# Modulo 4 - Funciones, Tuplas, Diccionarios, Excepciones y Procesamiento de Datos

- Estructuración de código y el concepto de función.
- Invocación de funciones y devolución de resultados de una función.
- Alcance de nombres y sombreado de variables.
- Tuplas y su propósito: construcción y uso de tuplas.
- Diccionarios y su propósito: construcción y uso de diccionarios.
- Introducción a las excepciones en Python.

## ¿Por qué necesitamos funciones?

Hasta ahorita 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.

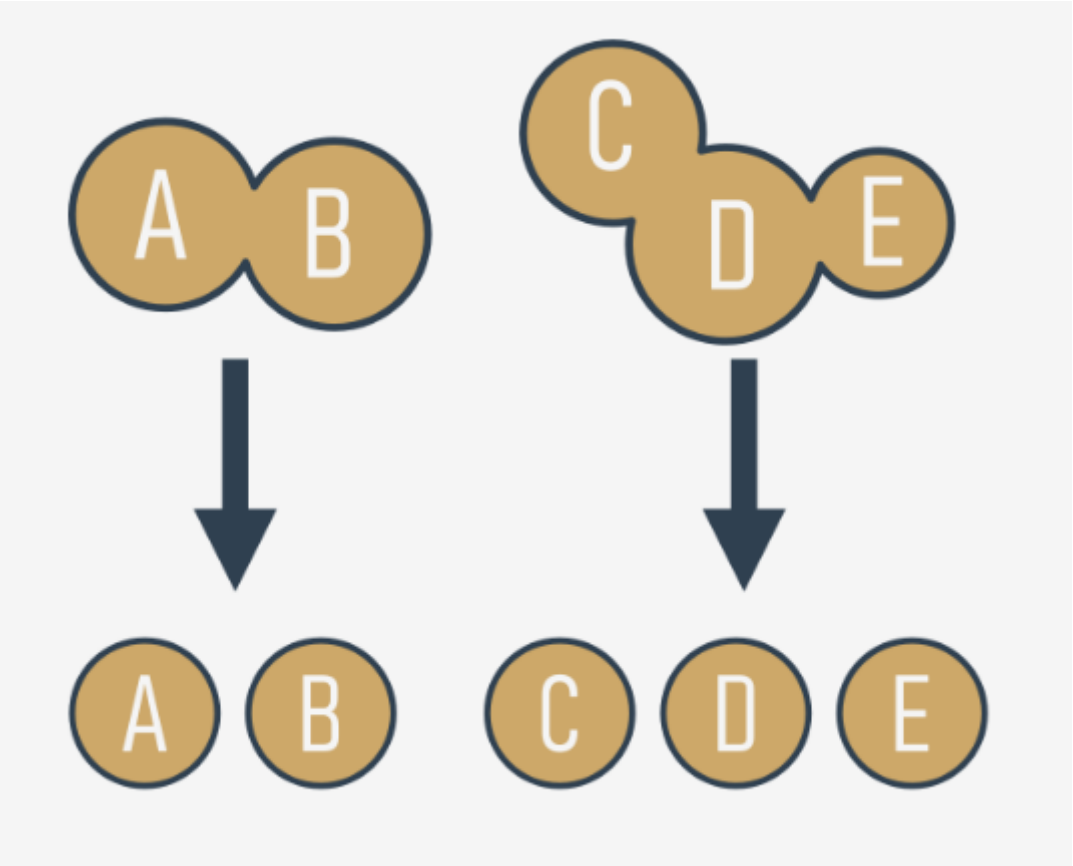
Definamos la primer condición por la cual es una buena idea comenzar a escribir funciones propias: si un fragmento de código comienza a aparecer en más de una ocasión, considera la posibilidad de aislarlo en la forma de una función invocando la función desde el lugar en el que originalmente se encontraba.

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.

