

Tema 0 - Introducción a python- 2

Semestre 2018-1

M. en C. Gustavo Contreras Mayén

M. en C. Abraham Lima Buendía

Facultad de Ciencias - UNAM

30 de enero de 2020



1. Instrucciones de entrada y salida
2. Estructuras de control
3. Manejo de excepciones

1. Instrucciones de entrada y salida

1.1 Entrada de datos

1.2 Salida de datos

2. Estructuras de control

3. Manejo de excepciones

Ingreso de datos en un código

Una buena parte de la solución de problemas mediante el uso de un lenguaje, se realiza con los valores que se hayan ingresado previamente en el algoritmo.

Pero será muy común que proporcionemos los valores manualmente, para que se ejecuten los cálculos.


Entrada estándar en un programa

Se le denomina entrada estándar al procedimiento en el cual se ingresan valores en un programa.

El dispositivo estándar de entrada de una computadora, es el teclado. Existen otros dispositivos de entrada tales como los puertos serial, paralelo, usb, etc.


Instrucciones de entrada con python

Existen dos maneras de ingresar valores a un programa de python.

El usuario al concluir el ingreso de la información, debe de presionar la tecla Enter , para que continúe la ejecución de las instrucciones.

La función `input()`

La función `input()` permite obtener texto escrito por teclado.

En la llamada a la función, el programa se detiene esperando que se escriba algo y se pulse la tecla Enter , como muestra el siguiente ejemplo:

La función input()

```
>>> nombre = input("¿Cómo te llamas? ")
¿Cómo te llamas? Gustavo
>>> print("Mucho gusto en conocerte, ", nombre)
Mucho gusto en conocerte, Gustavo
```


Conversión de tipos de datos

De forma predeterminada, la función `input()` convierte la entrada en un tipo de dato `cadena`.

Si se quiere que python interprete la entrada a un tipo de dato en particular, se debe de usar la respectiva función de conversión.

Funciones de conversión

La función `int()` convierte a un tipo de dato entero.

```
>>>a = int(input("Ingrese un entero"))
```

Funciones de conversión

La función `int()` convierte a un tipo de dato entero.

```
>>>a = int(input("Ingrese un entero"))
```

La función `float()` convierte a un tipo de dato real.

```
>>> b = float(input("Ingrese un real"))
```

Salida de datos en python.

La salida estándar de datos en python es a través de la pantalla.

Otros tipos de salida son: la impresora, a un archivo de datos, a un plotter, etc.

La función `print()`

Hasta ahora hemos usado la función `print()` para mostrar información en la terminal.

Frecuentemente necesitaremos más control sobre el formateo de la salida que simplemente imprimir valores separados por espacios.

Hay dos maneras de formatear la salida de datos:

- 1 La primera es hacer todo el manejo de las cadenas manualmente: usando rebanado de cadenas y operaciones de concatenado.

El tipo `string` contiene algunos métodos útiles para emparejar cadenas a un determinado ancho.

Hay dos maneras de formatear la salida de datos:

- 1 La primera es hacer todo el manejo de las cadenas manualmente: usando rebanado de cadenas y operaciones de concatenado.

El tipo `string` contiene algunos métodos útiles para emparejar cadenas a un determinado ancho.

- 2 La otra forma es usar “formatted string literals” o el método `str.format()`.

El módulo `string` contiene una clase `string.Template` que ofrece otra forma de sustituir valores en las cadenas.

Nos queda una pregunta, por supuesto: ¿cómo convertir valores a cadenas?

El módulo `string` contiene una clase `string.Template` que ofrece otra forma de sustituir valores en las cadenas.

Nos queda una pregunta, por supuesto: ¿cómo convertir valores a cadenas?

Afortunadamente, `python` tiene maneras de convertir cualquier valor a una cadena: con las funciones `repr()` o `str()`.

La función `str()` devuelve representaciones de los valores que son bastante legibles para los usuarios.

Mientras que `repr()` genera representaciones que pueden ser leídas por el el intérprete (o forzarían un `SyntaxError` si no hay sintaxis equivalente).

Para objetos que no tienen una representación en particular para consumo humano, `str()` devolverá el mismo valor que `repr()`.

Muchos valores, como números o estructuras como listas y diccionarios, tienen la misma representación usando cualquiera de las dos funciones. Las cadenas, en particular, tienen dos representaciones distintas.

