientes y los residuos
- Se sigue dividiendo hasta obtener 1 como cociente
- Se lee el número resultante desde el cociente hasta el primer residuo
- A cada uno de los 1s o 0s que componen el número binario se les conoce como bit

Ejemplo:

Se va a convertir el número 43 de base 2 a base 10

Se realizan las divisiones sucesivas

43/2 = 21 con residuo 1
21/2 = 10 con residuo 1
10/2 = 5 con residuo 0
5/2 =2 con residuo 1
2/2 =1 con residuo 0

1

Se lee el número desde el cociente 1 hasta el primer residuo:101011

Otra forma abreviada de realizar la conversión es ubicar en una tabla las potencias de 2 e ir seleccionando cuáles son los mayores números en la tabla que sean menores o iguales al número a convertir, una vez se ubique este número se marcará con 1 y al número original se le restará el número seleccionado, el proceso se repite hasta llegar a 0., por ejemplo, así:

Para convertir el número 43 se ubica en la tabla el mayor número que sea menor o igual que 43, para este caso es 32

512	256	128	64	32	16	8	4	2	1
				1					

Se resta de 43 el 32 seleccionado obteniendo como resultado 11, se ubica el número 8 que sería el mayor número menor que 11, después de restar los 8 quedaría pendiente asignar un 2 y luego un 1. Finalmente, los números no seleccionados se completan con 0, obteniendo el mismo resultado anterior 101011.

512	256	128	64	32	16	8	4	2	1
				1	0	1	0	1	1

Para el proceso inverso, usando el método de la tabla, solo bastaría con transcribir el número en binario en la tabla, arrancando por el bit de menor valor y luego se procede a sumar los números que quedaron acompañados por un 1, de tal manera que se obtiene 1+2+8+32, lo cual es igual a 43.

Números Enteros

Para representar números enteros en máquina se parte de trabajar con una longitud de palabra definida, es decir que contrario al caso de los naturales, el número no se deja de la cantidad de bits que resultó de la conversión, sino que se completa a una longitud fija, por ejemplo 8 bits

El número entero de 8 bits que representaría al 43 sería 00101011, en esta representación lo que varía es cuando se trata de números negativos, para el caso de los negativos se usa una notación llamada complemento a 2, que consiste en reemplazar cada bit por lo que le falta para ser 2 (10). La forma sencilla de hacerlo es invertir uno a uno todos los bits del número en positivo y luego sumarle 1 al número ya invertido, así: Siguiendo con el número -43:

Paso 1: se representa el número 43 en enteró: 00101011

Paso 2: se invierten los bits: 11010100

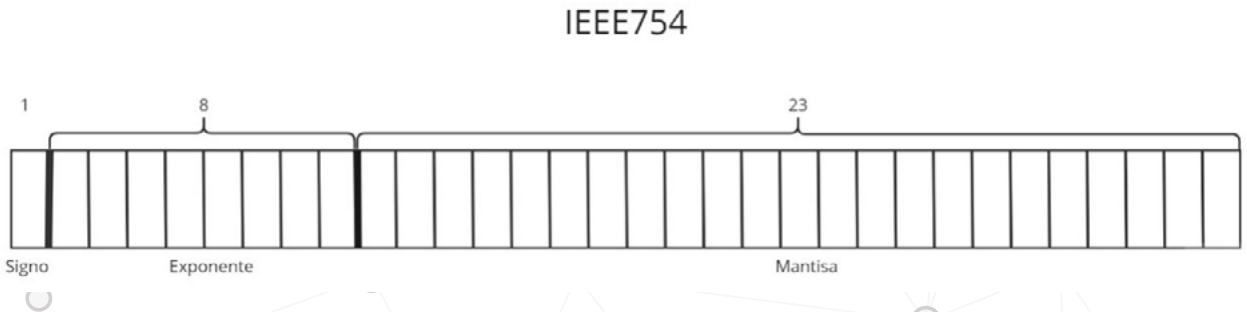
Paso 3: se suma 1 11010101

Nota: cuando se suma 1+1, el resultado es 0 y se lleva de acarreo 1.

Como resultado se tiene que el número -43 se representa así 11010101 en notación complemento a 2 de 8 bits

Números Reales

Para los números reales o de coma flotante se emplea una notación formulada por la IEEE, la 754. En la versión de 32 bits de esta notación, se toma una cadena de 32 bits y se divide así: 1 bit para signo 8 para exponente y 23 para la mantisa



El proceso para representar flotantes es el siguiente:

Paso 1, Usando el método de la tabla donde se tienen las potencias pre-calculadas, se ubica el número, teniendo especial cuidado con la ubicación de la coma

32	16	8	4	2	1	,	1/2	1/4	1/8	1/16	1/32

Luego se cuentan cuántas casillas se debe desplazar la coma para que quede solo un 1 a la izquierda, esa cantidad de veces que se mueve la coma será el exponente, al cual se le sumará 127 para garantizar que nunca vaya a ser negativo.

Finalmente, en la casilla de signo se guardará un 0 si es positivo o un 1 si es negativo, en las 8 siguientes se almacenará el exponente y en las 23 finales se colocarán todos los bits que quedaron a la derecha de la coma.

Ejemplo:

Representar el número 25,625

32	16	8	4	2	1	,	1/2	1/4	1/8	1/16	1/32
	1	1	0	0	1	,	1	0	1		

Para dejar solo un 1 a la izquierda, la coma se debe desplazar 4 veces, ese será el exponente, al cual se le sumará 127 es decir que se almacenará 131

	1	1	0	0	1	,	1	0	1		

1	,	1	0	0	1	1	0	1			

Lo que queda a la derecha de la coma será la mantisa
Como el número es positivo, el signo será 0, quedando así:
01000001110011010000000000000000
Signo: 0
Exponente: 131 -> 10000011
Mantisa: 10011010000000000000000

4.1.2 Tipos de dato os en base 2

En programación los tipos de datos representan al conjunto de posibles valores que puede tomar una variable, es decir, si esa variable puede almacenar datos numéricos o de otras características.