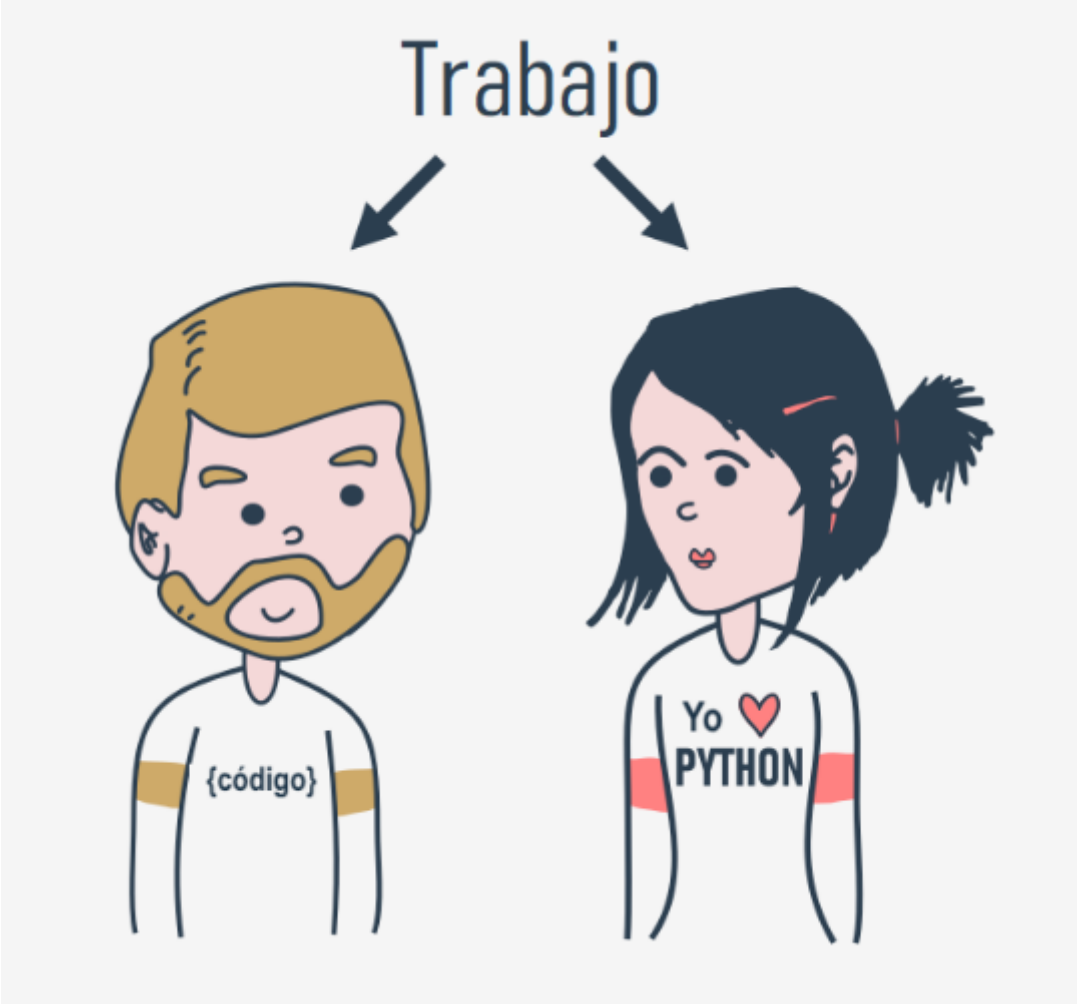



Existe una segunda condición: si un fragmento de código se hace tan extenso que leerlo o entenderlo se hace complicado, considera dividirlo pequeños problemas por separado e implementa cada uno de ellos como una función independiente.

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 el. 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 clarificara 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 acerca de esto en un momento).
- **Directamente del código**: tu puedes escribir tus propias funciones, colocarlas dentro del código, y usarlas libremente.
- Existe una posibilidad más, pero se relaciona con clases, se omitirá por ahora.

# Tu primera función

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

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

```
def function_name():
    function_body
```

- 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 (ahorita 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**:

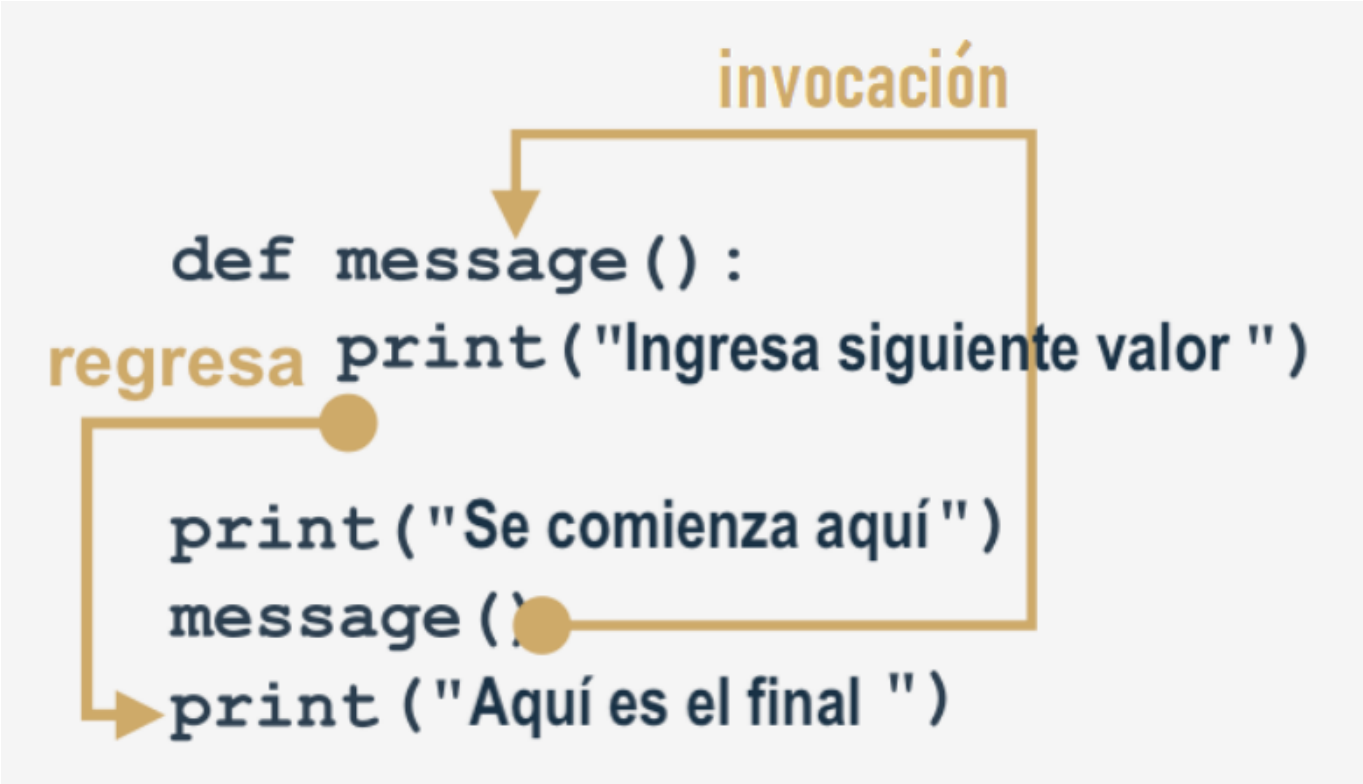
```
def message():
    print("Ingresa un valor: ")
```