Ejemplos I

```
>>> s = 'Hola mundo.'
>>> str(s)
'Hola mundo.'
>>> repr(s)
"'Hola mundo.'"
>>> str(1 / 7)
'0.142857142857'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'El valor de x es ' + repr(x) + ', y es ' + repr(y)
>>> print(s)
El valor de x es 32.5, y es 40000...
```

Ejemplos II

Ejemplos

```
>>> # El repr() de una cadena agrega apostrofes y barras in
... hola = 'hola mundo\n'
>>> holas = repr(hola)
>>> print(holas)
'hola mundo\n'
>>> # El argumento de repr() puede ser cualquier objeto de
... repr((x, y, ('optica', 'cuantica'))))
"(32.5, 40000, ('optica', 'cuantica'))"
```

Ejemplo con un bucle

```
>>> for x in range(1, 11):  
...     print(repr(x).rjust(2), repr(x * x).rjust(3), end='  ')  
...     # revisa el uso de 'end' en la linea anterior  
...     print(repr(x * x * x).rjust(4))  
...
```

Salida en la terminal

1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

Observaciones

Vemos en la salida que hay un espacio entre cada columna, esto debido a la manera en que la función `print()` trabaja: siempre agrega espacios entre sus argumentos.

Observaciones

Vemos en la salida que hay un espacio entre cada columna, esto debido a la manera en que la función `print()` trabaja: siempre agrega espacios entre sus argumentos.

Este ejemplo muestra el método `str.rjust()` de los objetos cadena, el cual ordena una cadena a la derecha en un campo del ancho dado llenándolo con espacios a la izquierda.

Hay métodos similares `str.ljust()` y `str.center()`.

Estos métodos no escriben nada, sólo devuelven una nueva cadena.

Otra manera de presentar la salida

```
>>> for x in range(1,11):  
...   print('{0:2d} {1:3d} {2:4d}'.format(x, x * x, x * x * x))  
...
```

Formatos de salida

s	Tipo cadena, es el tipo por defecto.
d	Decimales enteros, en base 10.
e	Notación exponencial, la letra <i>e</i> indica el exponente.
E	Notación exponencial, la letra <i>E</i> es el separador.
f	De punto fijo, muestra un número con punto fijo.

1. Instrucciones de entrada y salida

2. Estructuras de control

2.1 Condicionales

2.2 Bucles o Loops

2.3 Sentencia `for ... in`

2.4 El bucle `while`

3. Manejo de excepciones

Estructuras de control

En cualquier lenguaje de programación se incluye una serie de estructuras de control para ampliar las posibilidades de ejecución de un programa.

En `python`, manejaremos las más comunes, que son relativamente sencillas de usar, cuidado siempre la sintaxis respectiva.

Condicionales

Una sentencia condicional permite ejecutar una serie de instrucciones si se cumple que cierta condición tenga un valor **True**.

En caso de que el valor de la condición no se cumpla, es decir, tiene un valor **False**, no se ejecutan la instrucciones contenidas y pasa a la siguiente línea de código.

El condicional if

El condicional `if` requiere de una expresión inicial que va a evaluar, como ya se mencionó, en caso de que no se cumpla el valor `True`, no se ejecutan las instrucciones contenidas.

```
if expresion:  
    instruccion1  
    instruccion2
```

siguiente línea código

El condicional if compuesto

El condicional `if` puede ocuparse en el caso de que si una primera expresión no se cumple, se pueda evaluar una segunda expresión mediante

`elif expresion2 :`

Si `expresion2` devuelve un valor `True`, entonces se ejecutan las instrucciones contenidas en el bloque.

El condicional elif

```
if expresion:  
    instruccion1  
    instruccion2  
elif expresion2:  
    instruccion-1  
    instruccion-2
```

siguiente línea código

El bloque else:

Cuando tenemos un condicional con `if` y adicionalmente el `elif`: pudiera ser que las dos expresiones tengan un valor `False`.

Por lo que podemos recurrir a una instrucción `else`: que no evalúa expresión alguna, solo permite que se ejecuten las instrucciones contenidas dentro de este bloque.

El bloque else:

```
if expresion1:
    instruccion1
    instruccion2
elif expresion2:
    instruccion-1
    instruccion-2
else:
    instruccion-else-1
    instruccion-else-2
```

Ejemplo de condicional

```
a = int(input('Introduce el valor de a'))  
if a > 0:  
    print ("a es positivo")  
    a = a + 1  
elif a == 0:  
    print ("a es 0")  
else:  
    print ("a es negativo")
```

Un bucle es una sentencia que evalúa inicialmente una condición, en caso de que se cumple (valor **True**) se ejecuta un conjunto de instrucciones.

Posteriormente, se revisa el valor de la condición, mientras sea verdadero, las instrucciones se ejecutan nuevamente, el bucle termina hasta que el valor de la condición sea un valor **False**.

Hay que considerar que se puede conocer de antemano, el número de veces que se va a repetir el ciclo, pero hay que ser cuidadosos y evitar los bucles infinitos, es decir, sentencias que no modifican el valor de la condición y por tanto, siempre se mantendrá sin salir del bucle.

