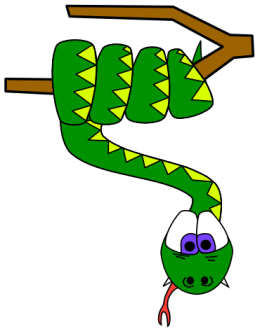



# Programación en Python

**Sesión 2**

**Sentencias de Control**

**Víctor Melchor**

# Objetivos

Después de completar este capítulo, podrás:

- Usar declaraciones de selección para poder elegir entre opciones en un programa.
- Construir condiciones apropiadas para bucles controlados por condición y declaraciones de selección.
- Usar operadores lógicos para construir expresiones booleanas compuestas.
- Utilizar una instrucción de selección y una instrucción de interrupción para salir de un bucle que no está controlado por entrada

# Objetivos

Después de completar este capítulo, podrás:

- Escribir un lazo para repetir una secuencia de acciones un número fijo de veces.
- Escribir un bucle para recorrer la secuencia de caracteres en una cadena.
- Escribir un bucle que cuenta hacia atrás y un bucle que cuenta hacia adelante.
- Escribir un bucle controlado por entrada que se detenga cuando una condición se vuelva falsa.

# Selección: declaraciones if y if-else

- **Las declaraciones de selección** permiten que una computadora tome decisiones.
  - Basado en una condición.

# El tipo booleano, comparaciones y expresiones booleanas

- El tipo de datos **booleano** consta de dos valores: verdadero y falso (generalmente a través de True / False)

| COMPARISON OPERATOR | MEANING |
|---|---|
| == | Equals |
| != | Not equals |
| < | Less than |
| > | Greater than |
| <= | Less than or equal |
| >= | Greater than or equal |

[TABLE 3.2] The comparison operators

- Ejemplo: `4 != 4` se evalúa a **`False`**

# Declaraciones **if-else**

- También se llama una declaración de selección de dos vías
- A menudo se utiliza para comprobar entradas en busca de errores:

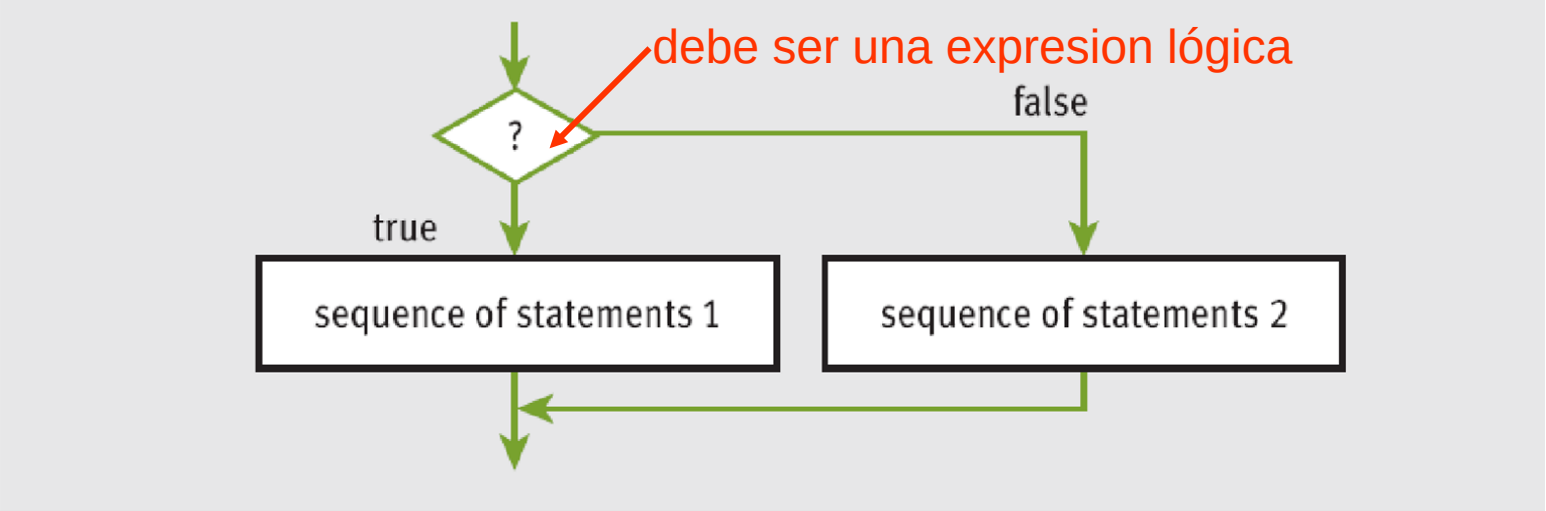
```
import math
area=float(input("Ingrese el area: "))

if area>0:
    radio=math.sqrt(area/math.pi)
    print("El radio es: ", radio)
else:
    print("Error, el area debe ser un valor positivo.")
```