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

```
def message():
    print("Ingresa un valor: ")

print("Se comienza aquí.")
message()
print("Se termina aquí.")
```

# el funcionamiento de las funciones




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.

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

El asignar un valor al nombre «message» causa que Python olvide su rol anterior. La función con el nombre de message ya no estará disponible.

Afortunadamente, es posible combinar o mezclar el código con las funciones - no es forzoso colocar todas las funciones al inicio del archivo fuente.

## 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 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.
- 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.

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:

```
def message(number):
    print("Ingresa un número:", number)

number = 1234
message(1)
print(number)
```

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.

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

## 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**.

## Paso de argumentos con 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:

```
def introduction(first_name, last_name):
    print("Hola, mi nombre es", first_name, last_name)
```

```
introduction(first_name = "James", last_name = "Bond")
introduction(last_name = "Skywalker", first_name = "Luke")
```

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.

Debes de poder predecir la salida. Ejecuta el código y verifica tu respuesta.

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

El siguiente código provocará un error de ejecución:

```
def introduction(first_name, last_name):
    print("Hola, mi nombre es", first_name, last_name)

introduction(surname="Skywalker", first_name="Luke")
```

Esto es lo que Python arrojará:

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

## Combinar argumentos posicionales y de palabra 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.

```
def adding(a, b, c):
    print(a, "+", b, "+", c, "=", a + b + c)

adding(1, 2, 3) # 1 + 2 + 3 = 6
adding(c = 1, a = 2, b = 3) # 2 + 3 + 1 = 6
adding(3, c = 1, b = 2) # 3 + 2 + 1 = 6
```

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

```
adding(3, a = 1, b = 2) # ERROR
```

## 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 default para el parámetro se asigna de la siguiente manera:

```
def introduction(first_name, last_name="González"):
    print("Hola, mi nombre es", first_name, last_name)
```

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

Invoquemos la función de manera normal:

```
introduction("Jorge", "Pérez") # Hola, mi nombre es Jorge Pérez.
```

No parece haber cambiado algo, pero cuando se invoca la función de una manera inusual, como esta:

```
introduction("Enrique") # Hola, mi nombre es Enrique González
```

o así:

```
introduction(first_name="Guillermo") # Hola, mi nombre es Guillermo González
```

Es importante recordar que **primero se especifican los argumentos posicionales y después los de palabras clave**. Es por esa razón que si se intenta ejecutar el siguiente código:

```
def subtra(a, b):
    print(a - b)

subtra(5, b=2)    # salida: 3
subtra(a=5, 2)    # Syntax Error
```

```
def add_numbers(a, b=2, c):
    print(a + b + c)

add_numbers(a=1, c=3) # SyntaxError - a non-default argument (c) follows a default
argument (b=2)
```

## Efectos y resultados: la instrucción return

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

La primera consiste en la palabra reservada en sí, sin nada que la siga.

Cuando se emplea dentro de una función, provoca la terminación inmediata de la ejecución de la función, y un retorno instantáneo (de ahí el nombre) al punto de invocación.

Nota: si una función no está destinada a producir un resultado, emplear la instrucción returnno es obligatorio, se ejecutará implícitamente al final de la función.

De cualquier manera, se puede emplear para terminar las actividades de una función, antes de que el control llegue a la última línea de la función.

