

Técnicas de Programación

Instituto de Formación Técnica Superior Nro. 11
Docente: Lic. Norberto A. Orlando



Índice

Funciones	5
¿Por qué necesitamos funciones?	5
Descomposición.....	7
¿De dónde provienen las funciones?.....	7
Tu primera función	8
Cómo funcionan las funciones	11
Cómo las funciones se comunican con su entorno	12
Funciones parametrizadas	12
Paso de parámetros posicionales	15
Paso de argumentos de palabra clave	16
Mezclando argumentos posicionales y de palabras clave	17
Funciones parametrizadas - más detalles.....	18
Retornando el resultado de una función	19
Efectos y resultados: la instrucción return	19
return sin una expresión	19
return con una expresión	20
Unas pocas palabras sobre None	21
Efectos y resultados: listas y funciones	22
Los alcances en Python	24
Funciones y alcances.....	24
Funciones y alcances: la palabra clave global	26
Cómo interactúa la función con sus argumentos.....	27
Creación de funciones multiparámetro.....	28
Ejemplos de funciones: Cálculo del IMC	28
Calcular el IMC y convertir unidades del Sistema Inglés al Sistema Métrico	28
Ejemplos de funciones: Triángulos	29
Triángulos y el Teorema de Pitágoras	30
Ejemplos de funciones: Factoriales	31

Números Fibonacci	32
Recursividad	33
Tuplas y diccionarios.....	34
Tipos de secuencia y mutabilidad.....	34
Tuplas	35
¿Cómo crear una tupla?	36
¿Cómo utilizar un tupla?	36
Diccionarios	38
¿Cómo crear un diccionario?	38
¿Cómo utilizar un diccionario?	40
Métodos y funciones de los diccionarios.....	41
El método keys()	41
Modificar, agregar y eliminar valores	42
La función sorted()	43
Agregando nuevas claves	43
Eliminado una clave	44
Las tuplas y los diccionarios pueden trabajar juntos	45
Excepciones	46
Errores - el pan diario del desarrollador	47
Errores en los datos frente a errores en el código.....	47
Cuando los datos no son lo que deberían ser	48
La rama try-except	49
La excepción confirma la regla	50
Cómo lidiar con más de una excepción	50
Dos excepciones después de un try	51
La excepción predeterminada y cómo usarla	51
Algunas excepciones útiles	52
ZeroDivisionError	52
ValueError.....	52
TypeError	52

AttributeError	52
SyntaxError	53
Por qué no puedes evitar probar tu código	53
Rastreando las rutas de ejecución	54
Pruebas, pruebas y más pruebas.....	54
Error frente a depuración (Bug vs. debug)	55
print debugging (depuración).....	55
Algunos consejos útiles.....	56
Pruebas unitarias - un mayor nivel de codificación.....	57

UNIDAD 4: Funciones, Tuplas, Diccionarios, Excepciones y Procesamiento de Datos

Aprenderás los conceptos básicos de funciones, así como la programación estructural y funcional; aprende a trabajar con tuplas y diccionarios, y cómo manejar excepciones y depurar código en Python

Funciones

Aprenderás a crear, usar, y mandar llamar a tus propias funciones.

¿Por qué necesitamos funciones?

Hasta ahora has implementado varias veces el uso de **funciones**, pero solo se han visto algunas de sus ventajas. Solo se han invocado funciones para utilizarlas como herramientas, con el fin de hacer la vida más fácil, y para simplificar tareas tediosas y repetitivas.

Cuando se desea mostrar o imprimir algo en consola se utiliza **print()**. Cuando se desea leer el valor de una variable se emplea **input()**, combinados posiblemente con **int()** o **float()**.

También se ha hecho uso de algunos **métodos**, los cuales también son funciones, pero declarados de una manera muy específica.

Ahora aprenderás a escribir tus propias funciones, y como utilizarlas. Escribiremos varias de ellas juntos, desde muy sencillas hasta algo complejas. Se requerirá de tu concentración y atención.

Muy a menudo ocurre que un cierto fragmento de código se repite muchas veces en un programa. Se repite de manera literal o, con algunas modificaciones menores, empleando algunas otras variables dentro del programa. También ocurre que un programador ha comenzado a copiar y pegar ciertas partes del código en más de una ocasión en el mismo programa.

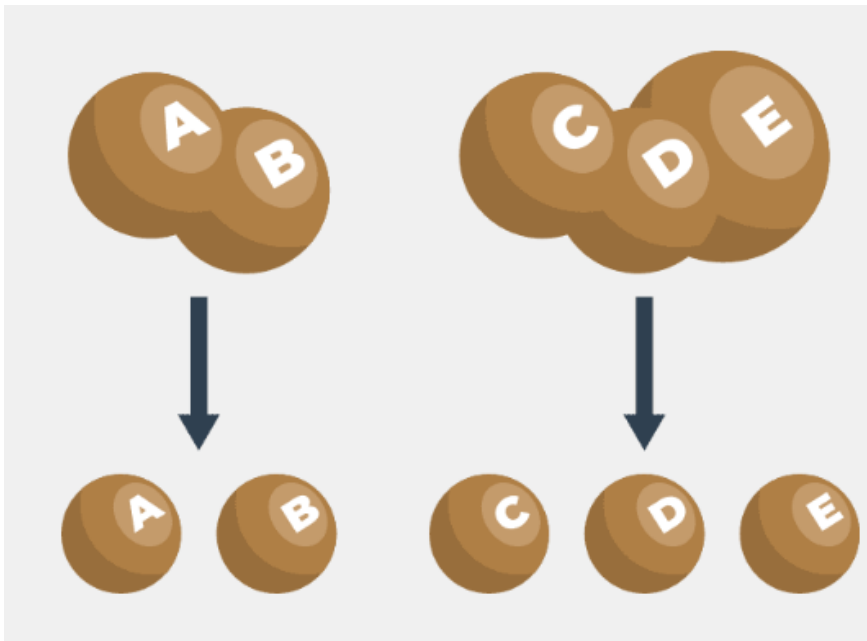
Puede ser muy frustrante percatarse de repente que existe un error en el código copiado. El programador tendrá que escarbar bastante para encontrar todos los lugares en el código donde hay que corregir el error. Además, existe un gran riesgo de que las correcciones produzcan errores adicionales.

Definamos la primera condición por la cual es una buena idea comenzar a escribir funciones propias: **si un fragmento de código comienza a aparecer en más de una ocasión, considera la posibilidad de aislarlo en la forma de una función invocando la función desde el lugar en el que originalmente se encontraba.**

Puede suceder que el algoritmo que se desea implementar sea tan complejo que el código comience a crecer de manera incontrolada y, de repente, ya no se puede navegar por él tan fácilmente.

Se puede intentar solucionar este problema comentando el código, pero pronto te darás cuenta que esto empeorará la situación - **demasiados comentarios hacen que el código sea más difícil de leer y entender.** Algunos dicen que **una función bien escrita debe ser comprensible con tan solo una mirada.**

Un buen desarrollador **divide el código** (o mejor dicho: el problema) en piezas aisladas, y **codifica cada una de ellas en la forma de una función.**



Esto simplifica considerablemente el trabajo del programa, debido a que cada pieza se codifica por separado, y consecuentemente se prueba por separado. A este proceso se le llama comúnmente **descomposición.**

Existe una segunda condición: **si un fragmento de código se hace tan extenso que leerlo o entenderlo se hace complicado, considera dividirlo en pequeños problemas, por separado, e implementa cada uno de ellos como una función independiente.**

Esta descomposición continúa hasta que se obtiene un conjunto de funciones cortas, fáciles de comprender y probar.

Descomposición

Es muy común que un programa sea tan largo y complejo que no puede ser asignado a un solo desarrollador, y en su lugar un **equipo de desarrolladores** trabajarán en él. El problema, debe ser dividido entre varios desarrolladores de una manera en que se pueda asegurar su eficiencia y cooperación.

Es inconcebible que más de un programador deba escribir el mismo código al mismo tiempo, por lo tanto, el trabajo debe de ser dividido entre todos los miembros del equipo.

Este tipo de descomposición tiene diferentes propósitos, no solo se trata de **compartir el trabajo**, sino también de **compartir la responsabilidad** entre varios desarrolladores.

Cada uno debe escribir un conjunto bien definido y claro de funciones, las cuales al ser **combinadas dentro de un módulo** (esto se clarificará un poco más adelante) nos dará como resultado el producto final.



Esto nos lleva directamente a la tercera condición: si se va a dividir el trabajo entre varios programadores, **se debe descomponer el problema para permitir que el producto sea implementado como un conjunto de funciones escritas por separado empacadas juntas en diferentes módulos.**

¿De dónde provienen las funciones?

En general, las funciones provienen de al menos tres lugares:



De Python mismo: varias funciones (como `print()`) son una parte integral de Python, y siempre están disponibles sin algún esfuerzo adicional del programador; se les llama a estas **funciones integradas**.



De los módulos preinstalados de Python: muchas de las funciones, las cuales comúnmente son menos utilizadas que las integradas, están disponibles en módulos instalados juntamente con Python; para poder utilizar estas funciones el programador debe realizar algunos pasos adicionales (se explicará mas adelante este tema).



Directamente del código: tu puedes escribir tus propias funciones, colocarlas dentro del código, y usarlas libremente.

Tu primera función

Observa el fragmento de código en el editor.


```
1 print("Ingresa un valor: ")
2 a = int(input())
3
4 print("Ingresa un valor: ")
5 b = int(input())
6
7 print("Ingresa un valor: ")
8 c = int(input())
9
10
```

Console >_
Ingresa un valor:
2
Ingresa un valor:
3

Es bastante sencillo, es un ejemplo de **como transformar una parte de código que se está repitiendo en una función**.

El mensaje enviado a la consola por la función **print()** es siempre el mismo. El código es funcional y no contiene errores, sin embargo imagina que tendrías que hacer si tu jefe pidiera cambiar el mensaje para que fuese más cortés, por ejemplo, que comience con la frase "Por favor,".

Tendrías que tomar algo de tiempo para cambiar el mensaje en todos los lugares donde aparece (podrías hacer uso de copiar y pegar, pero eso no lo haría más sencillo). Es muy probable que cometas errores durante el proceso de corrección, eso traería frustración a ti y a tu jefe.