Sentencia for ... in

Es una forma genérica de iterar sobre una secuencia.

Podemos usar como secuencia: tanto listas como tuplas o generar una para ejecutar el bucle un número determinado de veces.

Ejemplo

```
secuencia = ["uno", "dos", "tres"]  
for elemento in secuencia:  
    print (elemento)
```

Ejemplo

```
secuencia = ["uno", "dos", "tres"]  
for elemento in secuencia:  
    print (elemento)
```

En este ejemplo la instrucción `print` se ejecutará tantas veces como elementos haya en la lista, y en cada iteración la variable `elemento` tomará el valor de cada uno de los elementos de la lista `secuencia`.

Iteración sobre secuencia de números

¿Como se hace para iterar sobre una serie de números naturales consecutivos?

Por ejemplo de 1 a 20.

Iteración sobre secuencia de números

¿Como se hace para iterar sobre una serie de números naturales consecutivos?

Por ejemplo de 1 a 20.

Para ello usaremos la función `range()`. Esta función genera una lista con una progresión aritmética de números naturales.

Argumentos de la función range

- 1 Si le pasamos un único parámetro a generará una lista que va desde 0 hasta $n - 1$.

Argumentos de la función range

- ① Si le pasamos un único parámetro a generará una lista que va desde 0 hasta $n - 1$.
- ② Si le damos dos argumentos, generará una lista desde el primero hasta el segundo menos uno.

Argumentos de la función range

- ➊ Si le pasamos un único parámetro a generará una lista que va desde 0 hasta $n - 1$.
- ➋ Si le damos dos argumentos, generará una lista desde el primero hasta el segundo menos uno.
- ➌ Si le damos tres, usará el tercero como incremento para generar los elementos de la lista.

Ejemplos de for ... in

```
>>> print(list(range(10)))  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> print(list(range(5, 10)))  
[5, 6, 7, 8, 9]  
>>> print(list(range(0, 10, 3)))  
[0, 3, 6, 9]
```

Ejemplos de for ... in

En el siguiente ejemplo el programa imprimirá los números de 0 a 5 cada uno en una línea.

```
for i in range(6):  
    print(i)
```

Ejemplos de for ... in

Podemos usar una cadena de caracteres como secuencia, de forma que cada iteración del bucle tomaremos una letra de la cadena.

```
for char in 'ABCD':  
    print(char)
```

Ejemplos de for ... in

En caso de que necesitemos iterar sobre una lista y a la vez tener el índice de cada posición de la lista usaremos la función `enumerate()` que devuelve dos valores: la posición y el contenido de la lista.

```
secuencia = ["manzanas", "peras", "platanos"]  
for posicion, elemento in enumerate(secuencia):  
    print (posicion, elemento)
```

El bucle while

Ese bucle repite un conjunto de instrucciones mientras se cumpla una determinada condición que se evalúa al principio de cada ejecución.

Es evidente que las instrucciones del interior del bucle tendrán que hacer algo que pueda cambiar esa condición hasta que en algún momento deje de cumplirse.

En caso contrario tendremos un bucle infinito y el programa no terminará nunca su ejecución.

El bucle `while`

Una de las características del bucle `while` es que no está fijado previamente el número de veces que se ejecutan las instrucciones del bucle.

Se ejecutarán todas las que sean necesarias mientras se cumpla la condición.

Forzar la salida del bucle `while`

Como hemos mencionado, el ciclo `while` va a iterar mientras se cumpla una condición.

Pero vamos a encontrar que en ocasiones, necesitamos “salir” del bucle sin que tengamos que esperar a que la condición cambie.

Ejemplos while

Hay dos palabras reservadas que se usan dentro de un bucle, se trata de `break` y `continue`.

`continue` hace que pasemos de nuevo al principio del bucle aunque no se haya terminado de ejecutar el ciclo anterior.

Ejemplos while

Listing 1: Ejemplo con continue

```
edad = 0
while edad < 18:
    edad = edad + 1
    if edad % 2 == 0:
        continue
    print ("Felicidades, tienes " + str(edad))
```

Ejemplos while

Por su parte `break` hace que salgamos del bucle `while` directamente sin necesidad de volver a evaluar la condición y aunque siga siendo cierta.

Ejemplos while

Listing 2: Ejemplo con break

```
while True:
    entrada = input("> ")
    if entrada == "adios":
        break
    else:
        print (entrada)
```

1. Instrucciones de entrada y salida

2. Estructuras de control

3. Manejo de excepciones

3.1 Control de errores

Manejo de excepciones

Control de errores

Cuando comenzamos a programar, nos podemos encontrar con mensajes de error al momento de ejecutar el programa, siendo las causas más comunes:

Manejo de excepciones

- Errores de dedo, escribiendo incorrectamente una instrucción, sentencia, variable o constante.