Consideremos la siguiente función:

```
def happy_new_year(wishes = True):
```

```
    print("Tres...")
    print("Dos...")
    print("Uno...")
    if not wishes:
        return

    print("¡Feliz año nuevo!")
```

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:

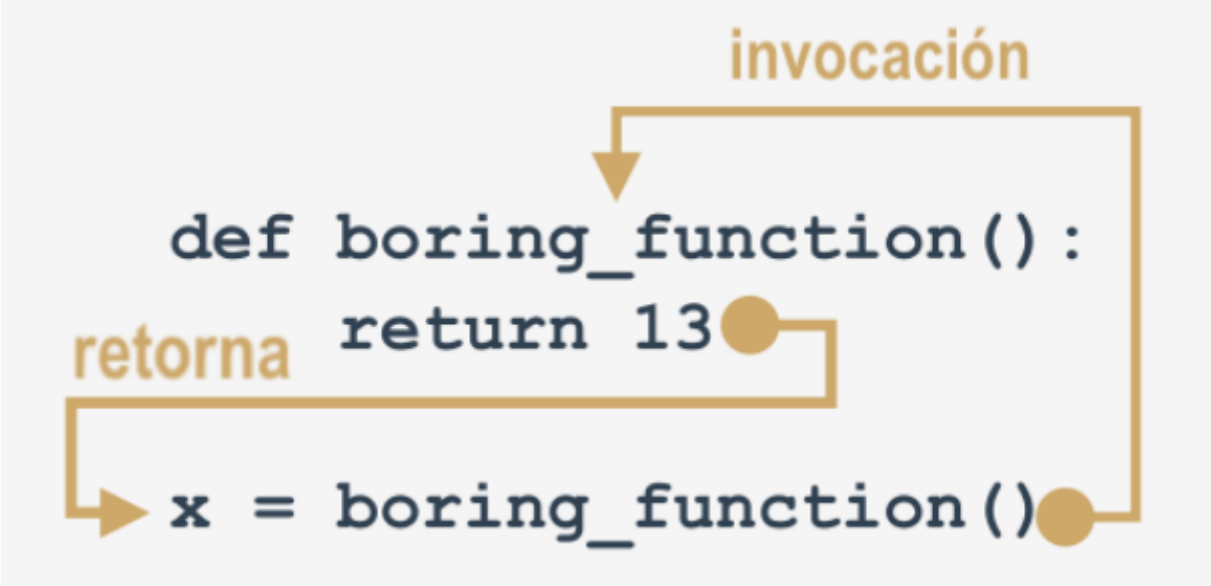
```
def boring_function():
    return 123

x = boring_function()

print("La función boring_function ha devuelto su resultado. Es:", x)
```

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.

Ten en cuenta que no estamos siendo muy educados aquí: la función devuelve un valor y lo ignoramos (no lo usamos de ninguna manera):

```
def boring_function():
    print("'Modo aburrimiento' ON.")
    return 123

print("¡Esta lección es interesante!")
boring_function()
print("Esta lección es aburrida...")
```

El programa produce el siguiente resultado:

```
¡Esta lección es interesante!
'Modo aburrimiento' ON.
Esta lección es aburrida...
```

No olvides:

- Siempre se te permite ignorar el resultado de la función y estar satisfecho con el efecto de la función (si la función tiene alguno).
- Si una función intenta devolver un resultado útil, debe contener la segunda variante de la instrucción return.

# Unas pocas palabras acerca de 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:

```
value = None
if value is None:
    print("Lo siento, no contienes ningún valor")
```

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

## Efectos y resultados: listas y funciones

### ¿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.

### ¿Puede una lista ser el resultado de una función?

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

## ejercicio

```
Tu tarea es escribir y probar una función que toma un argumento (un año) y devuelve
True si el año es un año bisiesto, o False si no lo es.

Parte del esqueleto de la función ya está en el editor.
```

```
Nota: también hemos preparado un breve código de prueba, que puedes utilizar para
probar tu función.

El código utiliza dos listas: una con los datos de prueba y la otra con los
resultados esperados. El código te dirá si alguno de tus resultados no es válido.
```

```
def is_year_leap(year):
  if year >= 1582:
    if year % 4 != 0: retorno = False
    elif year % 100 != 0: retorno = True
    elif year % 400 != 0: retorno = False
    else: retorno = True
  else:
    retorno = False

  return retorno


test_data = [1900, 2000, 2016, 1987]
test_results = [False, True, True, False]
for i in range(len(test_data)):
    yr = test_data[i]
    print(yr,"->",end="")
    result = is_year_leap(yr)
    if result == test_results[i]:
        print("OK")
    else:
        print("Fallido")
```

## ejercicio

```
Tu tarea es escribir y probar una función que toma dos argumentos (un año y un mes)
y devuelve el número de días del mes/año dado (mientras que solo febrero es
sensible al valor year, tu función debería ser universal).

La parte inicial de la función está lista. Ahora, haz que la función devuelva None
si los argumentos no tienen sentido.

Por supuesto, puedes (y debes) utilizar la función previamente escrita y probada
(LABORATORIO 4.1.3.6). Puede ser muy útil. Te recomendamos que utilices una lista
con los meses. Puedes crearla dentro de la función; este truco acortará
significativamente el código.

Hemos preparado un código de prueba. Amplíalo para incluir más casos de prueba.
```

```
def is_year_leap(year):
  if year >= 1582:
    if year % 4 != 0: retorno = False
    elif year % 100 != 0: retorno = True
    elif year % 400 != 0: retorno = False
    else: retorno = True
  else:
    retorno = False
```

```
  return retorno

def days_in_month(year, month):
  dias = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
  if is_year_leap(year):
      dias[1] = 29
  return dias[month-1]

test_years = [1900, 2000, 2016, 1987]
test_months = [2, 2, 1, 11]
test_results = [28, 29, 31, 30]
for i in range(len(test_years)):
    yr = test_years[i]
    mo = test_months[i]
    print(yr, mo, "->", end="")
    result = days_in_month(yr, mo)
    if result == test_results[i]:
        print("OK")
    else:
        print("Fallido")
```