¿Es posible separar ese código repetido, darle un nombre y hacerlo reutilizable? **Significaría que el cambio hecho en un solo lugar será propagado a todos los lugares donde se utilice.**

Para que esto funcione, dicho código debe ser invocado cada vez que se requiera.

Es posible, esto es exactamente para lo que existen las funciones.

Se necesita **definirla**. Aquí, la palabra definir es significativa.

Así es como se ve la definición más simple de una función:

```
def function_name():  
    cuerpo de la función
```

- Siempre comienza con la **palabra reservada def** (que significa definir)
- después de **def** va el **nombre de la función** (las reglas para darle nombre a las funciones son las mismas que para las variables)
- después del nombre de la función, hay un espacio para un par de **paréntesis** (por ahora no contienen algo, pero eso cambiará pronto)
- la línea debe de terminar con **dos puntos**
- la línea inmediatamente después de **def** marca el comienzo del **cuerpo de la función** - donde varias o (al menos una) **instrucción anidada**, será ejecutada cada vez que la función sea invocada; nota: **la función termina donde el anidamiento termina**, se debe ser cauteloso.

A continuación se **definirá** la función. Se llamará message – aquí está:

```
1  def message():
2      print("Ingresar valor: ")
```

La función es muy sencilla, pero completamente utilizable. Se ha nombrado **message**, pero eso es opcional, tu puedes cambiarlo. Hagamos uso de ella.

El código ahora contiene la definición de la función:

```
1  def message():
2      print("Ingresar valor: ")
3
4  print("Inicia aqui.")
5  print("Termina aqui.")
6
```

Nota: no se está utilizando la función - no se está invocando en el código.

Al correr el programa, se mostrará lo siguiente:

```
Inicia aqui.
Termina aqui.
```

Se ha modificado el código - se ha insertado la invocación de la función entre los dos mensajes:

```
1  def message():
2      print("Ingresar valor: ")
3
4  print("Inicia aqui.")
5  message()
6  print("Termina aqui.")
```

```
Inicia aqui.
Ingresar valor:
Termina aqui.
```

Cómo funcionan las funciones



La imagen intenta mostrarte el proceso completo:

- cuando se **invoca** una función, Python recuerda el lugar donde esto ocurre y salta hacia dentro de la función invocada.
- el cuerpo de la función es entonces **ejecutado**.
- al llegar al final de la función, Python **regresa** al lugar inmediato después de donde ocurrió la invocación.

Existen dos consideraciones muy importantes, la primera de ella es:

- **No se debe invocar una función antes de que se haya definido.**

Recuerda - Python lee el código de arriba hacia abajo. No va a adelantarse en el código para determinar si la función invocada está definida más adelante, (el lugar "correcto" para definirla es "antes de ser invocada".)

Se ha insertado un error en el siguiente código - ¿puedes notar la diferencia?

```
1  print("Inicia aqui.")  
2  message()  
3  print("Termina aqui.")  
4  
5  
6  def message():  
7      print("Ingresar valor: ")
```

El mensaje de error dirá:

```
NameError: name 'message' is not defined
```

La segunda consideración es más sencilla:

- Una función y una variable no pueden compartir el mismo nombre.

El siguiente fragmento de código es erróneo

```
1  def message():
2      print("Ingresar valor: ")
3
4  message = 1
```

Regresando al ejemplo inicial para implementar la función de manera correcta:

```
1  def message():
2      print("Ingresar valor: ")
3
4  message()
5  a = int(input())
6  message()
7  b = int(input())
8  message()
9  c = int(input())
10
```

El modificar el mensaje de entrada es ahora sencillo - se puede hacer con solo modificar el código una única vez - dentro del cuerpo de la función.

Cómo las funciones se comunican con su entorno

Aprenderemos sobre funciones con y sin parámetros, así como también a escribir funciones de uno, dos y tres parámetros y pasarles argumentos.

Funciones parametrizadas

El potencial completo de una función se revela cuando puede ser equipada con una interface que es capaz de aceptar datos provenientes de la invocación. Dichos datos pueden modificar el comportamiento de la función, haciéndola más flexible y adaptable a condiciones cambiantes.

Un parámetro es una variable, pero existen dos factores que hacen a un parámetro diferente:

- **los parámetros solo existen dentro de las funciones en donde han sido definidos**, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra clave reservada **def**.
- **la asignación de un valor a un parámetro de una función se hace en el momento en que la función se manda llamar o se invoca**, especificando el argumento correspondiente.

```
def function(parameter):
    ###
```

Recuerda que:

- **los parámetros solo existen dentro de las funciones** (este es su entorno natural)
- **los argumentos existen fuera de las funciones**, y son los que pasan los valores a los parámetros correspondientes.

Existe una clara división entre estos dos mundos.

Enriquezcamos la función anterior agregándole un parámetro - se utilizará para mostrar al usuario el valor de un número que la función pide.

```
def message(number):
    ###
```

Esta definición especifica que nuestra función opera con un solo parámetro con el nombre de **number**. Se puede utilizar como una variable normal, pero **solo dentro de la función** - no es visible en otro lugar.

Ahora hay que mejorar el cuerpo de la función:

```
1 | def message(number):
2 |     print("Ingresa un número:", number)
3 |
```

Se ha hecho buen uso del parámetro. Nota: No se le ha asignado al parámetro algún valor. ¿Es correcto?

Si, lo es.

Un valor para el parámetro llegará del entorno de la función.

Recuerda: **especificar uno o más parámetros en la definición de la función es un requerimiento**, y se debe de cumplir durante la invocación de la misma. Se debe **proveer el mismo número de argumentos como haya parámetros definidos**.

El no hacerlo provocará un error.

Intenta ejecutar el código en el editor.

```
1 def message(number):  
2     print("Ingresa un número:", number)  
3  
4 message()  
5
```

Esto es lo que aparecerá en consola:

```
TypeError: message() missing 1 required positional argument: 'number'
```

Aquí está ya de manera correcta:

```
1 def message(number):  
2     print("Ingresa un número:", number)  
3  
4 message(1)  
5
```

El código producirá la siguiente salida:

```
Ingresa un número: 1
```

Existe una circunstancia importante que se debe mencionar.

Es posible tener una variable con el mismo nombre del parámetro de la función.

El siguiente código muestra un ejemplo de esto:

```
1 def message(number):  
2     print("Ingresa un número:", number)  
3  
4 number = 1234  
5 message(1)  
6 print(number)  
7
```

Una situación como la anterior activa un mecanismo denominado **sombreado**:

- el parámetro `x` sombrea cualquier variable con el mismo nombre, pero...
- ... solo dentro de la función que define el parámetro.

- El parámetro llamado `number` es una entidad completamente diferente de la variable llamada `number`.

Esto significa que el código anterior producirá la siguiente salida:

```
Ingresar un número: 1
1234
```

Una función puede tener **tantos parámetros como se desee**, pero entre más parámetros, es más difícil memorizar su rol y propósito.

Modifiquemos la función- ahora tiene dos parámetros:

```
def message(what, number):
    print("Ingresar", what, "número", number)
```

Esto significa que para invocar la función se necesitan **dos argumentos**.

El primer parámetro va a contener el nombre del valor deseado.

```
1  def message(what, number):
2      print("Ingresar", what, "número", number)
3
4  message("teléfono", 11)
5  message("precio", 5)
6  message("número", "number")
7
```

Esta es la salida del código anterior

```
Ingresar teléfono número 11
Ingresar precio número 5
Ingresar número número número
```

Paso de parámetros posicionales

La técnica que asigna cada argumento al parámetro correspondiente, es llamada **paso de parámetros posicionales**, los argumentos pasados de esta manera son llamados **argumentos posicionales**.


```

1  def my_function(a, b, c):
2      print(a, b, c)
3
4  my_function(1, 2, 3)
5

```

```

1  def introduction(first_name, last_name):
2      print("Hola, mi nombre es", first_name, last_name)
3
4  introduction("Luke", "Skywalker")
5  introduction("Jesse", "Quick")
6  introduction("Clark", "Kent")
7

```

Paso de argumentos de palabra clave

Python ofrece otra manera de pasar argumentos, donde el **significado del argumento está definido por su nombre**, no su posición - a esto se le denomina **paso de argumentos con palabra clave**.

Observa el siguiente código:

```

1  def introduction(first_name, last_name):
2      print("Hola, mi nombre es", first_name, last_name)
3
4  introduction(first_name = "James", last_name = "Bond")
5  introduction(last_name = "Skywalker", first_name = "Luke")
6

```

El concepto es claro - los valores pasados a los parámetros son precedidos por el nombre del parámetro al que se le va a pasar el valor, seguido por el signo de =.

La posición no es relevante aquí - cada argumento conoce su destino con base en el nombre utilizado.

Por supuesto **que no se debe de utilizar el nombre de un parámetro que no existe**.

El siguiente código provocará un error de ejecución:

```

1  def introduction(first_name, last_name):
2      print("Hola, mi nombre es", first_name, last_name)
3
4  introduction(surname="Skywalker", first_name="Luke")
5

```

```
TypeError: introduction() got an unexpected keyword argument 'surname'
```

Mezclando argumentos posicionales y de palabras clave

Es posible combinar ambos tipos si se desea - solo hay una regla inquebrantable: **se deben colocar primero los argumentos posicionales y después los de palabra clave.**

Para mostrarte como funciona, se utilizará la siguiente función de tres parámetros:

```
1 | def adding(a, b, c):  
2 |     print(a, "+", b, "+", c, "=", a + b + c)  
3 |
```

Su propósito es el de evaluar y presentar la suma de todos sus argumentos.

La función, al ser invocada de la siguiente manera (ejemplo solo de argumentos posicionales):

```
1 | adding(1, 2, 3)  
2 |
```

Dara la siguiente salida

```
1 + 2 + 3 = 6
```

También, se puede reemplazar la invocación actual por una con palabras clave, como la siguiente:

```
1 | adding(c = 1, a = 2, b = 3)  
2 |
```

El programa dará como salida lo siguiente:

```
2 + 3 + 1 = 6
```

Ahora intentemos mezclar ambas formas.

Observa la siguiente invocación de la función:

```
1 | adding(3, c = 1, b = 2)
2 |
```

Vamos a analizarla:

- el argumento (3) para el parámetro **a** es pasado utilizando la forma posicional;
- los argumentos para c y b son especificados con palabras clave.