- Sintaxis:

```
if <condition>:
    <sequence of statements-1>
else:
    <sequence of statements-2>
```

# Declaraciones **if-else**




[FIGURE 3.2] The semantics of the **if-else** statement

```
prim=float(input("Ingrese el primer numero: "))
seg=float(input("Ingrese el segundo numero: "))

if prim>seg:
    maximo=prim
    minimo=seg
else:
    maximo=seg
    minimo=prim
print("Maximo: ",maximo)
print("Minimo: ",minimo)
```

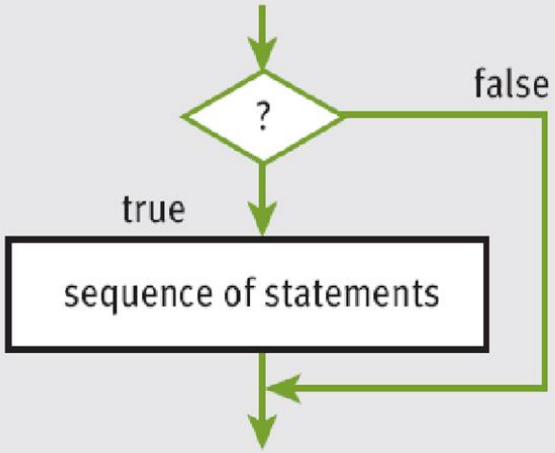
Mejor alternativa:

```
prim=float(input("Ingrese el primer numero: "))
seg=float(input("Ingrese el segundo numero: "))
print("Maximo: ",max(prim,seg))
print("Minimo: ",min(prim,seg))
```

# Declaraciones de selección unidireccional

- La forma más simple de selección es la instrucción **if**

```
if <condition>:
    <sequence of statements>
```




[FIGURE 3.3] The semantics of the `if` statement

```
>>> if x<0:
        x=-x
```

# Declaraciones Múltiples `if`

- Un programa puede enfrentar condiciones de prueba que implican más de dos cursos de acción alternativos

| LETTER GRADE | RANGE OF NUMERIC GRADES |
|---|---|
| A | All grades above 89 |
| B | All grades above 79 and below 90 |
| C | All grades above 69 and below 80 |
| F | All grades below 70 |

[TABLE 3.3] A simple grading scheme

- Puede ser descrito en el código por una declaración de selección de **múltiples if**

# Declaraciones Múltiples `if`

```
num=int(input("Ingrese la nota numerica: "))
if num>89:
    letra='A'
elif num>79:
    letra='B'
elif num>69:
    letra='C'
else:
    letra='F'
print("La nota en literal es: ",letra)
```

- Sintaxis:

```
if <condition-1>:
    <sequence of statements-1>

elif <condition-n>:
    <sequence of statements-n>
else:
    <default sequence of statements>
```

# Operadores lógicos y expresiones booleanas compuestas

- A menudo se debe tomar un curso de acción si se cumple alguna de las dos condiciones: A continuación se presentan dos enfoques

```
num=int(input("Ingrese la nota numerica: "))
if num>100:
    print("Error, la nota debe estar entre 0 y 100")
elif num<0:
    print("Error, la nota debe ser positiva")
else:
    #El codigo para calcular e imprimir los resultados
```

```
num=int(input("Ingrese la nota numerica: "))
if num>100 or num<0:
    print("Error, la nota debe estar entre 0 y 100")
else:
    #El codigo para calcular e imprimir los resultados
```

  - ¿Podríamos usar el operador lógico **and** en su lugar?