### ejercicio

```
Tu tarea es escribir y probar una función que toma tres argumentos (un año, un mes
y un día del mes) y devuelve el día correspondiente del año, o devuelve None si
cualquiera de los argumentos no es válido.

Debes utilizar las funciones previamente escritas y probadas. Agrega algunos casos
de prueba al código. Esta prueba es solo el comienzo.
```

## Las funciones y sus 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.

```
def scope_test():
    x = 123


scope_test()
print(x)
```

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

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?

```
def my_function():
    print("¿Conozco a la variable?", var)


var = 1
my_function()
print(var)
```

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

```
¿Conozco a la variable? 1
1
```

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:

```
def my_function():
    var = 2
    print("¿Conozco a la variable?", var)


var = 1
my_function()
print(var)
```

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

```
¿Conozco a la variable? 2
1
```

¿Qué es lo que ocurrió?

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

## Las funciones y sus alcances: la palabra clave reservada 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 incomodo.

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:

```
global name
global name1, name2, ...
```

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).

```
def my_function():
    global var
    var = 2
    print("¿Conozco a aquella variable?", var)


var = 1
my_function()
print(var)
```

Se ha agregado la palabra global a la función.

El código ahora da como salida:

```
¿Conozco a aquella variable? 2
2
```

## Como 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?

```
def my_function(n):
    print("Yo recibí", n)
    n += 1
    print("Ahora tengo", n)


var = 1
my_function(var)
print(var)
```

La salida del código es:

```
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.

Vale la pena revisar cómo funciona esto con las listas (¿Recuerdas las peculiaridades de asignar rebanadas de listas en lugar de asignar la lista entera?)

El siguiente ejemplo arrojará luz sobre el asunto:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    my_list_1 = [0, 1]
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)


my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

La salida del código es:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [0, 1]
Print #4: [2, 3]
Print #5: [2, 3]
```

Parece ser que se sigue aplicando la misma regla.

Finalmente, la diferencia se puede observar en el siguiente ejemplo:

```
def my_function(my_list_1):
    print("Print #1:", my_list_1)
    print("Print #2:", my_list_2)
    del my_list_1[0]  # Presta atención a esta línea.
    print("Print #3:", my_list_1)
    print("Print #4:", my_list_2)

my_list_2 = [2, 3]
my_function(my_list_2)
print("Print #5:", my_list_2)
```

No se modifica el valor del parámetro

```
my_list_1
```

(ya se sabe que no afectará el argumento), en lugar de ello se modificará la lista identificada por el.

El resultado puede ser sorprendente. Ejecuta el código y verifícalo:

```
Print #1: [2, 3]
Print #2: [2, 3]
Print #3: [3]
Print #4: [3]
Print #5: [3]
```

¿Lo puedes explicar?

Intentémoslo:

- Si el argumento es una lista, el cambiar el valor del parámetro correspondiente no afecta la lista (Recuerda: las variables que contienen listas son almacenadas de manera diferente que las escalares).
- Pero si se modifica la lista identificada por el parámetro (Nota: ¡La lista, no el parámetro!), la lista reflejará el cambio.

## Algunas funcione simples: 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:

```
def fib(n):
    if n < 1:
        return None
    if n < 3:
        return 1

    elem_1 = elem_2 = 1
    the_sum = 0
    for i in range(3, n + 1):
        the_sum = elem_1 + elem_2
        elem_1, elem_2 = elem_2, the_sum
    return the_sum


for n in range(1, 10):  # probando
    print(n, "->", fib(n))
```

¿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:

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

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.

El factorial también tiene un lado recursivo. Observa:

```
n! = 1 × 2 × 3 × ... × n-1 × n
```

Es obvio que:

```
1 × 2 × 3 × ... × n-1 = (n-1)!
```

Entonces, finalmente, el resultado es:

```
n! = (n-1)! × n
```

Esto se empleará en nuestra nueva solución.

```
def factorial_function(n):
    if n < 0:
        return None
    if n < 2:
        return 1
    return n * factorial_function(n - 1)
```

## Tipos de secuencias 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.

# ¿Qué es una tupla?

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:

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

Se definieron dos tuplas, ambas contienen cuatro elementos.

A continuación se imprimen en consola:

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

print(tuple_1)
print(tuple_2)
```

Esto es lo que se muestra en consola:

```
(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:

```
empty_tuple = ()
```

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.

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

print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:])
print(my_tuple[:-2])

for elem in my_tuple:
    print(elem)
```

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

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

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.

¿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.