Esto es lo que se verá en la consola:

```
3 + 2 + 1 = 6
```

Sé cuidadoso, ten cuidado de no cometer errores. Si se intenta pasar mas de un valor a un argumento, ocurrirá un error y se mostrará lo siguiente.

Observa la siguiente invocación - se le esta asignando dos veces un valor al parámetro a:

```
1 | adding(3, a = 1, b = 2)
2 |
```

Nos dará el siguiente error:

```
TypeError: adding() got multiple values for argument 'a'
```

Output

Funciones parametrizadas - más detalles

En ocasiones ocurre que algunos valores de ciertos argumentos son más utilizados que otros. Dichos argumentos tienen **valores predefinidos** los cuales pueden ser considerados cuando los argumentos correspondientes han sido omitidos.

Uno de los apellidos más comunes en Latinoamérica es González. Tomémoslo para el ejemplo.

El valor por predeterminada para el parámetro se asigna de la siguiente manera:

```
1 | def introduction(first_name, last_name="Smith"):
2 |     print("Hola, mi nombre es", first_name, last_name)
3 |
```

Solo se tiene que colocar el nombre del parámetro seguido del signo de = y el valor por predeterminada.

Retornando el resultado de una función

Aprenderás sobre los efectos y resultados de las funciones, la expresión **return** y el valor **None**. También aprenderás a pasar listas como argumentos de funciones, cómo devolver listas como resultados de funciones y cómo asignar resultados de funciones a variables.

Efectos y resultados: la instrucción return

Todas las funciones presentadas anteriormente tienen algún tipo de efecto - producen un texto y lo envían a la consola.

Por supuesto, las funciones - al igual que las funciones matemáticas - pueden tener resultados.

Para lograr que **las funciones devuelvan un valor** (pero no solo para ese propósito) se utiliza la **instrucción return** (regresar o retornar).

Esta palabra nos da una idea completa de sus capacidades. Nota: **es una palabra clave reservada** de Python.

La instrucción **return** tiene **dos variantes diferentes** - considerémoslas por separado.

return sin una expresión

Consideremos la siguiente función:

```
1  def happy_new_year(wishes = True):
2      print("Tres...")
3      print("Dos...")
4      print("Uno...")
5      if not wishes:
6          return
7
8      print("¡Feliz año nuevo!")
9
```

Cuando se invoca sin ningún argumento:

```
happy_new_year()
```

La función produce un poco de ruido - la salida se verá así:

```
Tres...
Dos...
Uno...
¡Feliz año nuevo!
```

Al proporcionar False como argumento:

```
happy_new_year(False)
```

Se modificará el comportamiento de la función - la instrucción return provocará su terminación justo antes de los deseos - esta es la salida actualizada:

```
Tres...
Dos...
Uno...
```

return con una expresión

La segunda variante de return está **extendida con una expresión**:

```
def function():
    return expression
```

Existen dos consecuencias de usarla:

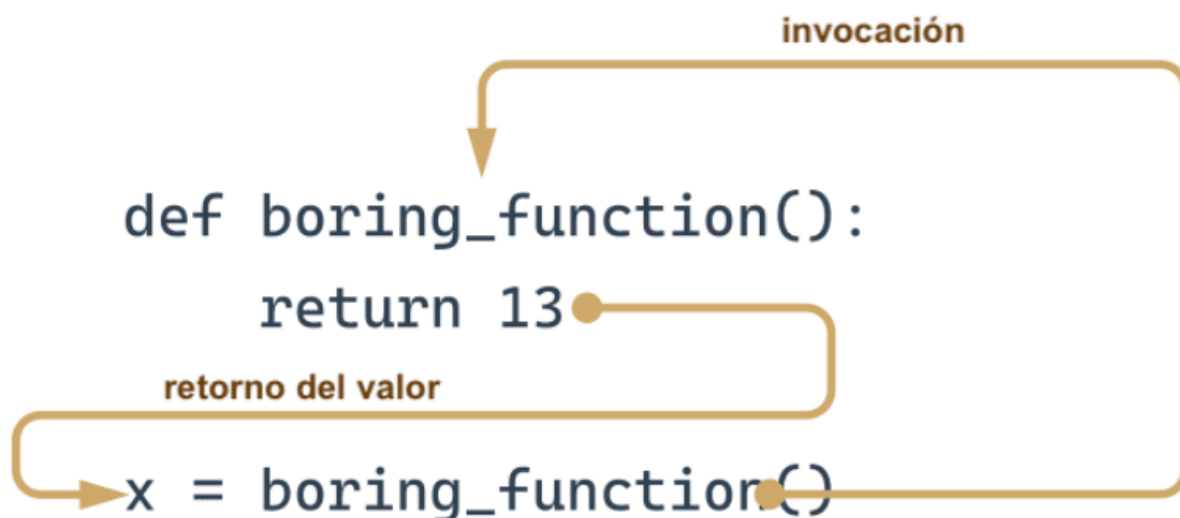
- provoca la **terminación inmediata de la ejecución de la función** (nada nuevo en comparación con la primer variante)
- además, la **función evaluará el valor de la expresión y lo devolverá** (de ahí el nombre una vez más) **como el resultado de la función**.

Si, este ejemplo es sencillo:

```
1  def boring_function():
2      return 123
3
4  x = boring_function()
5
6  print("La función boring_function ha devuelto su resultado. Es:", x)
7
```

El fragmento de código escribe el siguiente texto en la consola:

La función `boring_function` ha devuelto su resultado. Es: 123



La instrucción **return**, enriquecida con la expresión (la expresión es muy simple aquí), "transporta" el valor de la expresión al lugar donde se ha invocado la función.

El resultado se puede usar libremente aquí, por ejemplo, para ser asignado a una variable.

También puede ignorarse por completo y perderse sin dejar rastro.

Unas pocas palabras sobre None

Permítenos presentarte un valor muy curioso (para ser honestos, un valor que es ninguno) llamado **None**.

Sus datos no representan valor razonable alguno - en realidad, no es un valor en lo absoluto; por lo tanto, **no debe participar en ninguna expresión**.

Por ejemplo, un fragmento de código como el siguiente:

```
print(None + 2)
```

Causará un error de tiempo de ejecución, descrito por el siguiente mensaje de diagnóstico:

```
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Nota: **None** es una palabra clave reservada.

Solo existen dos tipos de circunstancias en las que **None** se puede usar de manera segura:

- cuando **se le asigna a una variable** (o se devuelve como el resultado de una función)
- cuando **se compara con una variable** para diagnosticar su estado interno.

Al igual que aquí:

```
1 value = None
2 if value is None:
3     print("Lo siento, no contiene ningún valor")
4
```

No olvides esto: si una función no devuelve un cierto valor utilizando una cláusula de expresión **return**, se asume que devuelve **implícitamente None**.

Vamos a probarlo.

Echa un vistazo al código en el editor.

```
1 def strange_function(n):
2     if(n % 2 == 0):
3         return True
4
```

Es obvio que la función `strange_Function` retorna `True` cuando su argumento es par.

Podemos usar el siguiente código para verificarlo:

```
1 print(strange_function(2))
2 print(strange_function(1))
3
```

Esto es lo que vemos en la consola:

```
True
None
```

Output

Efectos y resultados: listas y funciones

Existen dos preguntas adicionales que deben responderse aquí.

La primera es: **¿Se puede enviar una lista a una función como un argumento?**

¡Por supuesto que se puede! Cualquier entidad reconocible por Python puede desempeñar el papel de un argumento de función, aunque debes asegurarte de que la función sea capaz de hacer uso de él.

Entonces, si pasas una lista a una función, la función tiene que manejarla como una lista.

Una función como la siguiente:

```
1  def list_sum(lst):
2      s = 0
3
4      for elem in lst:
5          s += elem
6
7      return s
8
```

y se invoca así:

```
print(list_sum([5, 4, 3]))
```

retornará **12** como resultado, pero habrá problemas si la invocas de esta manera riesgosa:

```
print(list_sum(5))
```

La respuesta de Python será la siguiente:

```
TypeError: 'int' object is not iterable
```

Output

Esto se debe al hecho de que el **bucle for no puede iterar un solo valor entero**.

La segunda pregunta es: **¿Puede una lista ser el resultado de una función?**

¡Si, por supuesto! Cualquier entidad reconocible por Python puede ser un resultado de función.

Observa el código en el editor. La salida del programa será así:

```

1 def strange_list_fun(n):
2     strange_list = []
3
4     for i in range(0, n):
5         strange_list.insert(0, i)
6
7     return strange_list
8
9 print(strange_list_fun(5))
10

```

Observa el código en el editor. La salida del programa será así:

```
[4, 3, 2, 1, 0]
```

Output

Los alcances en Python

Aprenderás sobre los alcances en Python y la palabra clave global. Al final de la sección, podrás distinguir entre variables locales y globales, y sabrás cómo utilizar el mecanismo de espacios de nombres (namespaces) en tus programas.

Funciones y alcances

Comencemos con una definición:

El **alcance de un nombre** (por ejemplo, el nombre de una variable) es la parte del código donde el nombre es reconocido correctamente.

Por ejemplo, el alcance del parámetro de una función es la función en sí. El parámetro es inaccesible fuera de la función.

Vamos a revisarlo. Observa el código en el editor. ¿Qué ocurrirá cuando se ejecute?

```

1 def scope_test():
2     x = 123
3
4
5 scope_test()
6 print(x)
7

```

El programa no correrá. El mensaje de error dirá:

```
NameError: name 'x' is not defined
```

Output

Esto era de esperarse.

Vamos a conducir algunos experimentos para mostrar como es que Python define los alcances, y como los puedes utilizar para tu beneficio.

Comencemos revisando si una variable creada fuera de una función es visible dentro de una función. En otras palabras, ¿El nombre de la variable se propaga dentro del cuerpo de la función?

Observa el código en el editor.