# Operadores lógicos y expresiones booleanas compuestas

| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

| A | not A |
|---|---|
| True | False |
| False | True |

[FIGURE 3.4] The truth tables for **and**, **or**, and **not**

# Operadores lógicos y expresiones booleanas compuestas

- El siguiente ejemplo verifica algunas de las afirmaciones hechas en las tablas de verdad anteriores:

```
>>> A = True
>>> B = False
>>> A and B
False
>>> A or B
True
>>> not A
False
```

- Los operadores lógicos se evalúan después de las comparaciones pero antes del operador de asignación
  - **`not`** tiene mayor precedencia que **`and`** y **`or`**

# Operadores lógicos y expresiones booleanas compuestas

| TYPE OF OPERATOR | OPERATOR SYMBOL |
|---|---|
| Exponentiation | `**` |
| Arithmetic negation | `-` |
| Multiplication, division, remainder | `*`, `/`, `%` |
| Addition, subtraction | `+`, `-` |
| Comparison | `==`, `!=`, `<`, `>`, `<=`, `>=` |
| Logical negation | `not` |
| Logical conjunction and disjunction | `and`, `or` |
| Assignment | `=` |

[TABLE 3-4] Operator precedence, from highest to lowest

# Evaluación de cortocircuito

- En (A y B), si A es falso, entonces también lo es la expresión, y no hay necesidad de evaluar B
- En (A o B), si A es verdadero, entonces también lo es la expresión, y no hay necesidad de evaluar B
- **Evaluación de cortocircuito:** la evaluación se detiene lo antes posible

```
cont=int(input("Ingrese el contador: "))
suma=int(input("Ingrese la suma: "))
if cont>0 and suma/cont>10:
    print("Promedio > 10 ")
else:
    print("cont==0 o promedio <=10")
```

La evaluación de cortocircuito puede ser usado para evitar la división por cero

# Pruebas de selección de declaraciones

- Sugerencias:
  - Asegúrese de que todas las posibles alternativas en una declaración de selección se realicen.
  - Después de probar todas las acciones, examine todas las condiciones.
  - Condiciones de prueba que contienen expresiones booleanas compuestas que usan datos que producen todas las combinaciones posibles de valores de los operandos.

# Iteración definida: el bucle for

- Declaraciones de repetición (o bucles) repiten una acción
- Cada repetición de acción se conoce como **pasada** o **iteración**.
- Dos tipos de bucles
  - Aquellos que repiten la acción un número predefinido de veces **(iteración definida)**
  - Aquellos que realizan acciones hasta que el programa determine que debe detenerse **(iteración indefinida)**

# Ejecutar una declaración un número dado de veces

- El bucle for Python es una declaración de control que admite la iteración definida más fácilmente

```
>>> for i in range(4):
        print("Felices Fiestas ",end="")


Felices Fiestas Felices Fiestas Felices Fiestas Felices Fiestas
>>>
```

- La forma de este tipo de bucle es:

```
for  <variable>  in  range( <expresion entera>):
    <sentencia 1>



    <sentencia N>
```

← encabezado de bucle

← las declaraciones en el cuerpo deben estar indentadas y alineadas en la misma columna

# Ejecutar una declaración un número dado de veces

- **Ejemplo:** Bucle para calcular una exponenciación para un exponente no negativo

```
number=2
exponent=3
product=1
for i in range(exponent):
    product = product*number
    print(product,end = " ")
```

  - Si el exponente fuera 0, el cuerpo del bucle no se ejecutaría y el valor del producto se mantendría como 1

# Bucles controlados por contador

- Bucles que cuentan a través de un rango de números.

```
product=1
for cont in range(4):
    product = product*(cont+1)

print(product)
```

- Para especificar un límite inferior explícito:

```
product=1
for cont in range(1,5):
    product = product*(cont+1)

print(product)
```