```
my_tuple = (1, 10, 100)

t1 = my_tuple + (1000, 10000)
t2 = my_tuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in my_tuple)
print(-10 not in my_tuple)
```

La salida es la siguiente:

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

Una de las propiedades de las tuplas más útiles es que pueden **aparecer en el lado izquierdo del operador de asignación**. Este fenómeno ya se vio con anterioridad, cuando fue necesario encontrar una manera de intercambiar los valores entre dos variables.

Observa el siguiente fragmento de código:

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

Muestra tres tuplas interactuando en efecto, los valores almacenados en ellas «circulan» entre ellas. t1 se convierte en t2, t2 se convierte en t3, y t3 se convierte en t1.

Nota: el ejemplo presenta un importante hecho mas: los **elementos de una tupla pueden ser variables**, no solo literales. Además, pueden ser expresiones si se encuentran en el lado derecho del operador de asignación.

## ¿Qué es un diccionario?

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**.

Para explicar lo que es un diccionario en Python, es importante comprender de manera literal lo que es un diccionario.

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 **clave(key)**. La palabra que se obtiene del diccionario es denominada **valor**.

Esto significa que un diccionario es un conjunto de pares de **claves y valores**. Nota:

- Cada clave debe de ser única. No es posible tener una clave duplicada.
- Una clave puede ser un tipo de dato de cualquier tipo: puede ser un número (entero o flotante), o incluso una cadena.
- 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.

## ¿Cómo crear un diccionario?

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

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}
phone_numbers = {'jefe': 5551234567, 'Suzy': 22657854310}
empty_dictionary = {}

print(dictionary)
print(phone_numbers)
print(empty_dictionary)
```

En este primer ejemplo, el diccionario emplea claves y valores las cuales ambas son cadenas. En el segundo, las claves con 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** - nada inusual.

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?

Si deseas obtener cualquiera de los valores, debes de proporcionar una clave válida:

```
print(dictionary['gato'])
print(phone_numbers['Suzy'])
```

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'.

El fragmento de código da las siguientes salidas:

```
chat
5557654321
```

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

```
print(phone_numbers['presidente'])
```

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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}
words = ['gato', 'león', 'caballo']

for word in words:
    if word in dictionary:
        print(word, "->", dictionary[word])
    else:
        print(word, "no está en el diccionario")
```

La salida del código es la siguiente:

```
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:

```
# Ejemplo 1:
dictionary = {
              "gato": "chat",
              "perro": "chien",
              "caballo": "cheval"
              }

# Ejemplo 2:
phone_numbers = {'jefe': 5551234567,
                 'Suzy': 22657854310
                }
```

Este tipo de formato se llama **sangría francesa**.

# ¿Cómo utilizar un diccionario? 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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for key in dictionary.keys():
    print(key, "->", dictionary[key])
```

El código produce la siguiente salida:

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

# La función sorted()

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

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

La función **sorted()** hará su mejor esfuerzo y la salida será la siguiente:

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

# ¿Cómo utilizar un diccionario? Los métodos item() y values()

Otra manera de hacerlo es utilizar el método **items()**. Este método **regresa 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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}
```

```
for index, value in dictionary.items():
    print(index, "->", value)
```

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

El ejemplo imprime lo siguiente:

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

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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

for value in dictionary.values():
    print(value)
```

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

Esta es la salida esperada:

```
chat
chien
cheval
```

## ¿Cómo utilizar un diccionario? 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.

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

dictionary['gato'] = 'minou'
print(dictionary)
```

La salida es:

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

## 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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dictionary['cisne'] = 'cygne'
print(dictionary)
```

El ejemplo muestra como salida:

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

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

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

dictionary.update({"pato": "canard"})
print(dictionary)
```

## 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:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}

del dictionary['perro']
print(dictionary)
```

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

El ejemplo da como salida:

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

### EXTRA

Para eliminar el ultimo elemento de la lista, se puede emplear el método **popitem()**:

```
dictionary = {"gato" : "chat", "perro" : "chien", "caballo" : "cheval"}
```

```
dictionary.popitem()
print(dictionary)    # salida: {'gato': 'chat', 'perro': 'chien'}
```

En versiones anteriores de Python, por ejemplo, antes de la 3.6.7, el método **popitem()** elimina un elemento al azar del diccionario.

## 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 en el segundo parte de la serie del curso). Una lista con todos los nombre y el promedio de cada alumno debe ser mostrada al final.

