Argenitna Programa 4.0

Universidad Nacional de San Luis

Desarrollador Python

Lenguaje de Programación Python: Estructuras de Control y Funciones

Autor:

Dr. Mario Marcelo Berón

UNIDAD 4

LENGUAJE DE PROGRAMACIÓN PYTHON: ESTRUCTURAS DE CONTROL Y FUNCIONES

4.1. Introducción

Esta unidad está dividida en dos partes principales. En la primera se aborda el tema de las estructuras de control: *secuencia*, *selección*, *iteración* y *manejo de excepciones*. En la segunda parte se estudia el tema de funciones definidas por el programador. Dichas funciones permiten empaquetar y parametrizar funcionalidades, reducir el tamaño del código fuente, eliminar el código duplicado y fomentar la reutilización de código.

4.2. Estructuras de Control

En esta sección se describirán diferentes estructuras de control que el programador debe conocer a fin de poder construir programas estructurados. A medida que se avance más en el curso se proporcionarán más he-

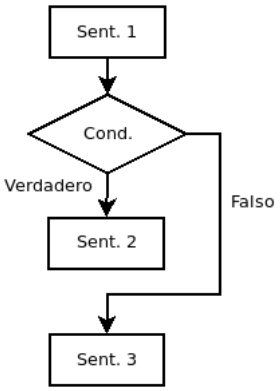


Figura 4.1: Sentencia if-then

rramientas de programación que posibilitarán la construcción de programas más complejos.

4.2.1. Bifurcación Condicional

En esta sección se explicarán diferentes formas de expresar la bifurcación condicional. Se considerarán el formato de bifurcación condicional como sentencia considerando sus diferentes formas y como la bifurcación se puede usar como expresión.

En la figura 4.1 se muestra el diagrama de flujo de una bifurcación condicional (sentencia *if*). El diagrama indica que si la *condición*[1] es verdadera se ejecutan las acciones indicadas por *True* (Verdadero) y si la condición es falsa se ejecutan las acciones indicadas por *False* (Falso).

La bifurcación condicional en Python tiene el siguiente formato:

```
if condición:
    sentencia
```

[1]Por condición se entiende una expresión que evalúa a True (distitno de cero o distinto de vacío en el caso de las estructuras de dato) o False (0 o vacío en el caso de las estructuras de datos). En la condición se pueden colocar expresiones aritméticas, booleanas, relacionales, mixtas.

En este caso si *condición* es verdadera se ejecuta *sentencia* en caso contrario se sigue con el flujo de ejecución normal, es decir con la próxima sentencia en el programa.

if-then

```
>>> a=int(input("Ingrese un número:"))
>>> b=int(input("Ingrese un número:"))
>>> if a > b:
      print("a: ",a," es mayor que b:",b)
```

También la sentencia *if* puede tener el formato que se detalla a continuación:

```
if condición:
 sententencia-1
else:
 sentencia-2
```

En este caso lo que sucede es que si *condición* es verdadera se ejecuta *sententecia-1* en caso contrario, es decir la condición es falsa, se ejecuta *sentencia-2*. Esta situación es la ilustrada en la figura 4.2.

if-then-else

```
>>> a=int(input("Ingrese un número:"))
>>> b=int(input("Ingrese un número:"))
>>> if a > b:
      print("a: ",a," es mayor que b:",b)
    else:
      print("Entre a: ",a," y b:",b," existe otra \
                                    relación")
```

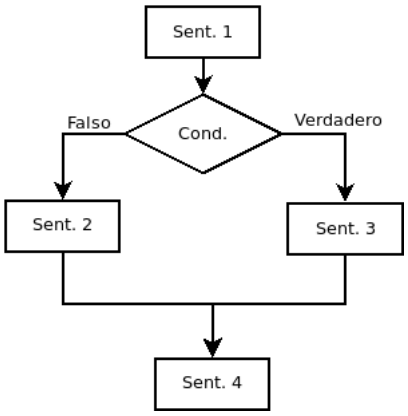
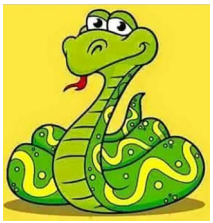



Figura 4.2: Sentencia if-then

La bifurcación condicional, también conocida más comúnmente como sentencia *if* tiene un formato más general que los presentados con anterioridad:

```
if condición-1 :
 sentencia-1
elif condición-2:
 sentencia-2
elif condición-3:
 sentencia-3
 .....
else:
 sentencia-n
```



El comportamiento indicado por esta forma es el siguiente: si *condición-1* es verdadera se ejecuta *sentencia-1*. Caso contrario si *condición-2* es verdadera se ejecuta *sentencia-2*. Caso contrario, si *condición-3* es verdadera se ejecuta *sentencia-3* y así siguiendo. En caso de que ninguna de las condiciones anteriores sea verdadera se ejecutan las sentencias indicadas en el *else* o sea *sentencia-n*.

if-then-else

```
>>> a=int(input("Ingrese un número:"))
>>> b=int(input("Ingrese un número:"))
>>> if a > b:
        print("a: ",a," es mayor que b:",b)
    elif a< b:
        print("a: ",a," es menor que b:",b)
    else:
        print("a: ",a," es igual b:",b)
```

Comentario Importante

La palabra clave *elif*: es la abreviatura de *else if*:. La construcción más general de la sentencia *if* permite reducir el texto que se necesitaría para escribir sentencias *if* anidadas.

Comentario Importante

En algunas ocasiones la bifurcación condicional se puede utilizar como una expresión y escribir de manera más reducida tal como se muestra a continuación:

```
expr.1 if expr._booleana else expr.2
```

En este caso si expr._booleana es verdadera se retorna como resultado el valor devuelto por expr.1 en otro caso se devuelve el valor calculado por expr.2. Este tipo de expresión se utilizan normalmente para asignar valores a variables o bien para retornar valores a funciones. Obviamente, pueden haber otros tipos de usos pero los mencionados con anterioridad son los más comunes.