# Bucles controlados por contador

- Aquí está la forma de esta versión del bucle for:

```
for  <variable>  in  range( <vmin> , <vmax> + 1):
      <sentencia 1>



      <sentencia N>
```

- Ejemplo: suma delimitada por límites

```
vmin=int(input("Ingrese el valor minimo: "))
vmax=int(input("Ingrese el valor maximo: "))
suma=0
for cont in range(vmin,vmax+1):
    suma = suma + cont

print(suma)
```

# Asignación aumentada

- **Operaciones de asignación aumentada**:

```
a = 17
s = "hi"

a += 3            # Equivalent to a = a + 3
a -= 3            # Equivalent to a = a - 3
a *= 3            # Equivalent to a = a * 3
a /= 3            # Equivalent to a = a / 3
a %= 3            # Equivalent to a = a % 3
s += " there"     # Equivalent to s = s + " there"
```

- Formato:

```
<variable> <operator>= <expression>
```

Equivalente a:

```
<variable> = <variable> <operator> <expression>
```

# Errores de bucle

- Ejemplo:

```
for cont in range(1,4):
        print(count,end="")
```

  El bucle en realidad cuenta de 1 a 3
- Esto no es un error de sintaxis, sino un error lógico.

# Recorriendo los contenidos de una secuencia de datos

- **`range` devuelve una lista**

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1, 5)
[1, 2, 3, 4]
```

- Strings también son secuencias de caracteres.
- Los valores en una secuencia se pueden visitar con un bucle for:

```
for <variable> in <sequence>:
    <do something with variable>
```

- Ejemplo:

```
for caracter in "Hola a todos!":
        print(caracter, end=" ")
```

# Especificando los pasos en Range

- **`range`** espera un tercer argumento que le permite especificar una razón

```
>>> range(1, 6, 1)    # Same as using two arguments
[1, 2, 3, 4, 5]
>>> range(1, 6, 2)    # Use every other number
[1, 3, 5]
>>> range(1, 6, 3)    # Use every third number
[1, 4]
```

- Ejemplo en un bucle:

```
suma=0
for cont in range(2,11,2):
        suma += cont
print(suma)
```

# Bucles que cuentan hacia atrás

- Ejemplo:

```
for cont in range(10, 0 , -1):
        print(cont,end=" ")
```

# Formato de texto para salida

- Muchas aplicaciones de procesamiento de datos requieren una salida que tenga **formato tabular**
- **Ancho del campo**: Número total de caracteres de datos y espacios adicionales para un dato en una cadena con formato

```
>>> for expo in range(7 , 11):
        print(expo,10**expo)


7 10000000
8 100000000
9 1000000000
10 10000000000
>>> 
```

```
>>> "%6s" % "four"         # Right justify
'  four'
>>> "%-6s" % "four"        # Left justify
'four  '
```

# Formato de texto para salida

```
<format string> % <datum>
```

- Esta versión contiene **formato de cadena**, **operador de formato** % y valor único para formatear
- Para formatear enteros, se usa la letra **d** en lugar de **s**

- Para formatear una secuencia de valores de datos:

```
<format string> % (<datum-1>, …, <datum-n>)
```

```
>>> for expo in range(7 , 11):
        print("%-3d%12d"%(expo,10**expo))


7        10000000
8       100000000
9      1000000000
10    10000000000
```

# Formato de texto para salida

- Para formatear datos del tipo float:

```
%<field width>.<precision>f
```

  donde ***.<precision>*** es opcional

- Ejemplos:

```
>>> salario=100.00
>>> print("Tu salario es $"+str(salario))
Tu salario es $100.0
>>> print("Tu salario es $%0.2f"%salario)
Tu salario es $100.00
>>> "%6.3f"%3.14
' 3.140'
>>>
```

# Caso de Estudio: un informe de inversión

- Requerimiento:
  - Escribe un programa que calcule un informe de inversión.

# Caso de Estudio: un informe de inversión

- Análisis:

```
Ingrese el monto de inversion: 10000
Ingrese el numero de años: 5
Ingrese la tasa: 5
Año      Balance Inicial      Interes      Balance Final
   1          10000.00        500.00           10500.00
   2          10500.00        525.00           11025.00
   3          11025.00        551.25           11576.25
   4          11576.25        578.81           12155.06
   5          12155.06        607.75           12762.82
Balance Final: $12762.82
Total de interes ganado: $ 2762.82
>>>
```

# Caso de Estudio: un informe de inversión

- **Diseño:**
  - Reciba las entradas del usuario e inicialice los datos.
  - Muestre el encabezado de la tabla
  - Calcule los resultados para cada año y visualícelos.
  - Muestre los totales

# Caso de Estudio: un informe de inversión

- Codificación:

```
#Calcular y mostrar los resultados cada año
for year in range(1 , 5+1):
    interes=balIni*tasa
    balFinal=balIni + interes
    print("%4d%18.2f%10.2f%16.2f"%\
          (year,balIni,interes,balFinal))
    balIni=balFinal
    totalInteres+= interes
```

# Iteración condicional: el bucle while

- El bucle while se puede usar para describir la iteración condicional
  - **Ejemplo:** El bucle de entrada de un programa que acepta valores hasta que el usuario ingresa un **centinela** que termina la entrada.

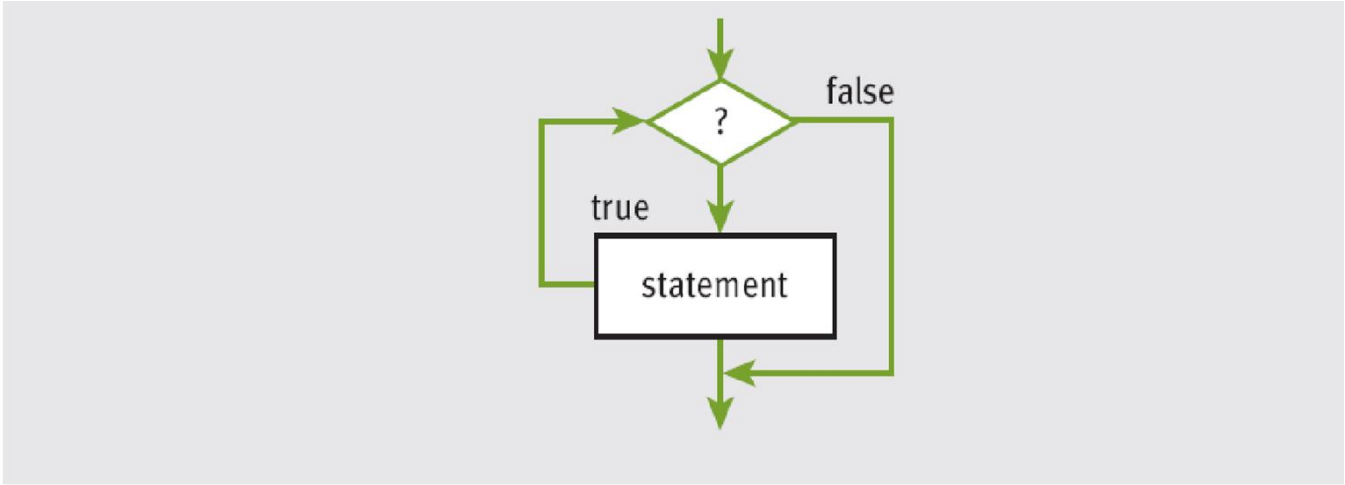
# La estructura y el comportamiento de un bucle `while`

- La iteración condicional requiere que la condición se pruebe dentro del bucle para determinar si debe continuar
  - llamada **Condición de continuación**

```
while <condition>:
    <sequence of statements>
```

  - El uso inadecuado puede conducir a un **bucle infinito**
- el bucle **`while`** también se llama **bucle de control de entrada**
  - La condición se prueba en la parte superior del bucle
  - Las declaraciones dentro del bucle pueden ejecutarse cero o más veces

# La estructura y el comportamiento de un bucle `while`




[FIGURE 3.5] The semantics of a **`while`** loop

# La estructura y el comportamiento de un bucle `while`