```
school_class = {}

while True:
    name = input("Ingresa el nombre del estudiante: ")
    if name == '':
        break
    score = int(input("Ingresa la calificación del estudiante (0-10): "))
    if score not in range(0, 11):
        break
    if name in school_class:
        school_class[name] += (score,)
    else:
        school_class[name] = (score,)
for name in sorted(school_class.keys()):
    adding = 0
    counter = 0
    for score in school_class[name]:
        adding += score
        counter += 1
    print(name, ":", adding / counter)
```

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, saldrémos 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 1-10).
- 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, tomado 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:

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

## Puntos Clave: Tuplas

1. Las Tuplas son colecciones de datos ordenadas e inmutables. Se puede pensar en ellas como listas inmutables. Se definen con paréntesis:

```
my_tuple = (1, 2, True, "una cadena", (3, 4), [5, 6], None)
print(my_tuple)

my_list = [1, 2, True, "una cadena", (3, 4), [5, 6], None]
print(my_list)
```

Cada elemento de la tupla puede ser de un tipo de dato diferente (por ejemplo, enteros, cadenas, boleanos, etc.). Las tuplas pueden contener otras tuplas o listas (y viceversa).

2. Se puede crear una tupla vacía de la siguiente manera:

```
empty_tuple = ()
print(type(empty_tuple))    # salida: <class 'tuple'>
```

3. La tupla de un solo elemento se define de la siguiente manera:

```
one_elem_tuple_1 = ("uno", )    # Paréntesis y una coma.
one_elem_tuple_2 = "uno",       # Sin paréntesis, solo la coma.
```

Si se elimina la coma, Python creará una variable no una tupla:

```
my_tuple_1 = 1,
print(type(my_tuple_1))    # salida: <class 'tuple'>

my_tuple_2 = 1             # Esto no es una tupla.
print(type(my_tuple_2))    # salida: <class 'int'>
```

4. Se pueden acceder los elementos de la tupla al indexarlos:

```
my_tuple = (1, 2.0, "cadena", [3, 4], (5, ), True)
print(my_tuple[3])    # salida: [3, 4]
```

5. Las tuplas son immutable, lo que significa que no se puede agregar, modificar, cambiar o quitar elementos. El siguiente fragmento de código provocará una excepción:

```
my_tuple = (1, 2.0, "cadena", [3, 4], (5, ), True)
my_tuple[2] = "guitarra"    # La excepción TypeError será lanzada.
```

Sin embargo, se puede eliminar la tupla completa:

```
my_tuple = 1, 2, 3,
del my_tuple
print(my_tuple)    # NameError: name 'my_tuple' is not defined
```

6. Puedes iterar a través de los elementos de una tupla con un bucle (Ejemplo 1), verificar si un elemento o no esta presente en la tupla (Ejemplo 2), emplear la función len() para verificar cuantos elementos existen en la tupla (Ejemplo 3), o incluso unir o multiplicar tuplas (Ejemplo 4):

```
# Ejemplo 1
tuple_1 = (1, 2, 3)
for elem in tuple_1:
    print(elem)

# Ejemplo 2
tuple_2 = (1, 2, 3, 4)
print(5 in tuple_2)
print(5 not in tuple_2)

# Ejemplo 3
tuple_3 = (1, 2, 3, 5)
print(len(tuple_3))

# Ejemplo 4
tuple_4 = tuple_1 + tuple_2
tuple_5 = tuple_3 * 2

print(tuple_4)
print(tuple_5)
```

## EXTRA

También se puede crear una tupla utilizando la función integrada de Python **tuple()**. Esto es particularmente útil cuando se desea convertir un iterable (por ejemplo, una lista, rango, cadena, etcétera) en una tupla:

```
my_tuple = tuple((1, 2, "cadena"))
```

```
print(my_tuple)

my_list = [2, 4, 6]
print(my_list)    # salida: [2, 4, 6]
print(type(my_list))    # salida: <class 'list'>
tup = tuple(my_list)
print(tup)    # salida: (2, 4, 6)
print(type(tup))    # salida: <class 'tuple'>
```

De la misma manera, cuando se desea convertir un iterable en una lista, se puede emplear la función integrada de Python denominada **list()**:

```
tup = 1, 2, 3,
my_list = list(tup)
print(type(my_list))    # salida: <class 'list'>
```

### Puntos Clave: Diccionarios

1. Los diccionarios son *colecciones indexadas de datos, mutables y desordenadas. (*En Python 3.6x los diccionarios están ordenados de manera predeterminada.

Cada diccionario es un par de clave : valor. Se puede crear empleado la siguiente sintaxis:

```
my_dictionary = {
    key1: value1,
    key2: value2,
    key3: value3,
    }
```

2. Si se desea acceder a un elemento del diccionario, se puede hacer haciendo referencia a su clave colocándola dentro de corchetes (Ejemplo 1) o utilizando el método `get()` (Ejemplo 2):

```
pol_esp_dictionary = {
    "kwiat": "flor",
    "woda": "agua",
    "gleba": "tierra"
    }

item_1 = pol_esp_dictionary["gleba"]    # Ejemplo 1.
print(item_1)    # salida: tierra

item_2 = pol_esp_dictionary.get("woda")    # Ejemplo 2.
print(item_2)    # salida: agua
```

3. Si se desea cambiar el valor asociado a una clave específica, se puede hacer haciendo referencia a la clave del elemento, a continuación se muestra un ejemplo:

```
pol_esp_dictionary = {
    "zamek" : "castillo",
    "woda"  : "agua",
    "gleba" : "tierra"
    }
```