El lenguaje de programación Python es un lenguaje fuertemente tipado, pero de tipado dinámico, es decir que los tipos de dato, aunque no se deban definir explícitamente, se hacen respetar hasta que no se les asigne un nuevo valor, esto, es debido a que, en Python para definir una variable, solo es necesario inicializarla con un valor así:

```
>>> a=2
>>> type(a)
<class 'int'>
```

Al llamar a type, este dirá cuál es el tipo de dato de la variable por la cual se le pregunte.

4.1.2.1 Numéricos

En Python se soportan diferentes tipos de datos, entre los que se encuentran los datos numéricos que se pueden descomponer en enteros, reales y complejos, es importante tener en cuenta que Python dimensiona los tamaños de las variables dependiendo de la memoria disponible, ósea que no existen varios tipos de datos para almacenar enteros o flotantes de diferentes tamaños como ocurre en otros lenguajes

Enteros: soportan signo

```
>>> a=2
>>> type(a)
<class 'int'>
```

Flotantes: manejan coma decimal

```
>>> b=5.3
>>> type(b)
<class 'float'>
```

Complejos: son aquellos que se componen de una parte real y una imaginaria.

```
>>> c=2+3j
>>> type(c)
<class 'complex'>
```

4.1.2.2 Caracteres

Cuando se habla de cadenas de caracteres se hace referencia a textos, en Python estos textos se representan mediante el tipo de dato str y los valores que se asignan se deben encerrar en comillas, ya sean dobles o sencillas

```
>>> d="hola"
>>> type(d)
<class 'str'>
>>> e='mundo'
>>> type(e)
<class 'str'>
```

4.1.2.3 Booleanos

Los valores booleanos son aquellos que pueden tomar el valor de verdadero o falso

```
>>> f=True
>>> type(f)
<class 'bool'>
```

4.1.3 Operadores

Los operadores son los símbolos utilizados para realizar operaciones de diferentes naturalezas.

4.1.3.1 Operadores aritméticos

Las operaciones aritméticas básicas se pueden hacer utilizando este tipo de operadores, de los cuales la mayoría son ampliamente conocidos, como es el caso de la suma, la resta, la multiplicación y la división, otros como el residuo, la división entera o la potencia no resultan tan familiares

Suma +

```
>>> 3+4
7
```

Resta –

```
>>> 3-2
1
```

Multiplicación *

```
>>> 3*5
15
```

División /: esta es la división de números reales, la cual retorna números con parte decimal.

```
>>> 4/3
1.3333333333333333
```

División entera //: esta división devuelve solo números enteros, dado que en la división entera existe el residuo.

```
>>> 4//3
1
```

Residuo %: retorna el módulo o residuo de una división entera

```
>>> 4%3
1
```

Potencia **: calcula el valor de elevar una base a un exponente

```
>>> 4**3
64
```

Precedencia de los operadores, en Python como en los otros lenguajes existe una prioridad entre los operadores, es decir que unas operaciones se aplican primero que otras, por ejemplo, una multiplicación siempre se ejecutará antes que una suma, a no ser que existan paréntesis que modifiquen la prioridad.

```
>>> 2+3*4
14
>>> (2+3)*4
20
```

En el caso anterior al hacer la operación sin usar paréntesis, se realiza primero la multiplicación 3*4 y luego la suma 2+12.

De manera análoga, cuando se opera una potencia junto con una multiplicación, primero se realizará primero la potencia y luego la suma.

```
>>> 3*2**2
12
```

Primero se realizó la potencia 2 elevado a la 2 y luego el resultado se multiplica por 3

4.1.3.2 Operadores lógicos

Los operadores lógicos trabajan con valores booleanos, combinándolos y retornando valores booleanos también.

Y: el operador lógico and se encarga de evaluar si los dos booleanos recibidos son verdaderos, si es así, retorna verdadero, en caso contrario retorna falso

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

```
>>> True and False
False
>>> True and True
True
>>> False and True
False
>>> False and False
False
```

O: el operador lógico or se encarga de evaluar si alguno de los dos booleanos recibidos es verdadero, si es así, retorna verdadero, en caso contrario retorna falso

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

```
>>> False or False
False
>>> False or True
True
>>> True or False
True
>>> True or True
True
```

No: el operador lógico not, se encarga de negar o invertir el valor booleano recibido, es decir, si se recibe un valor verdadero, retornará falso, o si recibe un valor falso, retornará verdadero

a	not a
True	False
False	True

```
>>> not True
False
>>> not False
True
```

4.1.3.3 Operadores relacionales

Este tipo de operadores permite establecer la comparación entre dos valores y retorna un valor booleano si la condición evaluada es verdadera o falsa.

```
>>> 2>3
False
>>> 2>1
True
```

- Mayor: >, este operador retorna verdadero si el primer valor es mayor que el segundo, en caso contrario retorna falso.
-

```
>>> 2<3
True
>>> 3<2
False
```

- Menor: <, este operador retorna verdadero si el primer valor es menor que el segundo, en caso contrario retorna falso.