```
suma=0.0
dato=input("Ingrese un dato o enter para salir: ")
while dato!="":
    numero=float(dato)
    suma+=numero
    dato=input("Ingrese un dato o enter para salir: ")
print("La suma es: ",suma)
```

```
Ingrese un dato o enter para salir: 3
Ingrese un dato o enter para salir: 4
Ingrese un dato o enter para salir: 5
Ingrese un dato o enter para salir:
La suma es:  12.0
```

# Bucle `while` controlado por contador

```
suma=0.0
for cont in range(1,100001):
    suma+=cont
print(suma)

suma=0.0
cont=1
while cont<= 100000:
    suma+=cont
    cont+=1
print("La suma es: ",suma)
```

```
for cont in range(10,0,-1):
    print(cont)

cont=10
while cont>= 1:
    print(cont)
    cont-=1
print("La suma es: ",suma)
```

# La sentencia **break** de un bucle

- La declaración break termina el ciclo que lo contiene. El control del programa fluye a la declaración inmediatamente después del cuerpo del bucle.

```
for val in "string":
    if val == "i":
        break
    print(val)

print("The end")
```

# La sentencia **continue** de un bucle

- La instrucción **continue** se usa para omitir el resto del código dentro de un bucle solo para la iteración actual. El bucle no termina pero continúa con la siguiente iteración.

```
for letra in "Python":
    if letra == "h":
        continue
    print ("Letra actual : " + letra)

# Primer ejemplo
var = 10
while var > 0:
    var = var -1
    if var == 5:
        continue
    print ("Valor actual de la variable : " + str(var))
```

# El bucle `while True` y la Sentencia `break`

- **`El bucle while`** puede ser complicado escribirlo correctamente
  - Es posible simplificar su estructura y mejorar su legibilidad.

```
suma=0.0

while True:
    dato=input("Ingrese un numero o enter para salir: ")
    if dato== "":
        break
    numero=float(dato)
    suma+=numero
print("La suma es: ",suma)
```

un bucle **`while True`** con una salida atrasada

condición de terminación del bucle

provoca una salida del bucle

# El bucle `while True` y la Sentencia `break`

```
while True:
    numero=int(input("Ingrese un numero o enter para salir: "))
    if numero>=0 and numero<= 100:
        break
    else:
        print("Error, la nota debe estar entre 0 y 100")
print("Numero valido: ",numero)
```

- Alternativa: Use una variable lógica para controlar el bucle

```
flag=False
while not flag:
    numero=int(input("Ingrese un numero o enter para salir: "))
    if numero>=0 and numero<= 100:
        flag=True
    else:
        print("Error, la nota debe estar entre 0 y 100")
print("Numero valido: ",numero)
```

# Números aleatorios

- Los lenguajes de programación incluyen recursos para generar números aleatorios.
- el módulo **`random`** soporta varias formas de hacer esto
  - **`randint`** devuelve un número aleatorio incluido entre dos argumentos.

```
>>> import random
>>> for x in range(10):
    print(random.randint(1,6),end=" ")


1 6 4 4 1 2 2 1 4 3
>>>
```

- Ejemplo: un simple juego de adivinanzas.

# Números aleatorios

**Ejemplo:** Escriba un programa que permita jugar a la adivinanza. La computadora escogerá un número entre 1 y 100 al azar, luego el usuario tratará de adivinar el número. Cada vez que el usuario ingresa un número ayudarle emitiendo los mensajes “muy alto”, “muy bajo”, o “¡adivinaste!” según sea el caso. Al final imprimir:

a) El número que escogió la computadora

b) La cantidad de intentos del usuario

c) La secuencia de todos los números ingresados en forma horizontal.