4.2.2. Ejercicios

Ejercicio 1: Escriba un programa que permita que el usuario ingrese un número el programa debe informar si el número ingresado es par o impar.

Ejercicio 2: Escriba un programa que permita que el usuario ingrese un número n. El programa debe informar si el número es capicúa o no.

Ejercicio 3: Escriba un programa que muestre las siguientes opciones:

1. Calcular el perímetro de un triángulo.
2. Calcular el área de un triángulo.

Luego dependiendo de la opción elegida por el usuario calcule el área o perímetro de un triángulo cuyos datos son también ingresados por el usuario.

Ejercicio 4: Escriba un programa que permita determinar si una ecuación de segundo grado tiene:

1. Dos soluciones reales distintas. Una ecuación de segundo grado tiene dos soluciones si el discriminante $b^2 - 4ac$ es mayor que cero.
2. Una única solución real. Una ecuación de segundo grado tiene dos soluciones si el discriminante $b^2 - 4ac$ es igual que cero.
3. No tiene solución real. Una ecuación de segundo grado tiene dos soluciones si el discriminante $b^2 - 4ac$ es menor que cero.

Nota: Todos los valores requeridos por la ecuación son ingresados por el usuario.

Ejercicio 5: Escriba un programa que permita que el usuario ingrese dos valores numéricos. Luego el programa le pregunta al usuario si quiere sumar, restar dividir o multiplicar y dependiendo de la opción elegida el programa realiza la operación correspondiente.

Ejercicio 6: Escriba un programa que permita que el usuario ingrese el nombre de dos equipos y la cantidad de goles que han hecho en un partido. Luego el programa imprime el nombre del equipo ganador y bien el nombre de los dos equipos en caso de empate.

Ejercicio 7: Escriba un programa que pregunte al usuario su edad y muestre por pantalla si es mayor de edad o no. El programa también le debe solicitar al usuario la edad a partir de la cual se considera una persona mayor de edad.

Ejercicio 8: Escriba un programa que:

1. Almacene una contraseña en una variable.
2. Pregunte al usuario por la contraseña.
3. Imprima por pantalla si la contraseña introducida por el usuario coincide con la guardada en la variable sin tener en cuenta mayúsculas y minúsculas.

Ejercicio 9: Para tributar un determinado impuesto se debe ser mayor de 16 años y tener unos ingresos iguales o superiores a $1000 mensuales.

Escriba un programa que pregunte al usuario su edad y sus ingresos mensuales y muestre por pantalla si el usuario tiene que tributar o no.

Ejercicio 10: Los alumnos de un curso se han dividido en dos grupos A y B de acuerdo al sexo y el nombre. El grupo A esta formado por las mujeres con un nombre anterior a la M y los hombres con un nombre posterior a la N y el grupo B por el resto. Escriba un programa que pregunte al usuario su nombre y sexo, y muestre por pantalla el grupo que le corresponde.

Ejercicio 11: Los tramos impositivos para la declaración de la renta en un determinado país son mostrados en la tabla 4.1.

Renta	Tipo Impositivo
Menos de $10000	5 %
Entre $10000 y $20000	15 %
Entre $20000 y $35000	20 %
Entre $35000 y $60000	30 %
Más de $60000	45 %

Cuadro 4.1: Tramos Impositivos

Escriba un programa que pregunte al usuario su renta anual y muestre por pantalla el tipo impositivo que le corresponde.

Ejercicio 12: En una determinada empresa, sus empleados son evaluados al final de cada año. Los puntos que pueden obtener en la evaluación comienzan en 0.0 y pueden ir aumentando, traduciéndose en mejores beneficios. Los puntos que pueden conseguir los empleados pueden ser 0.0, 0.4, 0.6 o más, pero no valores intermedios entre las cifras mencionadas. A continuación se muestra una tabla con los niveles correspondientes a cada puntuación. La cantidad de dinero conseguida en cada nivel es de $2.400 multiplicada por la puntuación del nivel.

Escriba un programa que lea la puntuación del usuario e indique su nivel de rendimiento, así como la cantidad de dinero que recibirá el

Nivel	Puntuación
Inaceptable	0.0
Aceptable	0.4
Meritorio	0.6 o más

usuario.

Ejercicio 13: Escriba un programa para una empresa que tiene salas de juegos para todas las edades y quiere calcular de forma automática el precio que debe cobrar a sus clientes por entrar. El programa debe preguntar al usuario la edad del cliente y mostrar el precio de la entrada. Si el cliente es menor de 4 años puede entrar gratis, si tiene entre 4 y 18 años debe pagar $5 y si es mayor de 18 años, $10.

Ejercicio 14: La pizzería *Roma* ofrece pizzas vegetarianas y no vegetarianas a sus clientes. Los ingredientes para cada tipo de pizza aparecen a continuación:

1. Ingredientes vegetarianos: Pimiento y tofu.
2. Ingredientes no vegetarianos: Peperoni, Jamón y Salmón.

Escriba un programa que pregunte al usuario si quiere una pizza vegetariana o no, y en función de su respuesta le muestre un menú con los ingredientes disponibles para que elija. Solo se puede elegir un ingrediente además de la mozzarella y el tomate que están en todas las pizzas. Al final se debe mostrar por pantalla si la pizza elegida es vegetariana o no y todos los ingredientes que lleva.

4.3. Iteraciones

En esta sección se describe otra estructura de control que es necesaria para realizar tareas repetitivas. Los lenguajes de programación poseen diferentes estructuras de control que son útiles para distintas situaciones. La sentencia *for* se usa cuando se requiere repetir una tarea n veces. La

sentencia *while* se emplea cuando se requiere hacer una tarea cero o más veces. En las secciones siguientes se explican con más detalles cada una de estas estructuras de control.