- Mayor o igual: >=, este operador retorna verdadero si el primer valor es mayor o igual que el segundo, en caso contrario retorna falso.

```
>>> 2>=3
False
>>> 2>=2
True
>>> 2>=1
True
```

- Menor o igual: <=, este operador retorna verdadero si el primer valor es menor o igual que el segundo, en caso contrario retorna falso.

```
>>> 3<=2
False
>>> 3<=3
True
>>> 3<=4
True
```

- Igual: ==, este operador retorna verdadero si el primer valor es igual al segundo, en caso contrario retorna falso.

```
>>> 3==4
False
>>> 3==3
True
```

- Diferente: !=, este operador retorna verdadero si el primer valor es diferente al segundo, en caso contrario retorna falso.

```
>>> 3!=3
False
>>> 3!=4
True
```

4.1.4 Construcción de algoritmos.

En programación, un algoritmo se entiende como una secuencia ordenada de pasos finitos y bien definidos, que usa recursos finitos y bien definidos para cumplir una tarea.

Dentro de las características de los algoritmos se encuentra que todo algoritmo por definición es determinístico, aún sin importar que su naturaleza sea estocástica. Esto último significa que siempre que se reciba una misma entrada se va a retornar una misma salida, sin importar si el algoritmo depende de números aleatorios, ya que en la máquina los aleatorios como tal no existen y son simplemente unos pseudo aleatorios que se generan a partir de una semilla, lo cual los pasa al contexto de lo determinístico.

Otra característica importante es que un algoritmo solo tiene un punto de entrada y un punto de salida.

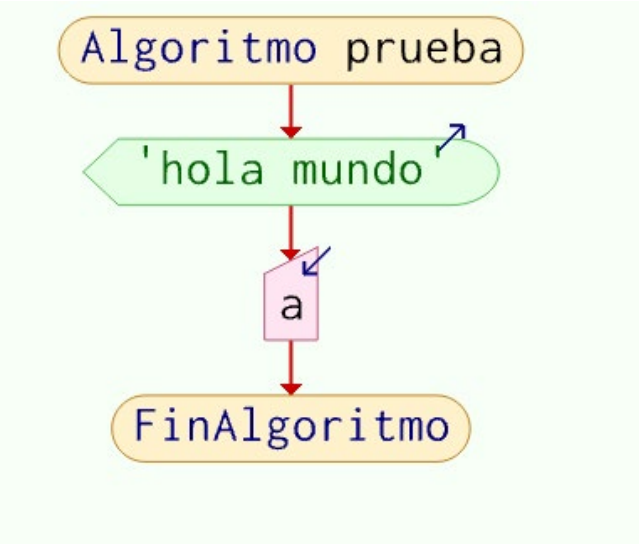
Estos algoritmos se pueden representar de diferentes formas, siendo los diagramas de flujo y el pseudocódigo una de las más usadas.

Adicional a las operaciones vistas anteriormente, a partir de este punto se van a incorporar dos nuevas operaciones que van a soportar la interacción con el usuario: La lectura y la escritura.

La lectura es la forma como se pueden recibir datos del usuario y la escritura la forma como se le enviaran mensajes a este último.

Ejemplo de operación de escritura y lectura

En diagrama de Flujo



En diagrama de Flujo

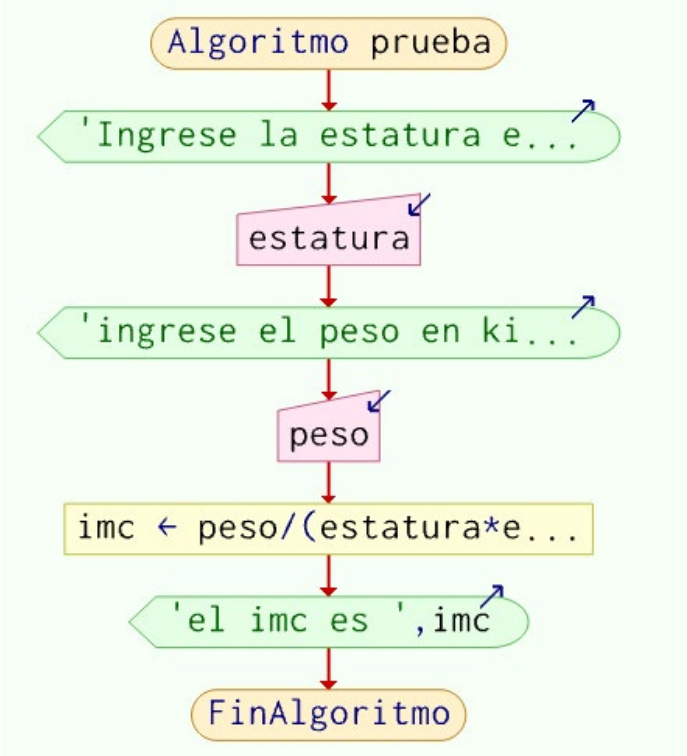
```
Algoritmo prueba
    escribir "hola mundo";
    leer a;
FinAlgoritmo
```

En Python

```
print("hola mundo")
a=input()
```

Una vez que se tienen estas herramientas para comunicarse con los usuarios, se empezará a formular algoritmos. Como se dijo previamente, los algoritmos tienen una tarea específica, la cual se resuelve mediante una secuencia ordenada de pasos, por ejemplo, si se quiere calcular el índice de masa corporal que tenga un usuario, lo primero que se debe es pedirle al usuario, tanto la estatura, como el peso, luego se procede a aplicar la fórmula que realiza el cálculo y finalmente se imprimirá en pantalla el resultado, así:

En diagrama de flujo:



En pseudocódigo:

```
Algoritmo prueba
    escribir "Ingrese la estatura en metros";
    leer estatura;
    escribir "ingrese el peso en kilogramos";
    leer peso;
    imc<-peso/(estatura*estatura);
    escribir "el imc es ",imc;
FinAlgoritmo
```

En Python:

```
estatura=float(input("Ingrese la estatura en metros"))
peso=float(input("ingrese el peso en kilogramos"))
imc=peso/estatura**2
print("El imc es",imc)
```

De manera análoga se pueden realizar múltiples algoritmos que tienen una ejecución netamente lineal, como por ejemplo calcular el volumen de una esfera dado el radio o convertir una temperatura de grados Celsius a Fahrenheit, estos dos algoritmos se presentan a continuación únicamente en lenguaje Python.