```
Valor mas bajo: 1
Valor mas alto: 100
Adivine el numero: 36
Muy bajo
Adivine el numero: 45
Muy bajo
Adivine el numero: 78
Muy alto
Adivine el numero: 65
Muy bajo
Adivine el numero: 76
Adivinaste en  5  intentos!
>>>
```

# Lógica de bucle, errores y pruebas

- Errores a descartar durante la prueba del bucle **`while`**:
  - Variable de control de bucle incorrectamente inicializada
  - Error al actualizar esta variable correctamente dentro del bucle
  - Fallo en probar correctamente en la condición de continuación.
- Para detener el bucle que parece bloquearse durante las pruebas, escriba **Control + C** en la ventana de terminal o el shell IDLE
- Si el bucle debe ejecutarse al menos una vez, use un bucle while True que debe incluir una condición de ruptura en el cuerpo del bucle.
  - Asegure una declaración **break** de modo que sea alcanzada eventualmente.

# Caso de Estudio: Aproximar Raíces Cuadradas

- Requerimiento:
  - Escribir un programa que calcule las raíces cuadradas
- Analisis:

```
Enter a positive number: 3
The program's estimate: 1.73205081001
Python's estimate:      1.73205080757
```

- Diseño:
  - Use el algoritmo aproximación de raíces cuadradas de Newton.
    - Raíz cuadrada *y de un número* positivo *x* es el número **y** tal que $y^2 = x$
    - Si es estimado inicial de *y* es *z*, un mejor estimado de *y* puede ser obtenido tomando el promedio de ***z*** junto con *x*/*z*

# Caso de Estudio: Aproximar Raíces Cuadradas

- Una sesión rápida con el intérprete de Python muestra este método de aproximaciones sucesivas en acción:

```
>>> x = 25
>>> y = 5                    # The actual square root of x
>>> z = 1                    # Our initial approximation
>>> z = (z + x / z) / 2      # Our first improvement
>>> z
13
>>> z = (z + x / z) / 2      # Our second improvement
>>> z
7
>>> z = (z + x / z) / 2      # Our third improvement - got it!
>>> z
5
>>>
```

# Caso de Estudio: Aproximar Raíces Cuadradas

- Diseño: Algoritmo

```
fijar x al valor ingresado por el usuario
fijar la tolerancia a 0.000001
fijar el estimado a 1.0
while True
    fijar el estimado a (estimado + x / estimado) / 2
    fijar la diferencia a abs(x - estimado ** 2)
    if diferencia <= tolerancia:
            break
mostrar el estimado
```

# Caso de Estudio: Aproximar Raíces Cuadradas

```
Ingrese un numero positivo: 136
Valor estimado del programa:  11.661903796158683
Valor estimado de Python:  11.661903789690601
>>>
```

# Resumen

- Una **cadena con formato** y su operador **%** permiten al programador dar formato a los datos usando el ancho de campo y precisión
- Se produce un error cuando el bucle no realiza el número deseado de iteraciones, siendo demasiadas o muy pocas
- Las expresiones booleanas evalúan a **`True`** o **`False`**
  - Se construyen utilizando operadores lógicos: **`and`**, **`or`**, **`not`**
  - Python utiliza la evaluación de cortocircuito en expresiones booleanas compuestas
- Las declaraciones de selección permiten que el programa tome decisiones.

# Resumen

- **`if-else`** es una declaración de selección de dos vías
- La iteración condicional es el proceso de ejecutar un conjunto de declaraciones mientras una condición es verdadera
  - Use el bucle **`while`** (que es un bucle de control de entrada)
- Se puede usar una sentencia **break** para salir del cuerpo de un bucle
- Cualquier bucle for se puede convertir en un bucle while equivalente
- **Bucle infinito:** la condición de continuación nunca se vuelve falsa y no se proporcionan otros puntos de salida
- **`random.randint`** devuelve un número aleatorio

# Resumen

- Las declaraciones de control determinan el orden en el que se ejecutan otras declaraciones en el programa
- La iteración definida es el proceso de ejecución de un conjunto de declaraciones fijas, un número predecible de veces
  - Ejemplo: uso del bucle **`for`**
- el bucle **`for`** consiste en un encabezado y un conjunto de declaraciones llamadas cuerpo
  - Se puede usar para implementar un ciclo controlado por contador
    - Use **`range`** para generar una secuencia de números
  - Puede recorrer y visitar los valores en cualquier secuencia.