4.3.1. Sentencia while

La figura 4.3 muestra la representación visual, utilizando un diagrama de flujo, de la sentencia *while*. En este caso, las *sentencias-while* se ejecutan mientras *condición* sea verdadera. Cuando *condición* es falsa el ciclo termina y se continúa con la próxima sentencia. En general, se parte de una *condición inical* que por lo general consiste en la inicialización de variables que se utilizan en la *condición final*. Luego se evalúa la *condición final* y si el resultado es verdadero se ejecutan las operaciones que se indican en la *sentencia-while* es decir las que forman parte de su cuerpo. Luego se actualizan las variables que se utilizan en la *condición final* con el objetivo de que en algún momento la iteración finalice. Cuando la *condición final* evalúa a *Falso* la ejecución continúa con la próxima sentencia después del *while*.

La sintaxis de la sentencia *while* es la siguiente:

```
while condición:
    sentencia—while
else:
    sententencia—else
```

El comportamiento de la sentencia fue comentado en diagrama de flujo de la sentencia *while*. La característica no contemplada en dicha explicación es la existencia del *else:*. En este caso cuando la sentencia *while* finaliza sin que una sentencia *break* o *return* se ejecute *sentencia-else* caso contrario no lo hacen. En los cuadros siguientes se muestran diferentes ejemplos del uso de la sentencia *while*.

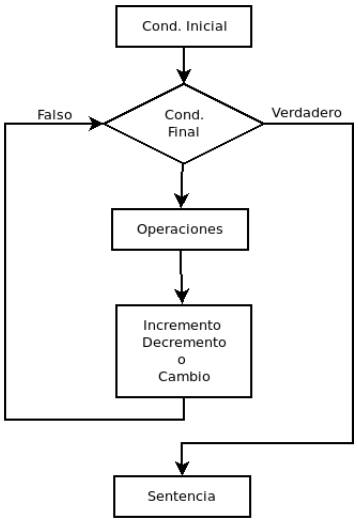


Figura 4.3: Diagrama de Flujo de la Sentencia while

Ejemplo

```
r=0
m=int(input("Ingrese un número:"))
i=1
while i<m:
  r=r + i**2
  i=i+1
print("Suma de Cuadrados:",r)
```

Ejemplo

```
r=0
i=1
while i!=-1:
 m=int(input("Ingrese un número:"))
 if m%2==0:
  r=r+m**2
 else:
  break
 i=int(input("Continúa? (-1 para terminar)"))
else:
 print("Correcto se ingresaron números pares")
print("Suma de Cuadrados de números Pares:",r)
```

Ejemplo

```
dato=""
while dato!="*":
 dato=input("Ingrese un dato:")
print("El ingreso de datos finalizó\
       el usuario ingresó un *")
```

Comentario Importante

La sentencia *while* se utiliza cuando el programador desea realizar tareas cero o más veces.

4.3.2. Sentencia for

La sentencia *for* repite una serie de acciones exactamente n-veces. El diagrama de flujo (ver figura 4.4) es muy similar al de la sentencia *while*.

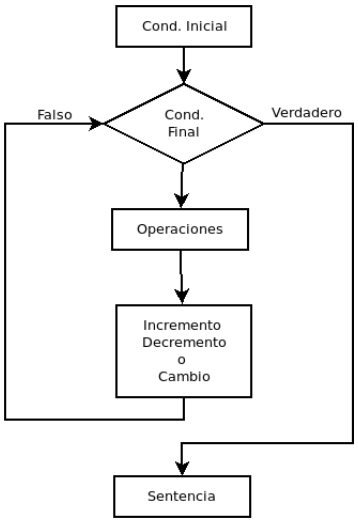


Figura 4.4: Diagrama de Flujo de la Sentencia for

Se parte de una *condición inical* en donde se inicializan las variables que forman parte de la *condición final* si esa condición es verdadera se ejecutan las operaciones dentro del *for* luego se incrementa o decrementa o cambia/n la/s variable/s que forman parte de la condición para que en algún momento produzca como resultado *falso* y la iteración finalice. Si bien los diagramas de flujo son similares las sentencias *while* y *for* se utilizan para diferentes situaciones. La primera de ella se usa cuando las operaciones se repiten 0 o más veces mientras que la sentencia *for* su usa cuando las operaciones se repiten exactamente n veces.

En Python la sintaxis de la sentencia for es la siguiente:

```
for expresión in iterable :
    sentencia_for
else:
    sentencia_else
```

La sentencia *for* en Python amerita una explicación adicional dado que difiere del formato común utilizado en otros lenguajes de programación como por

ejemplo *C*, *C++* y *Java*. En el caso de Python en *expresión* se encuentran las variables a las cuales se le asignarán los valores recuperados desde el *iterable* (strings, listas, tuplas, diccionarios, iteradores) y dichas variables por lo general se utilizarán en *sentencia-for*. A las variables se le seguirán asignando valores hasta que todo el iterable sea recorrido en su totalidad. *sentencia-else* de la claúsula *else* se ejecutará siempre que el *for* termine correctamente, es decir sin finalizar por una sentencia de ruptura como *break* o *return* o se *dispara una excepción*. En los cuadros siguientes se muestran ejemplo de uso de la sentencia *for*.

Ejemplos - Recorridos

```
#Recorrido de un string
for i in "Hola Mundo" :
 print(i)

#Recorrido de una lista
for i in [1,2,3,4,5,6,7]:
 print(i)


#Recorrido de una tupla
for i in (1,2,3,4,5,6,7):
 print(i)
```

Ejemplos - Recorridos

```
#Recorrido de un diccionario por claves
for clave in {1:"Carlos",2:"Martín",3:"Estela"}:
 print(clave)

#Recorrido de un diccionario por valor
for valor in {1:"Carlos",3:"Estela"}.values():
 print(valor)


#Recorrido de un diccionario por clave-valor
for c,v in {1:"Carlos",3:"Estela"}.items():
 print(c,":",v)
```

Ejemplo for-else

```
ln=[]
for i in range(10):
 n=int(input("Ingrese un Número:"))
 if n>0:
  ln.append(n)
 else:
  break
else:
 print("Entrada correcta todos números positivos")
print(ln)
```


4.3.3. Ejercicios

Ejercicio 1: Escriba un programa que permita que el usuario ingrese un número n. El programa debe imprimir los números pares comenzando desde 0 hasta n.