Algoritmo para calcular el volumen de una esfera

```
radio=float(input("Ingrese el radio de la esfera"))
volumen=4/3*3.141592654*radio**3
print("El volumen es",volumen)
```

Algoritmo para convertir temperatura

```
celsius=float(input("Ingrese temperatura en celsius"))
fahrenheit=32+celsius*9/5
print("la temperatura en Fahrenheit es",fahrenheit)
```

4.2 Estructuras de control

Los códigos escritos hasta este momento son códigos secuenciales, en los cuales las instrucciones se ejecutan en el estricto orden de la escritura, pero en la mayoría de los casos, se requiere que algunas instrucciones se ejecuten en lugar de otras o se repitan sin tener que ser escritas varias veces. Para este tipo de situaciones, se emplean las estructuras de control que se verán a continuación.

4.2.1 Estructuras de selección

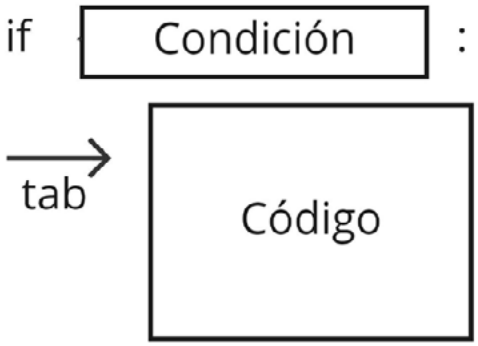
Este tipo de estructuras, permiten que dependiendo del cumplimiento o no de unas condiciones, se ejecuté un código u otro, cabe anotar que las condiciones que se formulan a través de evaluar expresiones cuyos únicos posibles resultados serán verdadero o falso, es decir, expresiones booleanas.

4.1.1.1 Si entonces-sino si entonces-sino

Este tipo de estructuras, permiten que dependiendo del cumplimiento o no de unas condiciones, se ejecuté un código u otro, cabe anotar que las condiciones que se formulan a través de evaluar expresiones cuyos únicos posibles resultados serán verdadero o falso, es decir, expresiones booleanas.

En el caso de Python se tiene solamente la estructura if elif else, aunque en otros lenguajes si existen varias de estas estructuras.

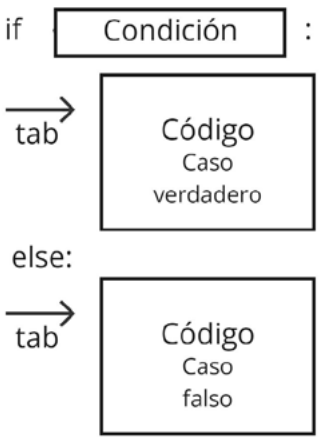
Esta estructura en Python tiene varias opciones, la primera de ellas consiste solo en condicionar la ejecución de un código dependiendo de si se cumple una expresión booleana o no, de esta manera:



Es importante tener en cuenta que en Python el alcance de una estructura de control está dada por la tabulación del código, es decir, todo lo que se encuentre dentro de la estructura, debe estar una tabulación dentro de la línea que abre la estructura, línea que finaliza con ":"

```
a=int(input("Ingrese a:"))
if a>0:
    print("positivo")
```

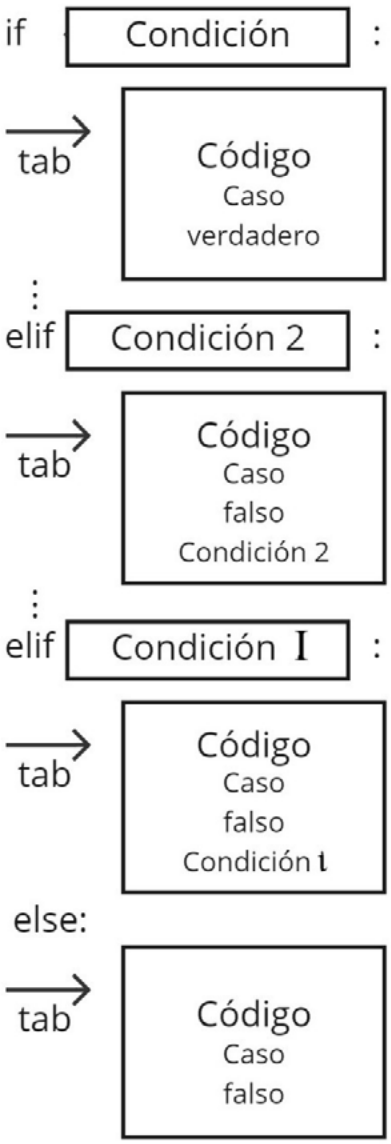
La segunda forma de usar este condicional es agregando una cláusula else (sino), la cuál será la opción para ejecutar si la condición al ser evaluada resulta falsa, así:



Para este caso se ejecutará alguna de las dos opciones.

```
a=int(input("Ingrese a:"))
if a>0:
    print("positivo")
else:
    print("No positivo")
```

La tercera opción es adicionar condiciones adicionales a evaluar en dado caso que no se cumplan las anteriores.



Si llegase el caso en que ninguna de las condiciones se cumple, se ejecutará el código escrito en la cláusula else.

