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

- Modulo 4: Funciones
  - 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.
- Modulo 4: Tuplas
  - Tuplas y su propósito: construcción y uso de tuplas.
- Modulo 4: Diccionarios
  - Diccionarios y su propósito: construcción y uso de diccionarios.
- Modulo 4: Excepciones
  - Introducción a las excepciones en Python.

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.

From:
https://miguelangel.torresegea.es/wiki/ - **miguel angel torres egea**

Permanent link:
https://miguelangel.torresegea.es/wiki/info:cursos:netacad:python:pe1m4?rev=1655832316

Last update: **21/06/2022 10:25**