Manejo de excepciones

- Errores de dedo, escribiendo incorrectamente una instrucción, sentencia, variable o constante.
- Errores al momento de introducir los datos, por ejemplo, si el valor que se debe de ingresar es 123.45, y si nosotros tecleamos 1234.5, el resultado ya se considera un error.

Manejo de excepciones

- Errores que se muestran en tiempo de ejecución, es decir, todo está bien escrito y los datos están bien introducidos, pero hay un error debido a la lógica del programa o del método utilizado, ejemplo: división entre cero.

Manejo de errores

En el siguiente ejemplo, obtendremos de antemano un error por intentar una operación matemática no permitida.

```
c = 12.0/0.0
```

Manejo de errores

En el siguiente ejemplo, obtendremos de antemano un error por intentar una operación matemática no permitida.

```
c = 12.0/0.0
```

```
Traceback (most recent call last):
```

```
File '<pyshell#0>', line 1, in ?
```

```
c = 12.0/0.0
```

```
ZeroDivisionError: float division
```

Manejo de excepciones

Veamos el siguiente ejemplo

```
>>> while True: print('Hola mundo')
```

Manejo de excepciones

Veamos el siguiente ejemplo

```
>>> while True print('Hola mundo')
```

```
File "<stdin>", line 1
```

```
    while True print('Hola mundo')
```

```
        ^
```

```
SyntaxError: invalid syntax
```

Información del error

El intérprete repite la línea culpable y muestra una pequeña “flecha” que apunta al primer lugar donde se detectó el error.

Este es causado por (o al menos detectado en) el símbolo que precede a la flecha: en el ejemplo, el error se detecta en la función **print()**, ya que faltan dos puntos **(:)** antes del mismo.

Se muestran el nombre del archivo y el número de línea para que sepas dónde mirar en caso de que la entrada venga de un programa

Tipos de error en python

Es importante conocer los distintos tipos de error que pueden generarse en python, en la documentación oficial, podremos encontrar una lista con el nombre del tipo de error y por qué se genera.

Errores aritméticos

① OverflowError

Errores aritméticos

- ① OverflowError
- ② ZeroDivisionError

Errores aritméticos

- ① OverflowError
- ② ZeroDivisionError
- ③ FloatingPointError

Errores generales

① ImportError

Errores generales

- ① ImportError
- ② IndexError

Errores generales

- ① ImportError
- ② IndexError
- ③ KeyboardInterrupt

Errores generales

- ① ImportError
- ② IndexError
- ③ KeyboardInterrupt
- ④ NameError

Errores generales

- ① ImportError
- ② IndexError
- ③ KeyboardInterrupt
- ④ NameError
- ⑤ SyntaxError

Errores generales

- ① ImportError
- ② IndexError
- ③ KeyboardInterrupt
- ④ NameError
- ⑤ SyntaxError
- ⑥ TabError

Errores generales

- ① ImportError
- ② IndexError
- ③ KeyboardInterrupt
- ④ NameError
- ⑤ SyntaxError
- ⑥ TabError
- ⑦ ValueError

Manejando excepciones

Es posible escribir programas que manejen determinadas excepciones.

En el siguiente ejemplo, se le pide al usuario una entrada hasta que ingrese un entero válido, pero permite al usuario interrumpir el programa (usando Control-C o lo que sea que el sistema operativo soporte)

Manejando excepciones

```
while True:
    try:
        x = int(input("Por favor ingrese un numero:"))
        break
    except ValueError:
        print("Oops! No era valido. Intente nuevamen
```

La declaración try

La declaración try funciona de la siguiente manera:

- Primero, se ejecuta el bloque **try** (el código entre las declaraciones **try** y **except**).

La declaración try

La declaración try funciona de la siguiente manera:

- Primero, se ejecuta el bloque **try** (el código entre las declaración **try** y **except**).
- Si no ocurre ninguna excepción, el bloque **except** se salta y termina la ejecución de la declaración **try**.

La declaración `try`

- Si ocurre una excepción durante la ejecución del bloque `try`, el resto del bloque se salta. Luego, si su tipo coincide con la excepción nombrada luego de la palabra reservada `except`, se ejecuta el bloque `except`, y la ejecución continúa luego de la declaración `try`.

La declaración try

- Si ocurre una excepción que no coincide con la excepción nombrada en el **except**, esta se pasa a declaraciones **try** de más afuera; si no se encuentra nada que la maneje, es una excepción no manejada, y la ejecución se frena con un mensaje como los mostrados arriba.