Ejercicio 2: Escriba un programa que imprima por pantalla la suma de los n primeros números naturales.

Ejercicio 3: Escriba un programa que muestre por pantalla la tabla de multiplicar de un número n ingresado por el usuario.

Ejercicio 4: Escriba un programa que permita que el usuario ingrese dos números n y m. El programa debe imprimir por pantalla los números entre 0 y m que son divisibles por n.

Ejercicio 5: Escriba un programa que permita que el usuario ingrese n números por teclado. El programa debe imprimir el mayor y menor número ingresado.

Ejercicio 6: Escriba un programa que permita que el usuario ingrese números por teclado hasta que ingrese un -1. Luego el programa debe informar la cantidad de números ingresados.

Ejercicio 7: Escriba un programa que permita que el usuario ingrese por teclado un string s. El programa deberá contar la cantidad de vocales y consonantes que tiene s.

Ejercicio 8: Escriba un programa que permita comprobar si un string es un palíndromo. Para resolver este ejercicio no realice conversiones.

Ejercicio 9: Escriba un programa que permita que el usuario ingrese n strings. El programa debe imprimir por pantalla el string de mayor longitud.

Ejercicio 10: Escriba un programa que permita que el usuario ingrese dos strings s0 y s1. El programa debe crear un nuevo string denominado merge el cual se forma a partir de s0 y s1 de la siguiente manera: primer

carácter de s0, primer carácter de s1, segundo carácter de s0, segundo carácter de s1 y así siguiendo. Finalmente, el programa imprime s0, s1 y r.

Ejercicio 11: Escriba un programa que permita que el usuario ingrese por teclado una lista l. El programa debe crear dos listas la lista vocales y la lista consonante. En la lista vocales se encuentran todas la vocales que están en s y en la lista consonante todas las consonantes que están en s. Luego el programa debe imprimir por pantalla la cantidad de vocales y la cantidad de consonantes que tiene s.

Ejercicio 12: Escriba un programa que permita que el usuario ingrese por teclado una lista de strings. El programa retorna como resultado la misma lista pero con los strings invertidos.

Ejercicio 13: Escriba un programa que permita que el usuario ingrese una lista l de números. El programa debe informar si la lista l contiene más números positivos que negativos o más números negativos que positivos o contiene la misma cantidad de números positivos que negativos.

Ejercicio 14: Escriba un programa que permita que el usuario ingrese una lista de elementos. El programa debe informar la cantidad de números y strings que contiene la lista.

Ejercicio 15: Escriba un programa que permita almacenar una lista de mercaderías. Los datos requeridos por cada mercadería son: nombre y precio. Dichos datos se almacenan en una tupla donde la primera componente es el nombre de la mercadería y la segunda componente es el precio. El programa debe permitir ingresar mercadería a la lista hasta que el usuario ingrese por teclado un *. Luego el programa debe imprimir por pantalla la lista de mercaderías ingresadas.

Ejercicio 16: Escriba un programa que permita registrar en una lista de tuplas las materias y las notas que un alumno obtuvo durante un trimestre. Luego el programa debe calcular el promedio general del trimestre ingresado.

Ejercicio 17: Escriba un programa que defina un diccionario cuya clave es un número y cuyo valor es una lista de tuplas como la del ejercicio anterior. El programa debe crear un diccionario con tres pares clave: valor. El primero para el primer trimestre, el segundo para el segundo trimestre y el tercero para el tercer trimestre. Luego el programa debe informar el promedio general del alumno.

Ejercicio 18: Cree un diccionario cuya clave sea un string y cuyo valor una lista de números. Luego realice las siguientes actividades:

- Imprima las claves.
- Imprima los valores.

Ejercicio 19: Cree un diccionario cuya clave sea un número y cuyo valor sea un string. Tanto la clave como el valor son requeridos al usuario. Luego el diccionario debe imprimir la clave que tenga como valor el string más largo.

Ejercicio 20: Represente los datos de una persona con un diccionario. Los datos requeridos por personas son: dni, nombre, edad, domicilio, trabajos. Por cada dato de una persona elija el tipo de dato más apropiado con excepción de los trabajos dado que una persona puede tener más de un trabajo entonces los mismos se representan con una tupla. Luego cree una lista de personas las cuales son ingresadas por el usuario. Posteriormente pida al usuario un número de dni e imprima los datos correspondiente a la persona que tenga el dni ingresado por el usuario.

4.4. Excepciones

Las condiciones de error o situaciones excepcionales se pueden informar a través del uso de excepciones. Dicha temática se abordará en las próximas subsecciones.

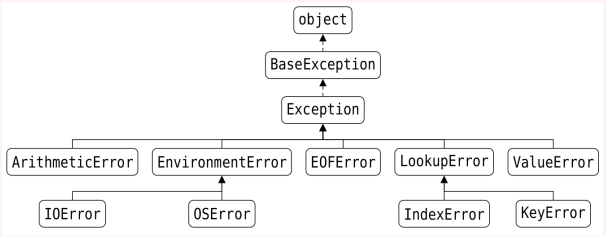
4.4.1. Captura y Generación de Excepciones

Las excepciones se capturan a través de un bloque *try ... except:* el cual tiene la sintaxis que se muestra a continuación:

```
try:
  sentencia-try
except except-grupo-1 as variable-1 :
  sentencia-except-1
  ...
except except-grupo-n as variable-n :
  sentencia-except-n
else:
  sentencia-else
finally:
  sentencia-finally
```