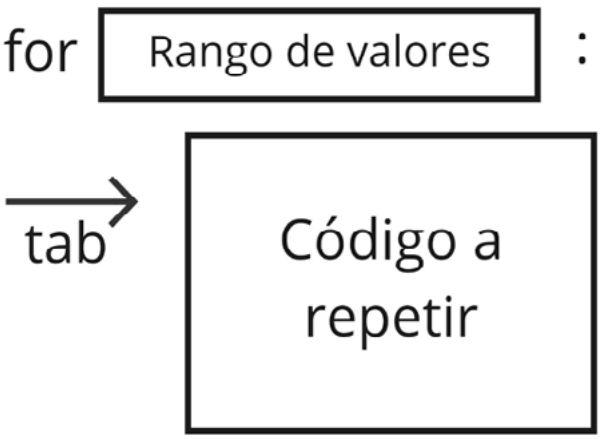
```
a=int(input("Ingrese a:"))
if a>0:
    print("positivo")
elif a==0:
    print("neutro")
else:
    print("Negativo")
```

4.2.2 Estructuras de repetición

Como su nombre lo indica, este tipo de estructuras, permiten que un bloque de código se ejecute varias veces

4.2.2.1 Estructura Para Todo

Esta estructura se usa cuando se conoce el número de veces que se deben repetir las instrucciones que se encuentran dentro de ella, en Python se usa la palabra reservada for, seguida de un iterador que se mueve en un rango, dicho rango se puede usar de tres maneras diferentes, dependiendo del número de parámetros que reciba.



Cuando el rango recibe un parámetro, este rango va desde 0 hasta el parámetro -1, por ejemplo, en el siguiente código, se imprimirán en pantalla los números de 0 a 9.

```
for i in range(10):
    print(i)
```

Si el rango recibe 2 parámetros, este irá desde el primer parámetro hasta el segundo menos 1, en el caso del siguiente ejemplo, se imprimirá desde el número 5 hasta el 9

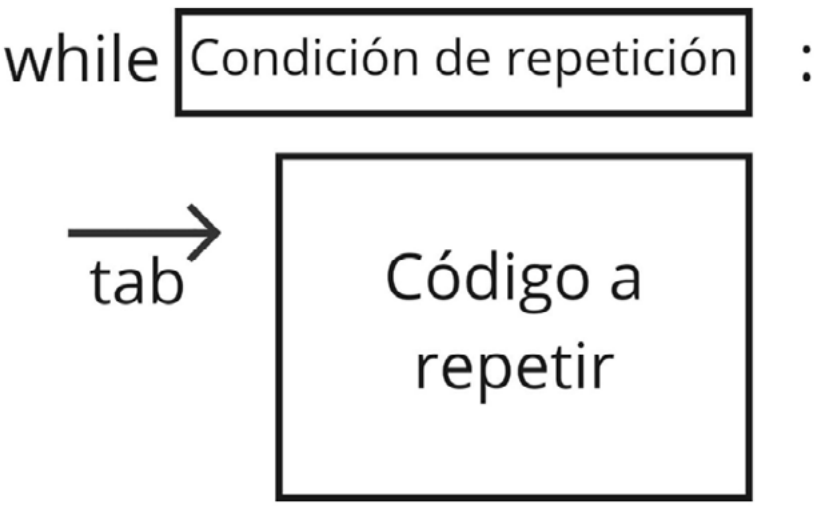
```
for i in range(5,10):
    print(i)
```

Por último, si el rango recibe 3 parámetros, este irá desde el primer parámetro, hasta el segundo -1, saltando lo que diga el tercer parámetro, para el siguiente ejemplo, se van a imprimir los números 5,10 y 15

```
for i in range(5,20,5):
    print(i)
```

4.2.2.2 Estructura Mientras

La estructura while realiza una repetición condicionada, es decir que, valida el cumplimiento de una condición, para poder seguir repitiendo. Las condiciones que aplican son de la misma naturaleza que las que aplican para la estructura if.



En el siguiente ejemplo, la condición de repetición es que a sea mayor que 1, y dentro del bloque de código se está dividiendo cada vez a la variable a entre 10 y se está imprimiendo este resultado, por lo tanto, se imprimirá 100, 10 y 1.

```
a=1000
while a>1:
    a=a//10
    print(a)
```

4.2.3 Construcción de algoritmos.

Ya en este punto, contando con la posibilidad de controlar el flujo de operaciones, resulta posible construir algoritmos más complejos y cercanos a las necesidades de automatización que se tienen en la actualidad.

Usando las estructuras de control se pueden construir algoritmos que ya no sean tan lineales, que evalúen diferentes escenarios y escojan el que sea más apropiado o que repitan varias veces una determinada tarea, como ejemplo, a continuación, se construirá un algoritmo que resuelva una ecuación cuadrática, sin importar si las raíces sean reales o imaginarias.

Para este algoritmo se usará una importación de una biblioteca que contiene funciones matemáticas para poder calcular la raíz cuadrada, esto se hará usando la línea import math. Luego se debe calcular el discriminante que es el valor al cual se le va a calcular la raíz cuadrada, en caso de que este sea positivo, se calculan raíces reales, en caso contrario se calculan raíces complejas.

```
import math
a=float(input("ingrese coeficiente a"))
b=float(input("ingrese coeficiente b"))
c=float(input("ingrese coeficiente c"))
discriminante=b**2-4*a*c
if discriminante>=0:
    raiz1 = (-b+math.sqrt(discriminante))/(2*a)
    raiz2 = (-b-math.sqrt(discriminante))/(2*a)
    print("raiz1:",raiz1,"raiz2:",raiz2)
else:
    print("raiz1:", -b/(2*a),"+",math.sqrt(-discriminante)/(2*a),"i")
    print("raiz2:",  -b/(2*a),"-",math.sqrt(-discriminante)/(2*a),"i")
```

