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

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.