Debe haber al menos un bloque *except* pero los bloques *else* y *finally* son opcionales. El bloque *else* se ejecuta cuando *sentencia-try* se ha ejecutado normalmente es decir no han provocado ninguna excepción. Si hay un bloque *finally* el mismo siempre se ejecuta al final. Cada claúsula *except* puede contener una excepción o una tupla (con paréntesis) de excepciones. Para cada grupo, la parte *as variable* es opcional. Si se utiliza la variable, la misma contiene la excepción que ocurrió y puede ser accedida en el bloque correspondiente. Si una excepción ocurre en el bloque *sentencia-try* se verifican cada claúsula *except*. Si una excepción concuerda con un grupo *except* las sentencias correspondientes a ese grupo se ejecutan. La concordancia se produce cuando la excepción que ocurrió se corresponde con la mencionada en el bloque *except* es decir son del mismo tipo o en todo caso es un subtipo.

Comentario Importante



Si se produce una excepción *KeyError* en una búsqueda en un diccionario la primer claúsula *except* que tenga una excepción *Exception* concordará con *KeyError*. Esto se debe a que *KeyError* es una subclase de *Exception*. Si por el contrario ningún grupo lista a *Exception* pero alguno tiene *LookupError* se producirá una concordancia dado que *KeyError* es una subclase de *LookupError*.

A continuación se describen algunos usos del mecanismo de excepciones.

Captura de Excepciones

```
try:
 dividendo=float(input("Ingrese el dividendo:"))
 divisor=float(input("Ingrese el divisor:"))
 print("Cociente:",dividendo/divisor)
except ValueError as ve:
 print(ve," no se puede convertir a float")
except ZeroDivisionError as zd:
 print("Se ha producido una división por cero")
finally:
 print("Finaliza la ejecución")
```

Colocar una Excepción General y luego una Específica

```
d={1:"Carlos",2:"Estela",3:"Nair",4:"Martín"}

try:
 x = d[5]
except LookupError :
 print("Ocurrió un error de búsqueda")
except KeyError:
 print("Clave Inválida")
```

Si se produjo una excepción *KeyError* su manejador nunca se alcanzará dado que dicha excepción concuerda con *LookupError*.

SIEMPRE SE TIENE QUE UTILIZAR UNA EXCEPCIÓN ESPECÍFICA PARA TENER UNA IDEA ACABADA DEL ERROR QUE SE PRODUJO EN EL PROGRAMA.

Si se produce una excepción y no se encuentra ninguna concordancia entonces Python irá a la función[2] que llamó a la función que produjo la excepción y buscará el manejador allí. En caso de no encontrar el manejador repetirá el proceso y si finalmente no lo encuentra el programa terminará e imprimirá por pantalla la excepción y la traza de ejecución. Como en el caso de las sentencias de iteración, si no se produce ninguna excepción la sentencias asociadas a la cláusula *else* se ejecutarán si la misma está especificada. Las sentencias asociadas a *finally* siempre se ejecutarán al final ocurra o no una excepción. Si ocurre una excepción y la misma no se maneja en ningún *except* las sentencias asociadas al *finally* se ejecutarán y la excepción se propagará

[2]En este caso no hay funciones porque el tema se aborda en la sección siguiente. En consecuencia cuando el manejador no se encuentra en el programa se producirá un error.

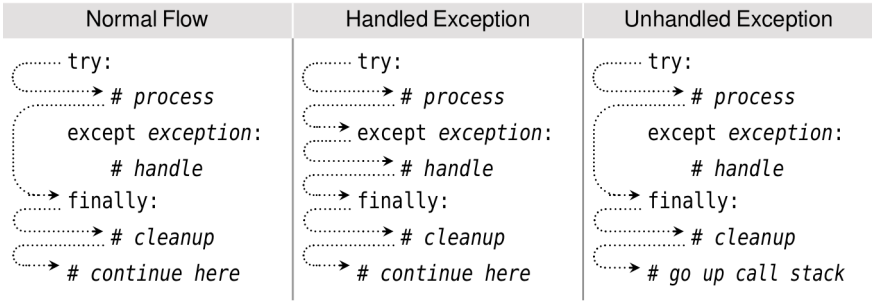


Figura 4.5: Flujo de Ejecución de Excepciones

a la función que invocó a la función que produjo la excepción. La figura 4.5 permite visualizar el flujo de ejecución para: i) El flujo normal; ii) Una excepción manejada; iii) Una excepción sin manejar.

Excepción IndexError no controlada

```
l=[1,2,3,4]
try:
 pos=int(input("Ingrese una posición:"))
 print(l[pos])
except ValueError as ve:
 print("ValueError:",ve)
else:
 print("Todo correcto")
finally:
 print("Fin del programa")
```

4.4.2. Generación de Excepciones

Las excepciones proveen un medio útil de cambiar el flujo de control. Se pueden generar excepciones provistas por Python o bien crear excepciones nuevas. Python provee tres formas de disparar excepciones:

- `raise excepción(argumentos)`
- `raise excepción(argumentos) from excepción_original`
- `raise`

La primer forma se emplea cuando la excepción que se utiliza es propia de Python o es definida por el programador. Si la excepción recibe un argumento textual dicho texto será la salida de la excepción cuando la misma sea tratada. La segunda forma[3] es una variación de la primera y se utiliza cuando una excepción se encadena con otra. Cuando la tercera forma se utiliza la tarea que se lleva a cabo es disparar nuevamente la excepción que se había producido. Si no se ha producido ninguna excepción esta sentencia disparará un error de tiempo de ejecución o un error de tipo dependiendo de la implementación de Python que se esté utilizando.

Disparar una Excepción

```
try:
 v=int(input("Ingrese un valor:"))
 if v<0:
  raise ValueError
except ValueError as e:
 print("Error:",e," es negativo o no es un int")
else:
 print("Fin sin excepciones")
finally:
 print("Fin del Programa")
```

[3]Se menciona que este tema es avanzado y por lo tanto está fuera del alcance de este curso.