Si para el ejemplo anterior se quiere seguir calculando hasta que el usuario lo desee, se puede incluir una estructura de repetición, así:

```
import math
repetir="S"
while repetir=="S" or repetir=="s":
    a=float(input("ingrese coeficiente a"))
    b=float(input("ingrese coeficiente b"))
    c=float(input("ingrese coeficiente c"))
    discriminante=b**2-4*a*c
    if discriminante>=0:
        raiz1 = (-b+math.sqrt(discriminante))/(2*a)
        raiz2 = (-b-math.sqrt(discriminante))/(2*a)
        print("raiz1:",raiz1,"raiz2:",raiz2)
    else:
        print("raiz1:", -b/(2*a),"+",math.sqrt(-discriminante)/(2*a),"i")
        print("raiz2:",  -b/(2*a),"-",math.sqrt(-discriminante)/(2*a),"i")
    repetir=input("Desea continuar S/N?")
```

4.3 Colecciones

Cuando se programa, a menudo es necesario definir variables que representan conjuntos de datos, esto resulta una tarea sencilla, usando las diferentes colecciones que Python tiene disponibles.

4.3.1 Tuplas

Estas son colecciones estáticas que pueden almacenar datos de diferentes naturalezas, pero por ser estáticas, una vez definidas no se puede cambiar su contenido., para definirlas se usan paréntesis que agrupan los elementos separados por comas, así:

```
tupla=(1,2,3,True,"hola")
```

Cuando se imprime una tupla, Python la imprime completa incluyendo los paréntesis

```
print(tupla)
```

```
(1, 2, 3, True, 'hola')
```

Para referirse a uno de los elementos de la tupla, se usa la posición de ésta, es importante tener en cuenta que dichas posiciones inician en 0 y finalizan en el tamaño de la tupla menos uno.

```
print(tupla[4])
```

```
hola
```

La estructura for vista anteriormente también se puede usar para navegar por la tupla, en el siguiente ejemplo, tomará el valor de cada uno de los elementos presentes.

```
for i in tupla:
    print(i)
```

```
1
2
3
True
hola
```

Otra forma de usar el for para moverse por la tupla, es hacer que i tome los valores del índice de cada una de las posiciones, definiendo un rango desde 0 hasta el tamaño de la tupla menos 1.

```
for i in range(len(tupla)):
    print(tupla[i])
```

```
1
2
3
True
hola
```

Por último, como se dijo al inicio, las tuplas son estáticas y sus componentes no pueden ser cambiadas, por ello si se trata de cambiar alguna, se generará un error.

```
tupla[i]=0
```

```
Traceback (most recent call last):
  File "C:\Users\User\PycharmProjects\pythonProject\main.py", line 13, in <module>
    tupla[i]=0
TypeError: 'tuple' object does not support item assignment
```

4.3.2 Listas

Estas colecciones funcionan de forma similar a las tuplas, pero tienen la ventaja de ser dinámicas, es decir que su contenido puede ser modificado, esto hace que las listas sean mucho más flexibles y usadas que las tuplas

A continuación, se mostrarán las mismas pruebas que se hicieron con las tuplas, pero usando listas.

La definición de la lista cambia con respecto a la definición de la tupla, la diferencia radica en que se usan paréntesis cuadrados en lugar de redondos.

```
lista=[1,2,3,True,"hola"]
```

A la hora de imprimir una lista, esta aparecerá con los paréntesis cuadrados

```
print(lista)
```

```
[1, 2, 3, True, 'hola']
```

Las siguientes operaciones funcionarán exactamente igual a las tuplas.

```
print(lista[4])

for i in lista:
    print(i)

for i in range(len(lista)):
    print(lista[i])
```

```
hola
1
2
3
True
hola
1
2
3
True
hola
```

Cuando se hace un cambio a una lista, esta se modifica sin generar errores.

```
lista[1]=0

print(lista)
```

```
[1, 0, 3, True, 'hola']
```

4.3.3 Slicing

El Slicing es una forma de moverse rápidamente por las colecciones, aplica tanto para listas como para tuplas y consiste en una forma especial de usar los subíndices así:

Para este ejemplo se partirá de una lista con los números del 1 al 10

```
lista=[1,2,3,4,5,6,7,8,9,10]
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

A esta lista le podemos acceder a sus componentes usando el subíndice como se había hecho previamente. En este caso si se imprime la posición 0, se mostrará el número 1

```
print(lista[0])
```

```
1
```

A esta lista le podemos acceder a sus componentes usando el subíndice como se había hecho previamente. En este caso si se imprime la posición 0, se mostrará el número 1, en Python también se puede acceder a un dato usando un subíndice negativo, el cual se ubicará iniciando desde el final de la lista, siendo esta la posición -1.

```
print(lista[-1])
```

```
10
```

Con el slicing se puede mostrar una parte de una colección indicando desde qué índice hasta cual índice -1 se quieren mostrar los datos.

```
print(lista[2:5])
```

```
[3, 4, 5]
```

Cuando se usan ":", no se requieren los dos límites para el intervalo, por ejemplo, si no se especifica el primer índice, el slicing iniciará en la posición 0

```
print(lista[:5])
```

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

De manera análoga, si no se especifica el segundo límite, los datos se traerán hasta la posición final

```
print(lista[2:])
```

```
[3, 4, 5, 6, 7, 8, 9, 10]
```

4.3.3 Diccionarios

A diferencia de las tuplas o las listas que usan índices numéricos para identificar los elementos, los diccionarios usan etiquetas, de tal manera que cada una de las componentes puede tener un "nombre" diferente, este tipo de estructura será la base de lo que posteriormente permitirá cargar tablas y datasets.

Cuando se crea un diccionario se usan llaves {} para encerrar a las diferentes componentes separadas por comas, cada componente ira de la forma "etiqueta":

```
diccionario={"primero":1,"segundo":2,"tercero":3}
print(diccionario)
```

```
{'primero': 1, 'segundo': 2, 'tercero': 3}
```

Para referirse a una componente se usará su etiqueta:

```
print(diccionario["primero"])
```

```
1
```

Los elementos de un diccionario se pueden cambiar, dado que estos diccionarios no son estáticos.