```
1 def my_function():
2     print("¿Conozco a la variable?", var)
3
4
5 var = 1
6 my_function()
7 print(var)
8
```

El resultado de la prueba es positivo - el código da como salida:

```
¿Conozco a la variable? 1
1
```

Output

La respuesta es: **una variable que existe fuera de una función tiene alcance dentro del cuerpo de la función.**

Esta regla tiene una excepción muy importante. Intentemos encontrarla.

Hagamos un pequeño cambio al código:

```
1 def my_function():
2     var = 2
3     print("¿Conozco a la variable?", var)
4
5
6 var = 1
7 my_function()
8 print(var)
9
```

El resultado ha cambiado también - el código arroja una salida con una ligera diferencia:

```
¿Conozco a la variable? 2
1
```

Output

¿Qué es lo que ocurrió?

- la variable **var** creada dentro de la función no es la misma que cuando se define fuera de ella, parece que hay dos variables diferentes con el mismo nombre.

- además, la variable de la función es una sombra de la variable fuera de la función.

La regla anterior se puede definir de una manera más precisa y adecuada:

Una variable que existe fuera de una función tiene un alcance dentro del cuerpo de la función, excluyendo a aquellas que tienen el mismo nombre.

También significa que **el alcance de una variable existente fuera de una función solo se puede implementar dentro de una función cuando su valor es leído**. El asignar un valor hace que la función cree su propia variable.

Funciones y alcances: la palabra clave global

Al llegar a este punto, debemos hacernos la siguiente pregunta: ¿Una función es capaz de modificar una variable que fue definida fuera de ella? Esto sería muy incómodo.

Afortunadamente, la respuesta es no.

Existe un método especial en Python **el cual puede extender el alcance de una variable incluyendo el cuerpo de las funciones** (para poder no solo leer los valores de las variables sino también modificarlos).

Este efecto es causado por la **palabra clave reservada llamada global**:

```
1 | global name
2 | global name1, name2, ...
3 |
```

El utilizar la palabra reservada dentro de una función con el nombre o nombres de las variables separados por comas, obliga a Python a abstenerse de crear una nueva variable dentro de la función - se empleará la que se puede acceder desde el exterior.

En otras palabras, este nombre se convierte en global (tiene un **alcance global**, y no importa si se esta leyendo o asignando un valor).

Observa el código en el editor.

```

1 def my_function():
2     global var
3     var = 2
4     print("¿Conozco a aquella variable?", var)
5
6
7 var = 1
8 my_function()
9 print(var)
10

```

Console >_

```

¿Conozco a aquella variable? 2
2

```

Cómo interactúa la función con sus argumentos

Ahora descubramos como la función interactúa con sus argumentos.

El código en el editor nos enseña algo. Como puedes observar, la función cambia el valor de su parámetro. ¿Este cambio afecta el argumento?

```

1 def my_function(n):
2     print("Yo recibí", n)
3     n += 1
4     print("Ahora tengo", n)
5
6
7 var = 1
8 my_function(var)
9 print(var)
10

```

Console >_

```

Yo recibí 1
Ahora tengo 2
1

```

La conclusión es obvia - al cambiar el valor del parámetro este no se propaga fuera de la función (más específicamente, no cuando la variable es un valor escalar, como en el ejemplo).

Esto también significa que una función recibe el valor del argumento, no el argumento en sí. Esto es cierto para los valores escalares.

Creación de funciones multiparámetro

Analizaremos los siguientes ejemplos de funciones con multiples parámetros: Calculadora de IMC, Convertidor de Unidades, Probador de Triángulos, Calculadora de Área de Triángulos, Factorial, Fibonacci, y Funciones Recursivas.


Ejemplos de funciones: Cálculo del IMC

Definamos una función que calcula el Índice de Masa Corporal (IMC).

Como puedes observar, la formula ocupa dos valores:

- **peso** (originalmente en kilogramos)
- **altura** (originalmente en metros)

La nueva función tendrá dos parámetros. Su nombre será bmi.

$$\text{IMC} = \frac{\text{peso en kilogramos}}{\text{altura en metros}^2}$$


Codifiquemos la función.

```
1 def bmi(weight, height):
2     return weight / height ** 2
3
4
5 print(bmi(52.5, 1.65))
6
```

El resultado del ejemplo anterior es el siguiente:

19.283746556473833

Output

La función hace lo que deseamos, pero es un poco sencilla - asume que los valores de ambos parámetros son significativos. Se debe comprobar que son confiables.

Vamos a comprobar ambos y regresar None si cualquiera de los dos es incorrecto.

Calcular el IMC y convertir unidades del Sistema Inglés al Sistema Métrico

Observa el código en el editor. Hay dos cosas a las cuales hay que prestar atención.

```

1 def bmi(weight, height):
2     if height < 1.0 or height > 2.5 or \
3     weight < 20 or weight > 200:
4         return None
5
6     return weight / height ** 2
7
8
9 print(bmi(352.5, 1.65))
10

```

Primero, se asegura que los datos que sean ingresados sean correctos - de lo contrario la salida será:

None

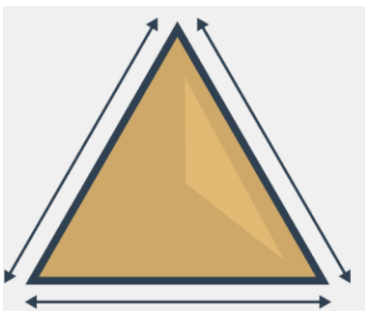
Output

Segundo, observa como el símbolo de diagonal invertida (\) es empleado. Si se termina una línea de código con el, Python entenderá que la línea continua en la siguiente.

Esto puede ser útil cuando se tienen largas líneas de código y se desea que sean más legibles.

Ejemplos de funciones: Triángulos

Ahora trabajaremos con triángulos. Comenzaremos con una función que verifique si tres lados de ciertas longitudes pueden formar un triángulo.



En la escuela aprendimos que la suma arbitraria de dos lados tiene que ser mayor que la longitud del tercer lado.

No será algo difícil. La función tendrá tres parámetros - uno para cada lado.

Regresará True si todos los lados pueden formar un triángulo, y False de lo contrario. En este caso, **is_a_triangle** es un buen nombre para dicha función.

Observa el código en el editor. Ahí se encuentra la función. Ejecuta el programa.


```

1 def is_a_triangle(a, b, c):
2     if a + b <= c:
3         return False
4     if b + c <= a:
5         return False
6     if c + a <= b:
7         return False
8     return True
9
10
11 print(is_a_triangle(1, 1, 1))
12 print(is_a_triangle(1, 1, 3))
13

```

Console >_

```

True
False

```

¿Se podrá hacer más compacta? Parece tener demasiadas palabras.

```

1 def is_a_triangle(a, b, c):
2     if a + b <= c or b + c <= a or c + a <= b:
3         return False
4     return True
5
6
7 print(is_a_triangle(1, 1, 1))
8 print(is_a_triangle(1, 1, 3))
9

```

¿Se puede compactar aun más?

```

1 def is_a_triangle(a, b, c):
2     return a + b > c and b + c > a and c + a > b
3
4
5 print(is_a_triangle(1, 1, 1))
6 print(is_a_triangle(1, 1, 3))
7

```

Se ha negado la condición (se invirtieron los operadores relacionales y se reemplazaron los ors con ands, obteniendo una expresión universal para probar triángulos).

Triángulos y el Teorema de Pitágoras

Observa el código en el editor. Le pide al usuario tres valores. Después hace uso de la función `is_a_triangle`.

```

1 def is_a_triangle(a, b, c):
2     return a + b > c and b + c > a and c + a > b
3
4
5 a = float(input('Ingresa la longitud del primer lado: '))
6 b = float(input('Ingresa la longitud del segundo lado: '))
7 c = float(input('Ingresa la longitud del tercer lado: '))
8
9 if is_a_triangle(a, b, c):
10     print('Si, si puede ser un triángulo.')
11 else:
12     print('No, no puede ser un triángulo.')
13
14

```

En el segundo paso, intentaremos verificar si un triángulo es un **triángulo rectángulo**.

Para ello haremos **uso del Teorema de Pitágoras**:

$$c^2 = a^2 + b^2$$

¿Cómo saber cual de los tres lados es la hipotenusa?

La hipotenusa es el lado más largo.

Aquí esta el código:

```

1 def is_a_triangle(a, b, c):
2     return a + b > c and b + c > a and c + a > b
3
4
5 def is_a_right_triangle(a, b, c):
6     if not is_a_triangle(a, b, c):
7         return False
8     if c > a and c > b:
9         return c ** 2 == a ** 2 + b ** 2 if a > b and a > c:
10     if a > b and a > c:
11         return a ** 2 == b ** 2 + c ** 2
12 print(is_a_right_triangle(5, 3, 4))
13 print(is_a_right_triangle(1, 3, 4))
14

```

Ejemplos de funciones: Factoriales

La siguiente función a definir calcula factoriales. ¿Recuerdas cómo se calcula un factorial?

$0! = 1$ (¡Sí!, es verdad) $1! = 1$ $2! = 1 * 2$ $3! = 1 * 2 * 3$ $4! = 1 * 2 * 3 * 4$: : $n! = 1 * 2 * 3 * 4 * \dots * n-1 * n$

Se expresa con un signo de exclamación, y es igual al producto de todos los números naturales previos al argumento o número dado.

Escribamos el código. Creemos una función con el nombre **factorial_function**. Aquí está el código:

```
1  def factorial_function(n):
2      if n < 0:
3          return None
4      if n < 2:
5          return 1
6
7      product = 1
8      for i in range(2, n + 1):
9          product *= i
10     return product
11
12
13     for n in range(1, 6): # probando
14         print(n, factorial_function(n))
15
```

Observa como se sigue el procedimiento matemático, y como se emplea el bucle **for** para encontrar el producto.

Estos son los resultados obtenidos del código de prueba:

```
1 1
2 2
3 6
4 24
5 120
```

Output

Números Fibonacci

¿Estás familiarizado con la serie Fibonacci?

Son una secuencia de números enteros los cuales siguen una regla sencilla:

- el primer elemento de la secuencia es igual a uno (Fib1 = 1)
- el segundo elemento también es igual a uno (Fib2 = 1)
- cada número después de ellos son la suman de los dos números anteriores (Fibi = Fibi-1 + Fibi-2)

Aquí están algunos de los primeros números en la serie Fibonacci:

```
fib_1 = 1 fib_2 = 1 fib_3 = 1 + 1 = 2 fib_4 = 1 + 2 = 3 fib_5 = 2 + 3 = 5 fib_6 = 3 + 5 = 8 fib_7 = 5 + 8 = 13
```

¿Que opinas acerca de implementarlo como una función?

Creemos nuestra propia función fib y probémosla, aquí esta:

```
1  def fib(n):
2      if n < 1:
3          return None
4      if n < 3:
5          return 1
6
7      elem_1 = elem_2 = 1
8      the_sum = 0
9      for i in range(3, n + 1):
10         the_sum = elem_1 + elem_2
11         elem_1, elem_2 = elem_2, the_sum
12     return the_sum
13
14
15 for n in range(1, 10): # probando
16     print(n, "->", fib(n))
17
```

Analiza el código del bucle **for** cuidadosamente, descifra como se mueven las variables `elem_1` y `elem_2` a través de los números subsecuentes de la serie Fibonacci.

Al probar el código, se generan los siguientes resultados:

```
1 -> 1
2 -> 1
3 -> 2
4 -> 3
5 -> 5
6 -> 8
7 -> 13
8 -> 21
9 -> 34
```

Output

Recursividad

Existe algo más que se desea mostrar: es la **recursividad**.

Este termino puede describir muchos conceptos distintos, pero uno de ellos, hace referencia a la programación computacional.

Aquí, la recursividad es **una técnica donde una función se invoca a si misma**.

Tanto el factorial como la serie Fibonacci, son las mejores opciones para ilustrar este fenómeno.

La serie de Fibonacci es un claro ejemplo de recursividad. Ya te dijimos que:

$Fibi = Fibi-1 + Fibi-2$

El número i se refiere al número i-1, y así sucesivamente hasta llegar a los primeros dos.

¿Puede ser empleado en el código? Por supuesto que puede. Puede hacer el código más corto y claro.

La segunda versión de la función fib() hace uso directo de la recursividad:

```
1  def fib(n):
2      if n < 1:
3          return None
4      if n < 3:
5          return 1
6      return fib(n - 1) + fib(n - 2)
7
```

El código es mucho más claro ahora.

¿Pero es realmente seguro? ¿Implica algún riesgo?

Si, existe algo de riesgo. **Si no se considera una condición que detenga las invocaciones recursivas, el programa puede entrar en un bucle infinito**. Se debe ser cuidadoso.

Tuplas y diccionarios

Aprenderás sobre los tipos de secuencias y el concepto de mutabilidad. Aprenderás sobre las tuplas y los diccionarios, y cómo puedes usarlos para almacenar y procesar valores de datos.

Tipos de secuencia y mutabilidad

Antes de comenzar a hablar acerca de **tuplas y diccionarios**, se deben introducir dos conceptos importantes: **tipos de secuencia y mutabilidad**.

Un **tipo de secuencia** es un tipo de dato en Python el cual es capaz de almacenar más de un valor (o ninguno si la secuencia esta vacía), los cuales pueden ser secuencialmente (de ahí el nombre) **examinados**, elemento por elemento.

Debido a que el bucle **for** es una herramienta especialmente diseñada para iterar a través de las secuencias, podemos definirlas de la siguiente manera: **una secuencia es un tipo de dato que puede ser escaneado por el bucle for**.

Hasta ahora, has trabajado con una secuencia en Python - la lista. La lista es un clásico ejemplo de una secuencia de Python. Aunque existen otras secuencias dignas de mencionar, las cuales se presentaran a continuación.

La segunda noción - la **mutabilidad** - es una propiedad de cualquier tipo de dato en Python que describe su disponibilidad para poder cambiar libremente durante la ejecución de un programa. Existen dos tipos de datos en Python: **mutables** e **inmutables**.

Los datos mutables pueden ser actualizados libremente en cualquier momento - a esta operación se le denomina "in situ".

In situ es una expresión en Latín que se traduce literalmente como en posición, en el lugar o momento. Por ejemplo, la siguiente instrucción modifica los datos "in situ":

```
list.append(1)
```

Los datos inmutables no pueden ser modificados de esta manera.

Imagina que una lista solo puede ser asignada y leída. No podrías adjuntar ni remover un elemento de la lista. Si se agrega un elemento al final de la lista provocaría que la lista se cree desde cero.

Se tendría que crear una lista completamente nueva, la cual contenga los elementos ya existentes más el nuevo elemento.

El tipo de datos que se desea tratar ahora se llama **tupla**. **Una tupla es una secuencia inmutable**. Se puede comportar como una lista pero no puede ser modificada en el momento.

Tuplas

Lo primero que distingue una lista de una tupla es la sintaxis empleada para crearlas - **las tuplas utilizan paréntesis**, mientras que las listas usan corchetes, aunque **también es posible crear una tupla tan solo separando los valores por comas**.

Observa el ejemplo:

```
1 | tuple_1 = (1, 2, 4, 8)
2 | tuple_2 = 1., .5, .25, .125
3 |
```

Se definieron dos tuplas, ambas contienen **cuatro elementos**.

A continuación se imprimen en consola:

```
1 | tuple_1 = (1, 2, 4, 8)
2 | tuple_2 = 1., .5, .25, .125
3 |
4 | print(tuple_1)
5 | print(tuple_2)
6 |
```

Output

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

Nota: **cada elemento de una tupla puede ser de distinto tipo** (punto flotante, entero, cadena, o cualquier otro tipo de dato).

¿Cómo crear una tupla?

¿Es posible crear una tupla vacía? Si, solo se necesitan unos paréntesis:

```
1 | empty_tuple = ()
2 |
```

Si se desea **crear una tupla de un solo elemento**, se debe de considerar el hecho de que, debido a la sintaxis (una tupla debe de poder distinguirse de un valor entero ordinario), se debe de colocar una coma al final:

```
one_element_tuple_1 = (1, )
one_element_tuple_2 = 1.,
```

El quitar las comas no arruinará el programa en el sentido sintáctico, pero serán dos variables, no tuplas.

¿Cómo utilizar un tupla?

Si deseas leer los elementos de una tupla, lo puedes hacer de la misma manera que se hace con las listas.

Observa el código en el editor.

```
1 my_tuple = (1, 10, 100, 1000)
2
3 print(my_tuple[0])
4 print(my_tuple[-1])
5 print(my_tuple[1:])
6 print(my_tuple[:-2])
7
8 for elem in my_tuple:
9     print(elem)
10
```

El programa debe de generar la siguiente salida, ejecútalo y comprueba:

```
1
1000
(10, 100, 1000)
(1, 10)
1
10
100
1000
```

Output

Las similitudes pueden ser engañosas - **no intentes modificar el contenido de la tupla** ¡No es una lista!

Todas estas instrucciones (con excepción de primera) causarán un error de ejecución:

```
1 my_tuple = (1, 10, 100, 1000)
2
3 my_tuple.append(10000)
4 del my_tuple[0]
5 my_tuple[1] = -10
6
```

```
AttributeError: 'tuple' object has no attribute 'append'
```

Output

¿Qué más pueden hacer las tuplas?

- la función **len()** acepta tuplas, y regresa el número de elementos contenidos dentro;
- el operador **+** puede unir tuplas (ya se ha mostrado esto antes)
- el operador ***** puede multiplicar las tuplas, así como las listas;
- los operadores **in** y **not in** funcionan de la misma manera que en las listas.

El fragmento de código en el editor presenta todo esto.

```

1 my_tuple = (1, 10, 100)
2
3 t1 = my_tuple + (1000, 10000)
4 t2 = my_tuple * 3
5
6 print(len(t2))
7 print(t1)
8 print(t2)
9 print(10 in my_tuple)
10 print(-10 not in my_tuple)
11

```

Output

```

9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True

```

Diccionarios

El **diccionario** es otro tipo de estructura de datos de Python. No es una secuencia (pero puede adaptarse fácilmente a un procesamiento secuencial) y además es mutable.

¿Cómo crear un diccionario?

Si deseas asignar algunos pares iniciales a un diccionario, utiliza la siguiente sintaxis:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2 phone_numbers = {'jefe': 5551234567, 'Suzy': 22657854310}
3 empty_dictionary = {}
4
5 print(dictionary)
6 print(phone_numbers)
7 print(empty_dictionary)
8

```

En este primer ejemplo, el diccionario emplea claves y valores las cuales ambas son cadenas. En el segundo, las claves son cadenas pero los valores son enteros. El orden inverso (claves → números, valores → cadenas) también es posible, así como la combinación número a número.

La lista de todos los pares es encerrada con llaves, mientras que los pares son separados por comas, y las claves y valores por dos puntos.

El primer diccionario es muy simple, es un diccionario Español-Francés. El segundo es un directorio telefónico muy pequeño.

Los diccionarios vacíos son construidos por un **par vacío de llaves**.

Un diccionario en Python funciona de la misma manera que un diccionario bilingüe. Por ejemplo, se tiene la palabra en español "gato" y se necesita su equivalente en francés. Lo que se haría es buscar en el diccionario para encontrar la palabra "gato". Eventualmente la encontrarás, y sabrás que la palabra equivalente en francés es "chat".

En el mundo de Python, la palabra que se esta buscando se denomina **key**. La palabra que se obtiene del diccionario es denominada **value**.

Esto significa que un diccionario es un conjunto de pares de key y value.

Nota:

- cada clave debe de ser **única** - no es posible tener una clave duplicada;
- una clave puede ser **un de dato de cualquier tipo** - puede ser un número (entero o flotante), incluso una cadena, pero no una lista.
- un diccionario no es una lista - una lista contiene un conjunto de valores numerados, mientras que **un diccionario almacena pares de valores**.
- la función **len()** aplica también para los diccionarios - regresa la cantidad de pares (clave-valor) en el diccionario.
- un diccionario **es una herramienta de un solo sentido** - si fuese un diccionario español-francés, podríamos buscar en español para encontrar su contraparte en francés más no viceversa.

A continuación veamos algunos ejemplos:

El diccionario entero se puede imprimir con una invocación a la función `print()`. El fragmento de código puede producir la siguiente salida:

```
{'perro': 'chien', 'caballo': 'cheval', 'gato': 'chat'}  
{ 'Suzy': 5557654321, 'jefe': 5551234567}  
{}
```

¿Has notado que el orden de los pares impresos es diferente a la asignación inicial? ¿Qué significa esto?

Primeramente, recordemos que **los diccionarios no son listas** - no guardan el orden de sus datos, el orden no tiene significado (a diferencia de los diccionarios reales). El orden en que un diccionario almacena sus datos esta fuera de nuestro control. Esto es normal. (*)

Nota

(*) En Python 3.6x los diccionarios se han convertido en colecciones ordenadas de manera predeterminada. Tu resultado puede variar dependiendo en la versión de Python que se este utilizando.

¿Cómo utilizar un diccionario?