Comentario Importante

El programador puede definir sus propias excepciones. No obstante, aunque no es un proceso complicado, se requieren conocimientos de *Programación Orientada a Objetos*. Por esta razón dicha temática se encuentra fuera del alcance de este curso.

4.4.3. Ejercicios

Ejercicio 1: Construya un programa que permita que transforme un string a un número. Controle las situaciones de error usando excepciones.

Ejercicio 2: Escriba un programa que permita insertar, modificar y eliminar elementos de una lista. Controle la situaciones de error usando excepciones.

Ejercicio 3: Escriba un programa que permita que el usuario ingrese una lista de elementos. El programa debe sumar los elementos numéricos que se encuentran en la lista. Cuando el programa encuentra un elemento que no es numérico dispara una excepción ValueError la cual tiene un manejador que inserta el elemento que provocó la excepción en una lista de elementos no numéricos.

Ejercicio 4: Escriba un programa que permita que el usuario cree un diccionario. El programa permite: insertar, eliminar y modificar los elementos del diccionario. Cuando se desea insertar un valor cuya clave ya existe el programa dispara una excepción KeyError cuyo manejador pregunta al usario si realmente desea modificar el valor. En caso afirmativo realiza la modificación correspondiente. En caso negativo el programa no realiza ninguna acción. El programa también debe manejar los errores por intento de acceso al diccionarion y otros que puedan surgir utilizando una clave incorrecta.

Ejercicio 5: Escriba un ejemplo que dispare una excepción, la controle y muestre el uso de finally.

4.5. Funciones

Las funciones son un mecanismo que poseen los lenguajes de programación para empaquetar y parametrizar funcionalidades.

Python provee cuatro clases de funciones:

Funciones Globales: son accesibles desde cualquier código del mismo módulo en donde fue creada y también se pueden acceder desde otros módulos.

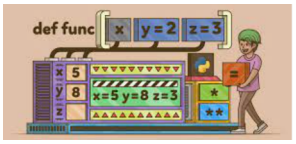
Funciones Locales: son funciones que se definen dentro de otra función.

Funciones lambda: son expresiones con lo cual se pueden crear en el lugar donde se usan. Son más limitadas que las funciones normales.

Métodos: son funciones que están asociadas con un tipo en particular. Forman parte del soporte orientado a objetos del lenguaje Python.

Una función en Python se define como sigue:

```
def nombreDeLaFunción(parámetros):
 cuerpo
```



Los parámetros son opcionales y en el caso de que la función los tenga tienen que estar separados por coma.

Definición de una Función

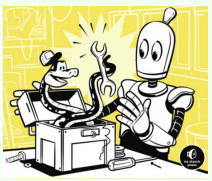
```
def heron (a, b, c):
 s = (a + b + c) / 2
 return math.sqrt(s * (s - a) * (s - b) * (s - c))
```

Dentro de la función los parámetros *a*, *b* y *c* se inicializan con los correspondientes valores con los que se invoca la función, es decir con sus parámetros reales. Cuando la función se invoca *heron(3,4,5)* la *a* se inicializa con 3, *b* con 4 y *c* con 5. Es decir la correspondencia de parámetros es posicional. Si en la invocación no se proporciona el número exacto de parámetros se producirá un error (siempre y cuando la función no reciba parámetros por defecto). Toda función retorna un valor (la sentencia *return* se utiliza para este fin) lo cual es recomendable a ignorar el valor de retorno. Si bien el valor de retorno es uno solo no existen limitaciones sobre el mismo dado que se puede retornar una colección con lo cual la posibilidad de retornar muchos valores envueltos en uno solo es posible. En el caso de que se utilice *return* sin argumentos o que la función no lo utilice el valor de retorno es *None*. Las funciones pueden tener parámetros con valores por defecto es decir parámetros que en caso de no ser provistos en la invocación de la función toman un valor establecido por el programador. Esta tarea se lleva a cabo con la sitnaxis *parámetro=valorPorDefecto*.

Definición de una Función con Parámetro por Defecto

```
def áreaTriángulo (b=1,a=1):
 return (b*a)/2
```

Esta función puede ser invocada de la siguiente manera:



1. áreaTriángulo(2,2) # en este caso calcula (2*2)/2
2. áreaTriángulo(2) # en este caso calcula (2*1)2
3. áreaTriángulo() # en este caso dalcula (1*1)/2

Comentario Importante

La sintaxis de los parámetros no permite que a los parámetros por defecto le sigan parámetros que no son por defecto.

```
def incorrecto(a,b=1,c):
    ....
```

La definición anterior no es correcta dado que luego de un parámetro que tiene valores por defecto como en el caso de b=1 le sigue otro que no lo tiene como es el caso de c.

Python también permite usar correspondencia de parámetros por nombre en lugar de posicional. Es decir en la invocación se indica que valor toman los parámetros explícitamente. Por ejemplo la función *áreaTriángulo* podría ser invocada de la siguiente manera: *áreaTriángulo(b=2,a=1)*.

4.5.1. Nombres y Docstrings

El uso de nombres claros y descriptivos para las funciones y los parámetros permite que otros programadores puedan entender la funcionalidad con más facilidad. A continuación se listan una serie de consejos útiles para escribir funciones.

1. Use un esquema de nombres y úselo consistentemente.
2. *MAYÚSCULAS* para constantes.
3. *NombreDeLaClase* para clases en programación orientada a objetos.
4. *nombredeDeFunción* para funciones y métodos y minúsculas separadas con guiones bajos para cualquier otra cosa.
5. Evitar las abreviaciones.

6. Los nombres deberían ser descriptivos. Por ejemplo el nombre de una variable debería describir el dato que almacena no el tipo.