```
diccionario["segundo"]=4
print(diccionario)
```

```
{'primero': 1, 'segundo': 4, 'tercero': 3}
```

4.4 Funciones y procedimientos

Las funciones y los procedimientos son fragmentos de código que se pueden escribir una vez para ser usados múltiples veces, esto ayuda a que el código resulte más corto, sencillo y comprensible.

4.4.1 Funciones

Las funciones se definen como bloques de código que tienen un nombre, que pueden o no recibir argumentos en su llamado y que retornan algo como resultado de su ejecución, por ejemplo, la raíz cuadrada que se utilizó anteriormente para calcular las raíces de una ecuación cuadrática. A cada función se le asigna una única responsabilidad y un nombre que sea bastante intuitivo, de tal manera que no genere confusión a quienes la van a usar.

Para definir una función se debe usar la palabra reservada def seguida del nombre a asignar, luego entre paréntesis deben venir los argumentos que se requieran cuando esta vaya a ser utilizada, finalmente esta línea de definición terminará en ":"

Después dentro de la función se escribe el código a ejecutar y al final se escribe una sentencia return acompañada del resultado.

A continuación, se presenta como ejemplo una función que calcule el área de un círculo, la cual debe recibir como parámetro el radio.

```
import math

def areaCirculo(radio):
    return math.pi*radio**2
```

Una vez creada la función puede llamarse múltiples veces

```
print(areaCirculo(2))
print(areaCirculo(3))
```

Por cada vez que sea invocada retornaré un resultado, acorde con los argumentos recibidos

```
12.566370614359172
28.274333882308138
```

4.4.1.1 Recursión

La recursión en programación hace referencia a una función que se llame a sí misma, logrando hacer repeticiones sin necesidad de usar estructuras de control, por ejemplo, para calcular un término en la sucesión de Fibonacci (0 1 1 2 3 5 8 13 21 34 55...) se podría calcular sumando otras dos invocaciones de la misma sucesión. A continuación se presentan dos opciones para construir una función que calcule el n-ésimo término de esta sucesión, la primera no será recursiva y la segunda si.

Fibonacci no recursivo, en esta función se usa un ciclo for para realizar el cálculo de los valores de las posiciones mayores a 2.

```
def fibonacci(n):
    f1=0
    f2=1
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        for i in range(3,n+1):
            temp=f2
            f2=f1+f2
            f1=temp
        return f2

print(fibonacci(10))
```

Al llamar a la función con parámetro 10, retorna 34

```
34
```

Fibonacci Recursivo, en esta función se calcula el valor de la posición n de fibonacci sumando el valor de las posiciones n-1 y n-2, siendo estos valores el resultado de invocar nuevamente a la función de fibonacciRecursivo, nótese que en este caso existen dos criterios de salida, cuando n=1 o cuando n=2, estos criterios de salida son la forma como se detiene la recursión y esta no continúa indefinidamente hasta generará un error.

```
def fibonacciRecursivo(n):
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        return fibonacciRecursivo(n-1)+fibonacciRecursivo(n-2)

print(fibonacciRecursivo(10))
```

Al llamar a la función con parámetro 10, retorna 34

```
34
```

La sintaxis de la función recursiva es más sencilla que la de la primera opción, pero computacionalmente la recursión resulta más pesada para la máquina.

4.4.1.2 Procedimientos

Estos se parecen a las funciones, también se definen como bloques de código que tienen un nombre, que pueden o no recibir argumentos en su llamado, pero estos no retornan nada como resultado de su ejecución. Se usan para ejecutar tareas de las cuales no se espera que retornen una respuesta para ser usada posteriormente.

Por ejemplo, se usan procedimientos para labores de impresión o grabado en archivos, a continuación, se presentará un procedimiento que imprima en pantalla la tabla de multiplicar del número recibido como parámetro.

```
def tablaMultiplicar(n):
    for i in range(11):
        print(n,"*",i,"=",n*i)
tablaMultiplicar(5)
```

Nótese que la sintaxis del procedimiento definido sólo varía de la sintaxis de una función en que el procedimiento no tiene la cláusula return.

```
5 * 0 = 0
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

4.4.1.3 Uso de funciones para la construcción de algoritmos

En programación se aplica la expresión coloquial “Divide y Vencerás”, que en este caso se utiliza partiendo un problema grande en varios problemas más pequeños, es decir, un problema se divide en funciones y/o procedimientos reduciendo la complejidad de las tareas, al final dichas funciones y/o procedimientos se integran para solucionar el problema completo.

Como ejemplo se plantea la construcción de un algoritmo que calcule algo muy usado en probabilidades, las permutaciones y combinaciones de una variable n en una variable r. La permutación es la cantidad de subconjuntos de r elementos seleccionados de un conjunto de n elementos importando el orden y la combinación es la cantidad de subconjuntos pero sin importar el orden.

Como se puede ver en las siguientes fórmulas, estos cálculos se hacen usando factoriales

$$nPr = \frac{n!}{(n-r)!}$$

$$nCr = \frac{n!}{r!\,(n-r)!}$$

Para solucionar el problema se creará una función factorial, una función combinación, una función permutación y se integrarán para crear el algoritmo completo.