Analiza el código en el editor:

```
1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2 phone_numbers = {'jefe' : 5551234567, 'Suzy' : 22657854310}
3 empty_dictionary = {}
4
5 # Imprimir valores aquí.
6 print(dictionary['gato'])
7 print(phone_numbers['Suzy'])
8
```

Console >_

chat

22657854310

El obtener el valor de un diccionario es semejante a la indexación, gracias a los corchetes alrededor del valor de la clave.

Nota:

- si una clave es una cadena, se tiene que especificar como una cadena;
- **las claves son sensibles a las mayúsculas y minúsculas:** 'Suzy' sería diferente a 'suzy'.

Ahora algo muy importante: **No se puede utilizar una clave que no exista.** Hacer algo como lo siguiente:

```
1 | print(phone_numbers['president'])
2 |
```

Provocará un error de ejecución. Inténtalo.

Afortunadamente, existe una manera simple de evitar dicha situación. El operador **in**, junto con su acompañante, **not in**, pueden salvarnos de esta situación.

El siguiente código busca de manera segura palabras en francés:

```

1 dictionary = {"cat": "gato", "perro": "chien", "caballo": "cheval"}
2 words = ['gato', 'león', 'caballo']
3
4 for word in words:
5     if word in dictionary:
6         print(word, "->", dictionary[word])
7     else:
8         print(word, "no está en el diccionario")
9

```

La salida del código es la siguiente:

Output

```

gato -> chat
león no está en el diccionario
caballo -> cheval

```

Nota

Cuando escribes una expresión grande o larga, puede ser una buena idea mantenerla alineada verticalmente. Así es como puede hacer que el código sea más legible y más amigable para el programador, por ejemplo:

```

1 # Ejemplo 1:
2 dictionary = {
3     "gato": "chat",
4     "perro": "chien",
5     "caballo": "cheval"
6 }
7 # Ejemplo 2:
8 phone_numbers = {'jefe': 5551234567,
9                  'Suzy': 22657854310
10 }
11

```

Métodos y funciones de los diccionarios

El método keys()

¿Pueden los diccionarios ser **examinados** utilizando el bucle for, como las listas o tuplas?

No y si.

No, porque **un diccionario no es un tipo de dato secuencial** - el bucle **for** no es útil aquí.

Si, porque hay herramientas simples y muy efectivas **que pueden adaptar cualquier diccionario a los requerimientos del bucle for** (en otras palabras, se construye un enlace intermedio entre el diccionario y una entidad secuencial temporal).

El primero de ellos es un método denominado **keys()**, el cual es parte de todo diccionario. El método **retorna o regresa una lista de todas las claves dentro del diccionario**. Al tener una lista de claves se puede acceder a todo el diccionario de una manera fácil y útil.

A continuación se muestra un ejemplo:

```
1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 for key in dictionary.keys():
4     print(key, "->", dictionary[key])
5
```

Otra manera de hacerlo es utilizar el **método items()**. Este **método retorna una lista de tuplas** (este es el primer ejemplo en el que las tuplas son mas que un ejemplo de si mismas) **donde cada tupla es un par de cada clave con su valor**.

Así es como funciona:

```
1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 for spanish, french in dictionary.items():
4     print(spanish, "->", french)
5
```

Nota la manera en que la tupla ha sido utilizada como una variable del bucle for.

Modificar, agregar y eliminar valores

El asignar un nuevo valor a una clave existente es sencillo - debido a que los diccionarios son completamente mutables, no existen obstáculos para modificarlos.

Se va a reemplazar el valor "chat" por "minou", lo cual no es muy adecuado, pero funcionará con nuestro ejemplo.

Observa:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 dictionary['gato'] = 'minou'
4 print(dictionary)
5

```

El código produce la siguiente salida:

```

caballo -> cheval
perro -> chien
gato -> chat

```

Output

La salida es:

```
{'gato': 'minou', 'perro': 'chien', 'caballo': 'cheval'}
```

Output

La función sorted()

¿Deseas que la salida este ordenada? Solo hay que agregar al bucle for lo siguiente:

```
for key in sorted(dictionary.keys()):
```

También existe un método denominado **values()**, funciona de manera muy similar al de **keys()**, pero regresa una lista de valores.

Este es un ejemplo sencillo:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 for french in dictionary.values():
4     print(french)
5

```

Como el diccionario no es capaz de automáticamente encontrar la clave de un valor dado, el rol de este método es algo limitado.

Agregando nuevas claves

El agregar una nueva clave con su valor a un diccionario es tan simple como cambiar un valor. Solo se tiene que asignar **un valor a una nueva clave que no haya existido antes**.

Nota: este es un comportamiento muy diferente comparado a las listas, las cuales no permiten asignar valores a índices no existentes.

A continuación se agrega un par nuevo al diccionario - un poco extraño pero válido:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 dictionary['cisne'] = 'cygne'
4 print(dictionary)
5

```

El ejemplo muestra como salida:

Output

```
{'gato': 'chat', 'perro': 'chien', 'caballo': 'cheval', 'cisne': 'cygne'}
```

Nota

También es posible insertar un elemento al diccionario utilizando el método **update()**, por ejemplo:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 dictionary.update({"pato": "canard"})
4 print(dictionary)
5

```

Eliminado una clave

¿Puedes deducir como eliminar una clave de un diccionario?

Nota: **al eliminar la clave también se removerá el valor asociado. Los valores no pueden existir sin sus claves.**

Esto se logra con la **instrucción del**.

A continuación un ejemplo:

```

1 dictionary = {"gato": "chat", "perro": "chien", "caballo": "cheval"}
2
3 del dictionary['perro']
4 print(dictionary)
5

```

Nota: **el eliminar una clave no existente, provocará un error.**

El ejemplo da como salida:

Output

```
{'gato': 'chat', 'caballo': 'cheval'}
```


Las tuplas y los diccionarios pueden trabajar juntos

Se ha preparado un ejemplo sencillo, mostrando como las tuplas y los diccionarios pueden trabajar juntos.

Imaginemos el siguiente problema:

- necesitas un programa para calcular los promedios de tus alumnos.
- el programa pide el nombre del alumno seguido de su calificación.
- los nombres son ingresados en cualquier orden.
- el ingresar un nombre vacío finaliza el ingreso de los datos (Nota 1: ingresar una puntuación vacía generará la excepción ValueError, pero no te preocupes por eso ahora, verás cómo manejar tales casos cuando hablemos de excepciones).
- una lista con todos los nombre y el promedio de cada alumno debe ser mostrada al final.

Observa el código en el editor, se muestra la solución.

```
1 school_class = {}
2
3 while True:
4     name = input("Ingresa el nombre del estudiante: ")
5     if name == '':
6         break
7
8     score = int(input("Ingresa la calificación del estudiante (0-10): "))
9     if score not in range(0, 11):
10         break
11
12     if name in school_class:
13         school_class[name] += (score,)
14     else:
15         school_class[name] = (score,)
16
17 for name in sorted(school_class.keys()):
18     adding = 0
19     counter = 0
20     for score in school_class[name]:
21         adding += score
22         counter += 1
23     print(name, ":", adding / counter)
24
```

Ahora se analizará línea por línea:

- línea 1: crea un diccionario vacío para ingresar los datos; el nombre del alumno es empleado como clave, mientras que todas las calificaciones asociadas son almacenadas en una tupla (la tupla puede ser el valor de un diccionario, esto no es un problema)
- línea 3: se ingresa a un bucle "infinito" (no te preocupes, saldremos de el en el momento indicado)

- línea 4: se lee el nombre del alumno aquí;
- línea 5-6: si el nombre es una cadena vacía (), salimos del bucle;
- línea 8: se pide la calificación del estudiante (un valor entero en el rango del 0-10)
- línea 9-10: si la puntuación ingresada no está dentro del rango de 0 a 10, se abandona el bucle;
- línea 12-13: si el nombre del estudiante ya se encuentra en el diccionario, se alarga la tupla asociada con la nueva calificación (observa el operador +=)
- línea 14-15: si el estudiante es nuevo (desconocido para el diccionario), se crea una entrada nueva, su valor es una tupla de un solo elemento la cual contiene la calificación ingresada;
- línea 17: se itera a través de los nombres ordenados de los estudiantes;
- línea 18-19: inicializa los datos necesarios para calcular el promedio (sum y counter).
- línea 20-22: se itera a través de la tupla, tomando todas las calificaciones subsecuentes y actualizando la suma junto con el contador.
- línea 23: se calcula e imprime el promedio del alumno junto con su nombre.

Este es un ejemplo del programa:

```
Ingresar el nombre del estudiante: Bob
Ingresar la calificación del estudiante (0-10): 7
Ingresar el nombre del estudiante: Andy
Ingresar la calificación del estudiante (0-10): 3
Ingresar el nombre del estudiante: Bob
Ingresar la calificación del estudiante (0-10): 2
Ingresar el nombre del estudiante: Andy
Ingresar la calificación del estudiante (0-10): 10
Ingresar el nombre del estudiante: Andy
Ingresar la calificación del estudiante (0-10): 3
Ingresar el nombre del estudiante: Bob
Ingresar la calificación del estudiante (0-10): 9
Ingresar el nombre del estudiante:
Andy : 5.333333333333333
Bob : 6.0
```

Excepciones

Aprenderás sobre el mecanismo de manejo de excepciones en Python. Explorarás los temas de errores en el código y aprenderás qué puedes hacer para evitar la finalización de tus programas. También se cubrirá el tema de las pruebas y depuración de código, y aprenderás algunos consejos sobre cómo hacer que tu proceso de escritura de código sea mejor y menos propenso a errores.

Errores - el pan diario del desarrollador

Parece indiscutible que todos los programadores (incluso tú) quieren escribir código libre de errores y hacen todo lo posible para lograr este objetivo. Desafortunadamente, nada es perfecto en este mundo y el software no es una excepción. Presta atención a la palabra **excepción**, ya que la veremos de nuevo muy pronto en un significado que no tiene nada que ver con el común.

Errar es humano, es imposible no cometer errores y es imposible escribir código sin errores. No nos malinterpretes, no queremos convencerte de que escribir programas desordenados y defectuosos es una virtud. Más bien queremos explicar que incluso el programador más cuidadoso no puede evitar defectos menores o mayores. **Sólo aquellos que no hacen nada no cometen errores.**



Paradójicamente, aceptar esta difícil verdad puede convertirte en un mejor programador y mejorar la calidad de tu código.