7. Los nombres de las funciones y de los métodos debería describir lo que ellos hacen pero no como lo hacen.

Además de todos los consejos mencionados con anterioridad a las funciones se le puede incorporar documentación por medio de la utilización de los *docstrings*. Un *docstring* es simplemente un string que comienza después del *def* y antes de la primera sentencia de la función.

Definición de una Función con Parámetro por Defecto

```
def áreaTriángulo (b,a):
    """Calcula el área de un triángulo"""
    return (b*a)/2
```

Utilidad del docstring

Los docstring son útiles para acceder a la descripción que puso el programador respecto de la tarea que realiza la función. Para llevar adelante esta actividad se debe colocar: *nombreFunción.__doc__* y el intérprete mostrará el docstring.

4.5.2. Desempaquetamiento de Argumentos y Parámetros

Se puede usar el operador de desempaquetado * para pasar argumentos a las funciones y también se puede usar en la lista de parámetros. Esta característica es útil cuando se desea crear funciones que utilizan un número variable de argumentos.

Uso de * en los parámetros

```
def producto ( *args ):
 resultado=1
 for arg in args:
  resultado=resultado*arg
 return resultado


producto(1,2,3,4) #args=(1,2,3,4) resultado=24
producto(5,3,8) #args=(5,3,8) resultado=120
producto(11) #args=(11,) resultado=11
```

Se pueden tener argumentos con valores por defecto después del *.

Combinación de * y parámetros por defecto

```
def sumaDePotencias ( *args , potencia=1 ):
resultado=0
for arg in args:
resultado=resultado+arg**potencia
return resultado


#Posibles invocaciones de la función
print(sumaDePotencias(1,3,5))
print(sumaDePotencias(1,3,4,potencia=2))
```

Es posible usar * en la lista de parámetros formales para indicar que no hay más parámetros posicionales después del * pero si pueden haber parámetros con valores por defecto.

Ejemplo de uso de * para indicar fin de parámetros posicionales

```
def heron2 (a, b, c, *, units="metros cuadrados"):
 s = (a + b + c) / 2
 area = math.sqrt(s * (s - a) * (s - b) * (s - c))
 return "{0} {1}".format(area, units)


#Ejemplo de invocaciones
heron2(25, 24, 7)
# retorna: '84.0 metros cuadrados'
heron2(41, 9, 40, units="pulgadas")
# retrona: '180.0 pulgadas'
#Error tiene un argumento posicional extra
heron2(25, 24, 7, "pulgadas")
```

Comentario Importante

Si se coloca * al inicio de la lista de parámetros formales no se pueden pasar parámetros posicionales.
Se debe tener en cuenta que al * le pueden seguir parámetros con valores por defecto.

También se puede usar el operador de desempaquetado de mapeos **[4] en la lista de parámetros de la misma forma que *. Esta característica permite que se puedan crear funciones que acepten muchos argumentos.

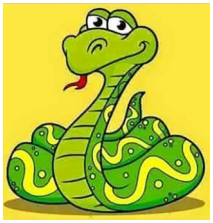
4.5.3. Acceso a las Variables en el Alcance Global

Cuando una función referencia a una variable, la misma se busca en el alcance local es decir dentro de la función. Si la variable no está defi-

[4] El funcionamiento de este operador no se verá en este curso.

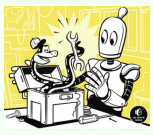
nida en la función la busca en la función que la encierra o en su defecto en el programa. Este proceso continúa hasta que la referencia se resuelve en caso contrario se producirá una excepción *NameError*. Las funciones también tienen una forma de informar que una variable utilizada en la función está definida en el alcance global para realizar esta tarea dentro de la función se escribe:

```
global nombreDeVariable
```



Ejemplo de uso de * para indicar fin de parámetros posicionales

```
operacion="+"
def sumarMultiplicar (x,y):
 resultado=0
 if operacion=="+":
  resultado=x+y
 else:
  resultado=x*y
 return resultado
```



Cuando la función referencia a operación la misma se busca en el cuerpo de la función. Dado que la variable no está definida en la función entonces se busca en el alcance que lo engloba que en este caso es el alcance del programa.

Comentario Importante

No se aconseja el uso de variables globales.

4.5.4. Funciones lambda

Una función *lambda* también conocida como función anónima se define en Python como sigue:

lambda parámetros : expresión

Los parámetros son opcionales y en el caso de que se coloquen tienen que estar separados por comas. La expresión no puede tener bifurcaciones, iteraciones ni sentencias *return*. Las expresiones *lambda* se conocen como *funciones anónimas*. Cuando una función *lambda* se invoca el valor que retorna es el producido por *expresión*. Si expresión es una tupla tiene que estar encerrada entre paréntesis.

Ejemplo de Función Lambda

```
s = lambda x: "" if x == 1 else "s"
```

La función lambda retorna una función la cual se asigna a s luego se puede invocar de la siguiente manera: s(1) la función retornará , s(2) la función retornará "s".

Uso de Funciones Lambda

Las funciones lambda son útiles para cambiar el orden de la función sort.

```
elementos=[(2,12,"ZZ"),(1,11,"aa"),(1,3,"ww"),\
           (2,4,"Be")]
```

Si se desea ordenar por la tercer componente de la tupla entonces:

```
elementos=[(2,12,"ZZ"),(1,11,"aa"),(1,3,"ww"),\
           (2,4,"Be")]
elementos.sort(key=lambda e:(e[2].lower()))
print(elementos)
```

4.5.5. Aserciones

Las sentencias *assert*, por lo general, se utilizan para especificar *precondiciones*[5] y *poscondiciones*[6]. La sintaxis para construir aserciones es la siguiente:

```
assert expresión—boobleana, expresión—opcional
```

Si *expresión-booleana* evalúa a falso se produce una excepción *AssertError*. Si se especifica *expresión-opcional* la misma se utiliza como argumento de la excepción *AssertError* lo cual es muy útil para proveer mensajes de error.