La función factorial se crea de forma recursiva, dejando n=0 como criterio de salida

```
def factorial(n):
    if n==0:
        return 1
    else:
        return factorial(n-1)*n
```

Las funciones permutación y combinación llaman en total 5 veces a la función factorial

```
def permutacion(n,r):
    return factorial(n)/factorial(n-r)
def combinacion(n,r):
    return factorial(n) / (factorial(n - r)*factorial(r))
```

Finalmente se leen los valores y se invocan a las funciones creadas

```
n=int(input("ingrese n:"))
r=int(input("ingrese r:"))
print(combinacion(n,r))
print(permutacion(n,r))
```

Como resultado de la ejecución se puede ver que al recibir los valores de 5 y 2, el resultado de la combinación fue 10 y el de la permutación fue 20.

```
ingrese n:5
ingrese r:2
10.0
20.0
```

4.5 Módulos y paquetes.

Continuando con la división de un problema en problemas más pequeños, es frecuente crear una estructura de archivos y carpetas que contengan los códigos, esto con el fin de que el código esté ordenado.

4.5.1 Construcción de módulos

La construcción de módulos es una tarea sencilla, consiste básicamente en crear archivos .py que contengan código python para ser invocado posteriormente, generalmente tienen funciones y procedimientos. Estos módulos podrán ser importados en los programas.

4.5.2 Construcción de paquetes

Para construir un paquete, simplemente se crea una carpeta que contenga módulos dentro de ella, estos paquetes pueden ser importados completos a los diferentes programas.

4.6 Nociones de Programación Orientada a Objetos

El paradigma de la orientación a objetos consiste en dividir los problemas a solucionar en los diferentes objetos que interactúan en esa solución, aunque el paradigma como tal es extenso de explicar, para este curso se abordarán unos conceptos que se usarán más adelante cuando se estén usando las bibliotecas propias del análisis de datos.

4.6.1 Objeto

Por definición un objeto es cualquier cosa que se pueda definir basándose en sus características de forma y de comportamiento. Cada objeto suele representar una abstracción de un elemento del mundo real.

Un ejemplo de objeto sería el computador que se está usando para leer este documento.

4.6.2 Clase

Una clase es la definición genérica de un conjunto de objetos, es decir que la clase es como la plantilla que se utiliza para construir los diferentes objetos. Es importante diferenciar el concepto de clase del concepto de objeto, ya que aunque el paradigma se llame orientado a objetos, a la hora de programar se codifican clases, dado que al modelar una clase, se están estableciendo las características de un conjunto de objetos que comparten su misma naturaleza. En términos generales, la clase es el conjunto y el objeto es sólo un individuo dentro de ese conjunto.

Siguiendo con el ejemplo usado para el objeto, como ejemplo de clase, se puede pensar en la definición genérica de un computador, cuando se habla de esta definición genérica, se está haciendo referencia a todos los computadores y no a ninguno en particular.

4.6.3 Atributos

Estos representan cada una de las características de forma que tiene un objeto.

Continuando con el ejemplo que se viene trabajando, un atributo de computador puede ser su marca o la cantidad de memoria que tiene.

4.6.4 Métodos

Los métodos representan a las características de comportamiento que tienen los objetos, es decir las acciones inherentes que estos tienen, para el caso del computador, una de estas acciones puede ser encenderse o apagarse.

4.6.5 Ejemplo

Para este ejemplo sencillo, se modelará la clase punto, la cual representará todos los puntos en dos dimensiones, los atributos serán x, y, se creará un único método distanciaOrigen, el cual retornará la distancia de un punto al punto (0,0)

La palabra reservada para crear una clase es la palabra class la cual vendrá seguida del nombre que se le dará a la clase, dentro de esta se define el método __init__ que es el que se encarga de construir los objetos nuevos, como parámetros recibe siempre self, acompañado de los diferentes atributos definidos.

Los otros métodos que se definan, también iniciarán con el argumento self y se encargará de desarrollar cada uno de los comportamientos inherentes a los objetos.

```
import math
class Punto:
    def __init__(self,x,y):
        self.x=x
        self.y=y
    def distanciaOrigen(self):
        return math.sqrt(self.x**2+self.y**2)
```

Una vez se tiene la clase, se procede a construir un objeto llamando al constructor de la misma, el cual se invoca usando el mismo nombre de la clase y se le pasan los argumentos adicionales a self que fueron definidos en el método __init__.

Finalmente, para invocar un método, este se invoca desde el objeto previamente construido.

```
p=Punto(1,1)
print(p.distanciaOrigen())
```

```
1.4142135623730951
```

Otros materiales para profundizar

Material complementario

Documentación oficial de Python https://docs.python.org/3/

Libro Python para todos: https://docs.google.com/viewer?a=v&pid=sites&srcid=-bWF0aG1vZGVsbGluZy5vcmd8d3d3fGd4OmMyNGI3ZmU0YzllNzk3Mg

Referencias bibliográficas de la unidad

Fred L. Drake, J. (n.d.). Python 3.11.1 Documentation. Retrieved December 19, 2022, from https://docs.python.org/3/

Dijkstra, Edsger. (1968). Go To Statement Considered Harmful. Communications of The ACM - CACM.


ALCALDÍA MAYOR
DE BOGOTÁ D.C.
SECRETARÍA DE EDUCACIÓN


ATENEA
AGENCIA DISTRITAL PARA LA EDUCACIÓN
SUPERIOR LA CIENCIA Y LA TECNOLOGÍA


BOGOTÁ


UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS
Acreditación Institucional de Alta Calidad