Te preguntarás: "¿Cómo podría ser esto posible?".

Errores en los datos frente a errores en el código

El lidiar con errores de programación tiene (al menos) dos partes. La primera es cuando te metes en problemas porque tu código, aparentemente correcto, se alimenta con datos incorrectos. Por ejemplo, esperas que se ingrese al código un valor entero, pero tu usuario descuidado ingresa algunas letras al azar.

Puede suceder que tu código termine en ese momento y el usuario se quede solo con un mensaje de error conciso y a la vez ambiguo en la pantalla. El usuario estará insatisfecho y tu también deberías estarlo.

Te mostraremos cómo proteger tu código de este tipo de fallas y cómo no provocar la ira del usuario.

La segunda parte de lidiar con errores de programación se revela cuando ocurre un comportamiento no deseado del programa debido a errores que se cometieron cuando se estaba escribiendo el código. Este tipo de error se denomina comúnmente "**bug**".

Cuando los datos no son lo que deberían ser

Escribamos un fragmento de código extremadamente trivial - leerá un número natural (un entero no negativo) e imprimirá su recíproco. De esta forma, 2 se convertirá en 0.5 (1/2) y 4 en 0.25 (1/4). Aquí está el programa:

```
1 value = int(input('Ingresa un número natural: '))
2 print('El recíproco de', value, 'es', 1/value)
3 |
```

Console >_

```
Ingresa un número natural: 4
El recíproco de 4 es 0.25
```

¿Hay algo que pueda salir mal? El código es tan breve y compacto que no parece que vayamos a encontrar ningún problema allí.

Parece que ya sabes hacia dónde vamos. Sí, tienes razón - ingresar datos que no sean un número entero (que también incluye ingresar nada) arruinará completamente la ejecución del programa. Esto es lo que verá el usuario del código:

```
Traceback (most recent call last):
  File "code.py", line 1, in
    value = int(input('Ingresa un número natural: '))
ValueError: invalid literal for int() with base 10: ''
```

Output

Todas las líneas que muestra Python son significativas e importantes, pero la última línea parece ser la más valiosa. La primera palabra de la línea es el nombre de la excepción la cual provoca que tu código se detenga. Su nombre aquí es `ValueError`. El resto de la línea es solo una breve explicación que especifica con mayor precisión la causa de la excepción ocurrida.

¿Cómo lo afrontas? ¿Cómo proteges tu código de la terminación abrupta, al usuario de la decepción y a ti mismo de la insatisfacción del usuario?

La primera idea que se te puede ocurrir es verificar si los datos proporcionados por el usuario son válidos y negarte a cooperar si los datos son incorrectos. En este caso, la verificación puede basarse en el hecho de que esperamos que la cadena de entrada contenga solo dígitos.

Ya deberías poder implementar esta verificación y escribirla tu mismo, ¿no es así? También es posible comprobar si la variable `value` es de tipo `int` (Python tiene un medio especial para este tipo de comprobaciones - es un operador llamado `is`). La revisión en sí puede verse de la siguiente manera:

```
type(value) is int
```

Su resultado es **true** si el valor actual de la variable es del tipo **int**.

Es posible que te sorprendas al saber que no queremos que realices ninguna validación preliminar de datos. ¿Por qué? Porque esta no es la forma que Python recomienda.

La rama try-except

En el mundo de Python, hay una regla que dice: "Es mejor pedir perdón que pedir permiso".

Detengámonos aquí por un momento. No nos malinterpretes. La regla se trata de otra cosa.

En realidad, la regla dice: "es mejor manejar un error cuando ocurre que tratar de evitarlo".

Observa el código en el editor:

```
1 try:
2     # Es un lugar donde
3     # tu puedes hacer algo
4     # sin pedir permiso.
5 except:
6     # Es un espacio dedicado
7     # exclusivamente para pedir perdón.
8
```

Puedes ver dos bloques aquí:

- el primero, comienza con la palabra clave reservada **try** este es el lugar donde se coloca el código que se sospecha que es riesgoso y puede terminar en caso de un error; nota: este tipo de error lleva por nombre **excepción**, mientras que la ocurrencia de la excepción se le denomina **generar** - podemos decir que se genera (o se generó) una excepción;
- el segundo, la parte del código que comienza con la palabra clave reservada **except** esta parte fue diseñada para manejar la excepción; depende de ti lo que quieras hacer aquí: puedes limpiar el desorden o simplemente puede barrer el problema debajo de la alfombra (aunque se prefiere la primera solución)

Entonces, podríamos decir que estos dos bloques funcionan así:

- la palabra clave reservada **try** marca el lugar donde intentas hacer algo sin permiso;
- la palabra clave reservada **except** comienza un lugar donde puedes mostrar tu talento para disculparte o pedir perdón.

Como puedes ver, este enfoque acepta errores (los trata como una parte normal de la vida del programa) en lugar de intensificar los esfuerzos para evitarlos por completo.

La excepción confirma la regla

Reescribamos el código para adoptar el enfoque de Python para la vida:

```
1 try:
2     value = int(input('Ingresa un número natural: '))
3     print('El recíproco de', value, 'es', 1/value)
4 except:
5     print('No se que hacer con', value)
6
```

Resumamos lo que hemos hablado:

- cualquier fragmento de código colocado entre try y except se ejecuta de una manera muy especial: cualquier error que ocurra aquí dentro no terminará la ejecución del programa. En cambio, el control saltará inmediatamente a la primera línea situada después de la palabra clave reservada except, y no se ejecutará ninguna otra línea del bloque try;
- el código en el bloque except se activa solo cuando se ha encontrado una excepción dentro del bloque try. No hay forma de llegar por ningún otro medio;
- cuando el bloque try o except se ejecutan con éxito, el control vuelve al proceso normal de ejecución y cualquier código ubicado más allá en el archivo fuente se ejecuta como si no hubiera pasado nada.

Cómo lidiar con más de una excepción

La respuesta obvia es "no" - hay más de una forma posible de plantear una excepción. Por ejemplo, un usuario puede ingresar cero como entrada - ¿puedes predecir lo que sucederá a continuación?

Sí, tienes razón - la división colocada dentro de la invocación de la función **print()** generará la **excepción ZeroDivisionError**. Como es de esperarse, el comportamiento del código será el mismo que en el caso anterior - el usuario verá el mensaje "No se que hacer con...", lo que parece bastante razonable en este contexto, pero también es posible que desees manejar este tipo de problema de una manera un poco diferente.

¿Es posible? Por supuesto que lo es. Hay al menos dos enfoques que puedes implementar aquí.

El primero de ellos es simple y complicado al mismo tiempo: puedes agregar dos bloques try por separado, uno que incluya la invocación de la función input() donde se puede generar la excepción ValueError y el segundo dedicado a manejar posibles problemas inducidos por la división. Ambos bloques try tendrían su propio except y de esa manera, tendrías un control total sobre dos errores diferentes.

Esta solución es buena, pero es un poco larga - el código se hincha innecesariamente. Además, no es el único peligro que te espera. Toma en cuenta que dejar el primer bloque try-except deja mucha incertidumbre - tendrás que agregar código adicional para asegurarte de que el valor que ingresó el usuario sea seguro para usar en la división. Así es como una solución aparentemente simple se vuelve demasiado complicada.

Afortunadamente, Python ofrece una forma más sencilla de afrontar este tipo de desafíos.

Dos excepciones después de un try

Observa el código en el editor. Como puedes ver, acabamos de agregar un segundo **except**. Esta no es la única diferencia - toma en cuenta que ambos except tienen **nombres** de excepción específicos. En esta variante, cada una de las excepciones esperadas tiene su propia forma de manejar el error, pero debe enfatizarse en que solo una de todas puede interceptar el control - **si se ejecuta una, todas las demás permanecen inactivas**.

```
1 try:
2     value = int(input('Ingresa un número natural: '))
3     print('El recíproco de', value, 'es', 1/value)
4 except ValueError:
5     print('No se que hacer con', value)
6 except ZeroDivisionError:
7     print('La división entre cero no está permitida en nuestro Universo.')
8
```

Además, la cantidad de **except** no está limitada - puedes especificar tantas o tan pocas como necesites, **pero no se te olvide que ninguna de las excepciones se puede especificar más de una vez**.

La excepción predeterminada y cómo usarla

El código ha cambiado de nuevo - ¿puedes ver la diferencia?

```
1 try:
2     value = int(input('Ingresa un número natural: '))
3     print('El recíproco de', value, 'es', 1/value)
4 except ValueError:
5     ('No se que hacer con.')
6 except ZeroDivisionError:
7     print('La división entre cero no está permitida en nuestro Universo.')
8 except:
9     print('Ha sucedido algo extraño, ¡lo siento!')
10
```

Hemos agregado un tercer **except**, pero esta vez **no tiene un nombre de excepción específico** - podemos decir que es anónimo o (lo que está más cerca de su función real) es el por defecto. Puedes esperar que cuando se genere una excepción y no haya un except dedicado a esa excepción, esta será manejada por la excepción por defecto.

Nota

¡el except por defecto debe ser el último except! ¡Siempre!

Algunas excepciones útiles

Analicemos con más detalle algunas excepciones útiles (o más bien, las más comunes) que puedes llegar a experimentar.

ZeroDivisionError

Esta aparece cuando intentas forzar a Python a realizar cualquier operación que provoque una división en la que el divisor es cero o no se puede distinguir de cero. Toma en cuenta que hay más de un operador de Python que puede hacer que se genere esta excepción. ¿Puedes adivinarlos todos? **Si, estos son: /, //, y %.**

ValueError

Espera esta excepción cuando estás manejando valores que pueden usarse de manera inapropiada en algún contexto. En general, esta excepción se genera cuando una función (como **int()** o **float()**) recibe un argumento de un tipo adecuado, pero su valor es inaceptable.

TypeError

Esta excepción aparece cuando intentas aplicar un dato cuyo tipo no se puede aceptar en el contexto actual. Mira el ejemplo:

```
1 | short_list = [1]
2 | one_value = short_list[0.5]
3 |
```

No está permitido usar un valor flotante como índice de una lista (la misma regla también se aplica a las tuplas). **TypeError** es un nombre adecuado para describir el problema y una excepción adecuada a generar.