```
pol_esp_dictionary["zamek"] = "cerradura"
item = pol_esp_dictionary["zamek"]
print(item)  # salida: cerradura
```

4. Para agregar o eliminar una clave (junto con su valor asociado), emplea la siguiente sintaxis:

```
phonebook = {}    # un diccionario vacío

phonebook["Adán"] = 3456783958    # crear/agregar un par clave-valor
print(phonebook)    # salida: {'Adán': 3456783958}

del phonebook["Adán"]
print(phonebook)    # salida: {}
```

Además, se puede insertar un elemento a un diccionario utilizando el método `update()`, y eliminar el ultimo elemento con el método `popitem()`, por ejemplo:

```
pol_esp_dictionary = {"kwiat": "flor"}

pol_esp_dictionary.update({"gleba": "tierra"})
print(pol_esp_dictionary)    # salida: {'kwiat': 'flor', 'gleba': 'tierra'}

pol_esp_dictionary.popitem()
print(pol_esp_dictionary)    # salida: {'kwiat': 'flor'}
```

5. Se puede emplear el bucle `for` para iterar a través del diccionario, por ejemplo:

```
pol_esp_dictionary = {
    "zamek": "castillo",
    "woda": "agua",
    "gleba": "tierra"
    }

for item in pol_esp_dictionary:
    print(item)

# salida: zamek
#         woda
#         gleba
```

6. Si deseas examinar los elementos (claves y valores) del diccionario, puedes emplear el método `items()`, por ejemplo:

```
pol_esp_dictionary = {
    "zamek" : "castillo",
    "woda"  : "agua",
    "gleba" : "tierra"
    }

for key, value in pol_esp_dictionary.items():
    print("Pol/Esp ->", key, ":", value)
```

7. Para comprobar si una clave existe en un diccionario, se puede emplear la palabra clave reservada in:

```
pol_esp_dictionary = {
    "zamek" : "castillo",
    "woda"  : "agua",
    "gleba" : "tierra"
    }

if "zamek" in pol_esp_dictionary:
    print("Si")
else:
    print("No")
```

8. Se puede emplear la palabra reservada `del` para eliminar un elemento, o un diccionario entero. Para eliminar todos los elementos de un diccionario se debe emplear el método `clear()`:

```
pol_esp_dictionary = {
    "zamek" : "castillo",
    "woda"  : "agua",
    "gleba" : "tierra"
    }

print(len(pol_esp_dictionary))    # salida: 3
del pol_esp_dictionary["zamek"]    # eliminar un elemento
print(len(pol_esp_dictionary))    # salida: 2

pol_esp_dictionary.clear()   # eliminar todos los elementos
print(len(pol_esp_dictionary))    # salida: 0

del pol_esp_dictionary    # elimina el diccionario
```

9. Para copiar un diccionario, emplea el método `copy()`:

```
pol_esp_dictionary = {
    "zamek" : "castillo",
    "woda"  : "agua",
    "gleba" : "tierra"
    }

copy_dictionary = pol_esp_dictionary.copy()
```

# Excepciones

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» (bicho en inglés), que es una manifestación de una creencia bien

establecida de que, si un programa funciona mal, esto debe ser causado por bichos maliciosos que viven dentro del hardware de la computadora y causan cortocircuitos u otras interferencias.

Esta idea no es tan descabellada como puede parecer: incidentes de este tipo eran comunes en tiempos en que las computadoras ocupaban grandes pasillos, consumían kilovatios de electricidad y producían enormes cantidades de calor. Afortunadamente, o no, estos tiempos se han ido para siempre y los únicos errores que pueden estropear tu código son los que tú mismo sembraste en el código. Por lo tanto, intentaremos mostrarte cómo encontrar y eliminar tus errores, en otras palabras, cómo depurar tu código.

## 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).

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

¿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: ''
```

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 verdadero si el valor actual de la variable value es del tipo int.

Perdónanos si no dedicamos más tiempo a esto ahora; encontrarás explicaciones más detalladas sobre el operador `is` en un módulo del curso dedicado a la programación orientada a objetos.

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.

## El Código Python

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, no queremos que apliques la regla en tu vida diaria. No tomes el automóvil de nadie sin permiso, con la esperanza de que puedas ser tan convincente que evites la condena por lo ocurrido. La regla se trata de otra cosa.

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

«De acuerdo», puedes decir, «pero ¿cómo debo pedir perdón cuando el programa finaliza y no queda nada que más por hacer?». Aquí es donde algo llamado excepción entra en escena.

```
try:
    # Es un lugar donde
    # tu puedes hacer algo
    # sin pedir permiso.
except:
    # Es un espacio dedicado
    # exclusivamente para pedir perdón.
```

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.

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

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.

Ahora queremos hacerte una pregunta: ¿Es `ValueError` la única forma en que el control podría caer dentro del bloque `except`?

## 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.

## Dos excepciones después de un try.

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

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 se debe enfatizarse en que **solo una** de todas puede interceptar el control; **si se ejecuta una, todas las demás permanecen inactivas**. Además, la cantidad de excepciones no está limitado: 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.

Pero esta todavía no es la última palabra de Python sobre excepciones.