[5] Condiciones que se tienen que cumplir antes de realizar una acción
[6] Condiciones que tienen que ser cumplidas luego de realizada una acción

Ejemplo de Aserción

```
def suma(x=0,y=0):
 assert type(x)==type(1) and type(y)==type(1),\
        "tipo inválido de parámetros"
 return x+y
```

Uso de Funciones Lambda

Una vez que el programa a pasado con éxito todas las aserciones (pre o post condiciones) es posible deshabilitarlas para que no consuman tiempo de ejecución. Esto se logra ejecutando el programa con la opción -O. Es decir de la siguiente manera: *python -O programa.py*. Los *docstrings* también se pueden eliminar ejecutando el programa de la siguiente manera: *python -OO programa.py*.

4.5.6. Ejercicios

Ejercicio 1: Defina la función sumaPar la cual recibe como parámetro una lista l de números enteros y produce como resultado la suma de los números pares que se encuentran en l. Luego construya un programa principal que:

- Permita que el usuario ingrese una lista de números enteros.
- Imprima la suma de los números pares que se encuentran en la lista.

Ejercicio 2: Realice las siguientes actividades:

1. Defina la función pares la cual recibe como parámetro una lista de números enteros l y retorna como resultado una lista que contiene los números pares l.

2. Defina la función impares la cual recibe como parámetro una lista de números enteros l y retorna como resultado una lista de números impares.
3. Defina la función mayoría la cual recibe como parámetro dos listas de números enteros, una que contiene números pares y otra que contiene números impares. La función informa si hay mayoría de números pares o mayoría de números impares o no hay igualdad de números pares e impares.
4. Construya un programa principal que:
   a) Permita que el usuario ingrese una lista de números enteros.
   b) Informe si la lista tiene mayoría de pares o mayoría de impares o tiene la misma cantidad de pares que impares.

Ejercicio 3: Escriba una función que reciba como parámetros una lista de strings. La función crea un diccionario cuya clave son las letras del string y cuyo valor la cantidad de veces que aparece la clave en la lista de string. Luego escriba un programa principal que permita que el usario ingrese una lista de string e imprima el diccionario resultante de aplicar la función a la lista ingresada por el usuario.

Ejercicio 4: Realice las siguientes actividades:

1. Defina la función invertir string la cual recibe como parámetro un string s y retorna como resultado otro string cuyo contenido es el de s pero en orden inverso.
2. Defina la función invertirTodo la cual recibe como parámetro la lista l de strings y retorna como resultado la lista l con los strings invertidos.
3. Construya un programa principal que:
   a) Permita que el usuario ingrese una lista de strings ls.
   b) Inverta los strings de ls.
   c) Imprima ls y ls invertida.

Ejercicio 5: Implemente una calculadora de cuatro funciones. Para llevar adelante esta tarea ud debe:

1. Definir una función suma.
2. Definir una función resta.
3. Definir una función multiplicación.
4. Definir una función división.

El programa debe mostrar un menú de opciones que le de la posibilidad al usuario de elegir la operación deseada y ejecutarla. El programa debe finalizar cuando el usuario ingresa la opción -1. Si el usuario ingresa una opción no válida el programa debe informar la situación y volver a mostrar el menú de opciones.

Ejercicio 6: Dado el siguiente programa:

```
def coordenadaZ(x,y):
 x=x+10
 y=y+15
 return x+y


#programa principal
x=int(input("Coordenada eje x: "))
y=int(input("Coordenada eje y: "))
for i in range(3):
 z=coordenadaZ(x,y)
 x=x+1
 y=y+1
 print(x," . ",y," . ",z)
```

Se pide:

1. De un ejemplo de ejecución del programa.
2. Diga que hace el programa.

Ejercicio 7: Dado el siguiente programa:

```
def maximo(x,y):
 return x if x>y else: y


def minimo(x,y):
 return x if x<y else: y


#programa principal
x=int(input("Un número: "))
y=int(input("Otro número: "))
print(maximo(x-3, minimo(x+2, y-5)))
```

Se pide:

1. De un ejemplo de ejecución del programa.
2. Diga que hace el programa.
3. Critique la organización del código.

Ejercicio 8: Escriba una función que, dado un número de DNI, retorne True si el número es válido y False si no lo es. Para que un número de DNI sea válido debe tener entre 7 y 8 dígitos en base 10. Luego escriba un programa que pruebe la función.

Ejercicio 9: Implemente un padrón de personas. Por cada persona se almacena el nombre, dni y domicilio. El programa debe permitir que el usuario pueda:

1. Incorporar personas al padrón.
2. Eliminar personas del padrón.
3. Modificar los datos de una persona en el padrón.
4. Imprimir por pantalla los datos de una persona específica.

El programa debe mostrar por pantalla un menú de opciones con los ítems descritos anteriormente. El programa finalizará si el usuario ingresa como opción un 0.

Ejercicio 10 Construya una función que reciba como parámetro dos valores si esos valores son enteros la función retorna como resultado la suma de dichos valores. Si los valores son strings la función retorna como resultado la concatenación de los strings. En otro caso la función dispara una excepción ValueError.

Ejercicio 11: Deina la función porcentaje la cual recibe como parámetro una lista de números flotantes l y un número entero p. La función retorna como resultado el porcentaje p de la suma de los elementos de l. La función recibe como parámetro por defecto la lista vacía.

Ejercicio 12: Implemente las siguientes funciones:

- Factorial(n)= $1 \times 2 \times 3 \times ..... \times n-1 \times n$
- Fibonacci(n) la cual se define como sigue:
  - Fibonacci(0)=0
  - Fibonacci(1)=1
  - Fibonacci(n)=Fibonacci(n-1)+Fibonacci(n-2).
- Potencia(n,m)=n^m

Luego construya un programa principal que permita probar las funciones. Esto es invocarlas con argumentos correctos e incorrectos. Maneje los errores que surgen por la invocación de argumentos incorrectos con excepciones.