AttributeError

Esta excepción llega - entre otras ocasiones - cuando intentas activar un método que no existe en un elemento con el que se está tratando. Por ejemplo:


```
1 short_list = [1]
2 short_list.append(2)
3 short_list.depend(3)
4
```

La tercera línea de nuestro ejemplo intenta hacer uso de un método que no está incluido en las listas. Este es el lugar donde se genera la excepción `AttributeError`.

SyntaxError

Esta excepción se genera cuando el control llega a una línea de código que viola la gramática de Python. Puede sonar extraño, pero algunos errores de este tipo no se pueden identificar sin ejecutar primero el código. Este tipo de comportamiento es típico de los lenguajes interpretados - el intérprete siempre trabaja con prisa y no tiene tiempo para escanear todo el código fuente. Se conforma con comprobar el código que se está ejecutando en el momento. Muy pronto se te presentará un ejemplo de esta categoría.

Es una mala idea manejar este tipo de excepciones en tus programas. Deberías producir código sin errores de sintaxis, en lugar de enmascarar las fallas que has causado.

Por qué no puedes evitar probar tu código

Ahora queremos contarte sobre el segundo lado de la lucha interminable contra los errores - el destino inevitable de la vida de un desarrollador. Como no puedes evitar la creación de errores en tu código, siempre debes estar listo para buscarlos y destruirlos. No entierres la cabeza en la arena - ignorar los errores no los hará desaparecer.

Un deber importante para los desarrolladores es probar el código recién creado, pero no debes olvidar que las pruebas no son una forma de demostrar que el código está libre de errores. Paradójicamente, lo único que las pruebas determinan, es que tu código contiene errores. No creas que puedes relajarte después de una prueba exitosa.

El segundo aspecto importante de las pruebas de software es estrictamente psicológico. Es una verdad conocida desde hace años que los autores - incluso aquellos que son confiables y conscientes de sí mismos - no pueden evaluar y verificar objetivamente sus trabajos.

Es por eso por lo que cada novelista necesita un editor y cada programador necesita un tester. Algunos dicen - con un poco de rencor, pero con sinceridad - que los desarrolladores prueban su código para mostrar su perfección, no para encontrar problemas que puedan frustrarlos. Los testers o probadores están libres de tales dilemas, y es por eso por lo que su trabajo es más efectivo y rentable.

Por supuesto, esto no te exime de estar atento y cauteloso. Prueba tu código lo mejor que puedas. No facilites demasiado el trabajo a los probadores.

Rastreando las rutas de ejecución

Supón que acabas de terminar de escribir este fragmento de código:

```
1 temperature = float(input('Ingresa la temperatura actual:'))
2
3 if temperature > 0:
4     print("Por encima de cero")
5 elif temperature < 0:
6     print("Por debajo de cero")
7 else:
8     print("Cero")
9
```

Existen tres rutas o caminos de ejecución independientes en el código - ¿puedes verlas? Están determinadas por las sentencias **if-elif-else**. Por supuesto, las rutas de ejecución pueden construirse mediante muchas otras sentencias como bucles, o incluso bloques try-except.

Si vas a probar tu código de manera justa y quieres dormir profundamente y soñar sin pesadillas (las pesadillas sobre errores pueden ser devastadoras para el rendimiento de un desarrollador), estás obligado a preparar un conjunto de datos de prueba que obligará a tu código a negociar todos los posibles caminos.

En nuestro ejemplo, el conjunto debe contener al menos tres valores flotantes: uno positivo, uno negativo y cero.

Pruebas, pruebas y más pruebas

La respuesta es más simple de lo esperado y también un poco decepcionante. Python - como seguramente sabes - es un **lenguaje interpretado**. Esto significa que el código fuente se analiza y ejecuta al mismo tiempo. En consecuencia, es posible que Python no tenga tiempo para analizar las líneas de código que no están sujetas a ejecución. Como dice un antiguo dicho de los desarrolladores: "es una característica, no un error" (no utilices esta frase para justificar el comportamiento extraño de tu código).

¿Entiendes ahora por qué el pasar por todos los caminos de ejecución es tan vital e inevitable?

Supongamos que terminas tu código y que las pruebas que has realizado son exitosas. Entregas tu código a los tester y - ¡afortunadamente! - encontraron algunos errores en él. Estamos usando la palabra "afortunadamente" de manera completamente consciente. Debes aceptar que, en primer lugar, los tester son los mejores amigos del desarrollador - no debes tratar a los errores que ellos encuentran como una ofensa o una malignidad; y, en segundo lugar, cada error que encuentran los tester es un error que no afectará a los usuarios. Ambos factores son valiosos y merecen tu atención.

Ya sabes que tu código contiene un error o errores (lo segundo es más probable). Ahora, ¿cómo los localizas y cómo arreglas tu código?

Error frente a depuración (Bug vs. debug)

La medida básica que un desarrollador puede utilizar contra los errores es - como era de esperarse - un **depurador**, mientras que el proceso durante el cual se eliminan los errores del código se llama depuración.

Un depurador es un software especializado que puede controlar cómo se ejecuta tu programa. Con el depurador, puedes ejecutar tu código línea por línea, inspeccionar todos los estados de las variables y cambiar sus valores en cualquier momento sin modificar el código fuente, detener la ejecución del programa cuando se cumplen o no ciertas condiciones, y hacer muchas otras tareas útiles.

Podemos decir que todo IDLE está equipado con un depurador más o menos avanzado. Incluso IDLE tiene uno, aunque puedes encontrar su manejo un poco complicado y problemático. Si deseas utilizar el depurador integrado de IDLE, debes activarlo mediante la opción "**Debug**" en la barra de menú de la ventana principal de IDLE. Es el punto de partida para la depuración.

Haz clic aquí (<https://www.cs.uky.edu/~keen/help/debug-tutorial/debug.html>) para ver las capturas de pantalla que muestran el depurador IDLE durante una sesión de depuración simple.

Puedes ver cómo el depurador visualiza las variables y los valores de los parámetros, y observa la pila de llamadas que muestra la cadena de invocaciones que van desde la función actualmente ejecutada hasta el nivel del intérprete.

Si deseas saber más sobre el depurador IDLE, consulta la documentación IDLE (<https://docs.python.org/3/library/idle.html>).

print debugging (depuración)

Esta forma de depuración, que se puede aplicar a tu código mediante cualquier tipo de depurador, a veces se denomina depuración interactiva. El significado del término se explica por sí mismo - el proceso necesita su interacción (la del desarrollador) para que se lleve a cabo.

Algunas otras técnicas de depuración se pueden utilizar para cazar errores. Es posible que no puedas o no quieras usar un depurador (las razones pueden variar). ¿Estás entonces indefenso? ¡Absolutamente no!

Puedes utilizar una de las tácticas de depuración más simples y antiguas (pero aún útil) conocida como la depuración por impresión. El nombre habla por sí mismo - simplemente insertas varias invocaciones **print()** adicionales dentro de tu código para generar datos que ilustran la ruta que tu código está negociando actualmente. Puedes imprimir los valores de las variables que pueden afectar la ejecución.



Estas impresiones pueden generar texto significativo como "Estoy aquí", "Ingresé a la función foo()", "El resultado es 0", o pueden contener secuencias de caracteres que solo tu puedes leer. Por favor, no uses palabras obscenas o indecentes para ese propósito, aunque puedas sentir una fuerte tentación - tu reputación puede arruinarse en un momento si estas payasadas se filtran al público.

Como puedes ver, este tipo de depuración no es realmente interactiva en lo absoluto, o es interactiva solo en pequeña medida, cuando decides aplicar la función `input()` para detener o retrasar la ejecución del código.

Una vez que se encuentran y eliminan los errores, las impresiones adicionales pueden comentarse o eliminarse - tu decides. No permitas que se ejecuten en el código final.

Algunos consejos útiles

Aquí hay algunos consejos que pueden ayudarte a encontrar y eliminar errores. Ninguno de ellos es definitivo. Úsalos de manera flexible y confía en tu intuición. No te creas a ti mismo - comprueba todo dos veces.

- **Intenta decirle a alguien** (por ejemplo, a tu amigo o compañero de trabajo) qué es lo que se espera que haga tu código y cómo se espera que se comporte. Sé concreto y no omitas detalles. Responde todas las preguntas que te hagan. Es probable que te des cuenta de la causa del problema mientras cuentas tu historia, ya que el hablar activa esas partes de tu cerebro que permanecen inactivas mientras codificas.
- **Intenta aislar el problema.** Puedes extraer la parte de tu código que se sospecha que es responsable de tus problemas y ejecutarla por separado. Puedes comentar partes del código para ocultar el problema. Asigna valores concretos a las variables en lugar de leerlos desde la consola. Prueba tus funciones aplicando valores de argumentos predecibles. Analiza el código cuidadosamente. Léelo en voz alta.
- Si el error apareció recientemente y no había aparecido antes, **analiza todos los cambios que has introducido en tu código** - uno de ellos puede ser la razón.
- **Tómate un descanso**, bebe una taza de café, toma a tu perro y sal a caminar, lee un buen libro, incluso dos, haz una llamada telefónica a tu mejor amigo - te sorprenderás de la frecuencia con la que esto ayuda.
- **Se optimista** - eventualmente encontrarás el error; te lo prometemos.

Pruebas unitarias - un mayor nivel de codificación

También existe una técnica de programación importante y ampliamente utilizada que tendrás que adoptar tarde o temprano durante tu carrera de desarrollador - se llama **prueba unitaria**. El nombre puede ser un poco confuso, ya que no se trata solo de probar el software, sino también (y, sobre todo) de cómo se escribe el código.

Para resumir la historia - las pruebas unitarias asumen que las pruebas son partes inseparables del código y la preparación de los datos de prueba es una parte inseparable de la codificación. Esto significa que cuando escribes una función o un conjunto de funciones cooperativas, también estás obligado a crear un conjunto de datos para los cuales el comportamiento de tu código es predecible y conocido.

Además, debes equipar a tu código con una interfaz que pueda ser utilizada por un entorno de pruebas automatizado. En este enfoque, cualquier enmienda realizada al código (incluso la menos significativa) debe ir seguida de la ejecución de todas las pruebas unitarias que acompañan al código fuente.

Para estandarizar este enfoque y facilitar su aplicación, Python proporciona un módulo dedicado llamado unittest. No vamos a discutirlo aquí - es un tema amplio y complejo.