Actividad 02 - Tipos de datos, modularización y excepciones

Paradigmas de la Programación

Carlos Augusto Zayas Guggiari

10 de marzo de 2024

Resumen

Con la ayuda del contenido de este documento, el alumno resolverá un par de ejercicios mientras aprende acerca del sistema de tipado de datos de Python, sus tipos básicos y estructurados, y las herramientas de introspección que el lenguaje ofrece para analizar los tipos de datos.

Contenido

1 Objetivos

1. Conocer más características del lenguaje Python:
   - Sistema de tipado.
   - Tipos básicos y estructurados.
   - Introspección y reflexión.
2. Modularización de programas en Python:
   - Módulos
   - Paquetes
3. Manejo de excepciones en Python:
   - LBYL: Look Before You Leap (mirá antes de saltar)
   - EAFP: Easier to Ask Forgiveness than Permission (es más fácil pedir perdón que permiso)

2 Tipos de datos en Python

En un programa, cada dato que registramos como variable pertenece a un tipo determinado. En un lenguaje de programación, a la manera en la que se manipulan esos tipos de datos y cómo interactúan entre sí se le conoce como **sistema de tipado de datos**.

Python es un lenguaje de **tipado dinámico**, esto quiere decir que el tipo de dato de una variable determinada se asigna y verifica en tiempo de ejecución, debido a que estamos usando un lenguaje interpretado, no compilado. Esto también implica que las variables se definen mediante una simple asignación del valor que van a almacenar. No es necesario declararlas, como ocurre en los lenguajes de tipado estático, generalmente compilados, como por ejemplo C y Java.

```
a = 1 # Se crea un objeto de tipo 'int' con valor entero 1
      # y se le asigna el identificador 'a'.
```

Sin embargo, a diferencia de otros lenguajes interpretados (Javascript, por ejemplo) Python es un lenguaje de **tipado fuerte**, esto quiere decir que no se pueden combinar datos de distinto tipo sin antes convertirlos a un tipo determinado. En lenguajes de tipado débil, para evitar una excepción (error) el intérprete determina el tipo final del resultado de una combinación de datos de distintos tipos.

```
>>> a = 1
>>> b = '1'
>>> a + b
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

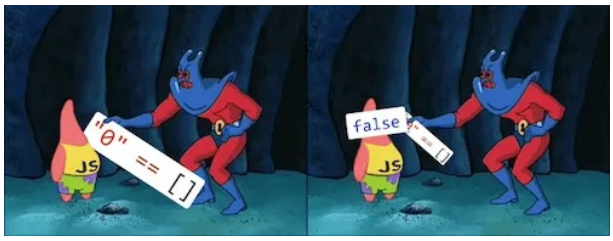
El siguiente meme expresa brillantemente algunas incoherencias del sistema de tipado de datos de Javascript:

ATENCIÓN:
No confundir el operador de *comparación* `==` con el de *asignación* `=` ya que es una fuente común de errores en el código, y a veces son difíciles de detectar.

El cero comparado con un caracter "0" da verdadero en JS.

El cero comparado con una lista vacía también da verdadero.

Entonces, si comparamos un caracter "0" y una lista vacía...



...no obtenemos el resultado lógico esperado.

PRUEBA:
Comprobemos las expresiones del meme en un intérprete Javascript (por ejemplo el incorporado en el navegador de nuestra PC, accesible mediante la tecla F12) y comparemos los resultados con los que se obtienen en Python.

Los operadores de comparación se utilizan para comparar dos valores:

Operador	Nombre	Ejemplo
==	Igual	x == y
!=	Distinto	x != y
>	Mayor	x > y
<	Menor	x < y
>=	Mayor o Igual	x >= y
<=	Menor o Igual	x <= y

Python no necesita que se declaren los tipos de las variables, pero para mejorar la legibilidad del código, permite el **type hinting** (insinuación de tipo - *hint*: pista). Su uso no es obligatorio, sino recomendado como una manera práctica de agregar documentación a nuestro código:

```
edad: int = 30
nombre: str = "Ana Torres"
inscripta: bool = True

def saludo(nombre: str) -> str:
    return "Hola, " + nombre
```

El uso de la insinuación o pista de tipo no implica que, si se le asigna a una variable un valor de tipo distinto al definido, el intérprete generará una excepción (error).

NOTA:
Si bien Python es un lenguaje **multiparadigma** (implementa nativamente los paradigmas imperativo, estructurado, modular, orientado a objetos y parcialmente funcional, pudiendo incorporar otros mediante módulos) en realidad todo en él es un objeto, incluso los datos (variables). Por ende, los tipos de datos son técnicamente **clases**.

2.1 Tipos básicos

Tipo	Clase
Numérico (escalar)	`int` , `float` , `complex`
Alfanumérico (cadena)	`str`
Binario (bytes)	`bytes` , `bytearray` , `memoryview`
Lógico (booleano)	`bool`

Un valor binario puede expresarse como una cadena de caracteres con el prefijo **b**. Ejemplo: `b'00110100'`

Un valor booleano puede ser `True` o `False` (T y F mayúsculas). Cada tipo de dato tiene implícitamente un valor lógico. Por ejemplo, el valor 0 entero da falso, mientras que cualquier otro valor (positivo o negativo) da verdadero. Una cadena vacía da falso, en cambio una cadena de al menos un caracter da verdadero.

Python incluye funciones de conversión que permiten combinar datos de distinto tipo:

Función	Retorna
`int()`	Objeto numérico entero en base a uno real (float) o cadena de dígitos.
`float()`	Objeto numérico real (float) a uno entero o cadena de dígitos.
`complex()`	Objeto numérico complejo con componentes real e imaginario.
`pow()`	Potencia de los números especificados.
`abs()`	Valor absoluto de un número sin considerar su signo.
`round()`	Redondeo de un número en base a una cantidad determinada de decimales.
`bool()`	Valor lógico de una expresión.
`str()`	Cadena de caracteres en base al valor proporcionado.

PRUEBA:
En el ambiente interactivo de Python, basándonos en los fragmentos de código de ejemplo mostrados, probemos las funciones de conversión y analicemos los resultados.

2.2 Tipos estructurados

Tipo	Clase
Secuencia	`list` , `tuple` , `range`
Mapeo	`dict`
Conjunto	`set` , `frozenset`

Los tipos de datos estructurados sirven para agrupar y organizar los datos de diferentes formas, lo que nos permite modelar de manera más eficiente los problemas y situaciones del mundo real en nuestro código. Al agrupar datos relacionados juntos, podemos encapsular la información y el comportamiento asociado en una misma estructura, el objeto, lo cual facilita su manipulación y hace el código más intuitivo y mantenible. Además, esto nos permite aprovechar características importantes de la programación orientada a objetos como la herencia, el polimorfismo y la encapsulación.

Tipos de secuencia: Incluyen listas, tuplas y rangos.

- **Listas**: Se definen con corchetes [] y pueden contener diferentes tipos de datos. Por ejemplo, `mi_lista = [1, 'dos', 3.0]`.
- **Tuplas**: Son similares a las listas, pero son inmutables. Se definen con paréntesis (). Por ejemplo, `mi_tupla = (1, 'dos', 3.0)`.
- **Rangos**: Se utilizan para representar una secuencia de números. Por ejemplo, `mi_rango = range(1, 10)` representa los números del 1 al 9.

Las listas son estructuras de datos muy versátiles que permiten almacenar varios elementos en una única variable. Este es un resumen de las principales operaciones que se pueden realizar con una lista:

- **Creación de una lista**: Se puede crear una lista colocando varios elementos entre corchetes y separándolos con comas. Por ejemplo:

```
mi_lista = [1, 'dos', 3.0]
```

- **Acceso a elementos**: Se puede acceder a un elemento de la lista utilizando su índice. Los índices en Python comienzan en 0. Por ejemplo:

```
print(mi_lista[1])  # Imprime 'dos'
```

- **Modificación de elementos**: Las listas son mutables, lo que significa que se pueden cambiar sus elementos. Por ejemplo:

```
mi_lista[1] = 'two'
print(mi_lista)  # Imprime [1, 'two', 3.0]
```

- **Agregar elementos**: Se puede agregar un elemento al final de la lista con el método `append()`. Por ejemplo:

```
mi_lista.append('cuatro')
print(mi_lista)  # Imprime [1, 'two', 3.0, 'cuatro']
```

  También se puede insertar un elemento en una posición específica con el método `insert()`. Por ejemplo:

```
mi_lista.insert(1, 'one-and-a-half')
print(mi_lista)  # Imprime [1, 'one-and-a-half', 'two', 3.0, 'cuatro']
```

- **Eliminar elementos**: Se puede eliminar un elemento con el método `remove()`, que elimina la primera aparición del valor especificado. Por ejemplo:

```
mi_lista.remove('two')
print(mi_lista)  # Imprime [1, 'one-and-a-half', 3.0, 'cuatro']
```

  También se puede eliminar un elemento en una posición específica con el método `pop()`. Si no se especifica un índice, `pop()` elimina y devuelve el último elemento de la lista. Por ejemplo:

```
elemento_eliminado = mi_lista.pop(1)
print(mi_lista)  # Imprime [1, 3.0, 'cuatro']
print(elemento_eliminado)  # Imprime 'one-and-a-half'
```

- **Ordenar elementos**: Se pueden ordenar los elementos de una lista con el método `sort()`. Por ejemplo:

```
lista_numeros = [3, 1, 4, 1, 5, 9, 2]
lista_numeros.sort()
print(lista_numeros)  # Imprime [1, 1, 2, 3, 4, 5, 9]
```

  Hay que tener en cuenta que `sort()` ordena la lista en su lugar, lo que significa que cambia la lista original. Si lo que se quiere es una versión ordenada de la lista sin cambiar la original, se puede usar la función `sorted()`.

- **Invertir elementos**: Se puede invertir el orden de los elementos de una lista con el método `reverse()`. Por ejemplo:

```
lista_numeros.reverse()
print(lista_numeros)  # Imprime [9, 5, 4, 3, 2, 1, 1]
```

Las tuplas son preferibles sobre las listas en los siguientes casos:

- Cuando se necesita una secuencia de datos que no cambiará: Las tuplas son inmutables, lo que significa que una vez definida, no puede ser modificada.
- Cuando se necesita una secuencia de datos que pueda ser utilizada como clave en un diccionario: Las listas no pueden ser usadas como claves en un diccionario debido a que son mutables, mientras que las tuplas sí pueden debido a su inmutabilidad.
- Cuando se desea optimizar el rendimiento de un programa: Las tuplas, al ser inmutables, son más eficientes en términos de rendimiento y uso de memoria en comparación con las listas.

Los rangos son útiles en varios casos:

- Cuando se necesita generar una secuencia de números, especialmente para bucles. Por ejemplo, para iterar sobre un bucle un número específico de veces.
- Cuando se desea representar una secuencia de números sin tener que crear una lista o tupla, lo que puede ahorrar memoria.
- Cuando se necesita una secuencia inmutable de números, ya que los rangos son inmutables.

Los rangos en Python son considerados como **generadores**. Los generadores son objetos iterables (iteradores) que generan los valores sobre la marcha, lo que significa que los valores no se almacenan en la memoria hasta que se necesitan. Esto puede ser beneficioso en términos de rendimiento y eficiencia de la memoria cuando se trabaja con secuencias grandes de números.

NOTA:
Un **iterable** es un objeto que se puede recorrer (iterar) con la ayuda de una estructura lógica de ciclo o bucle (por ejemplo `for`). Todos los tipos estructurados y las cadenas de caracteres (`str`) son iterables.
Sin embargo, ser iterable no implica ser **iterador**. En Python, un iterador es un objeto que representa un flujo de datos que produce un valor a la vez, en base a un protocolo que consta de los métodos `__iter__()` y `__next__()`, y mantiene un estado que recuerda dónde se encuentra durante la iteración.
Los iteradores solo pueden avanzar usando `__next__`, no pueden retroceder ni restablecerse.
La función `iter()` devuelve un iterador en base a un objeto iterable. Con la función `next()` se puede invocar el siguiente valor del iterador.

Tipos de mapeo: Los diccionarios son el único tipo de mapeo en Python.

- **Diccionarios**: Son una colección de pares clave-valor. Se definen con llaves {}. Por ejemplo, `mi_diccionario = {'nombre': 'Juan', 'edad': 25}`.

Los diccionarios en Python son una estructura de datos muy útil que permite almacenar pares de elementos bajo el concepto de clave y valor. Cada valor almacenado en un diccionario puede ser accedido utilizando su clave correspondiente, lo cual hace a los diccionarios extremadamente eficientes para ciertos tipos de operaciones.

Además de ser mutables, los diccionarios tienen algunas características únicas. Por ejemplo, a diferencia de las listas y las tuplas, los diccionarios no mantienen un orden específico para sus elementos. Esto significa que si iteras sobre los elementos de un diccionario, no hay garantía de que los obtendrás en el mismo orden en el que los agregaste.

Los diccionarios también son dinámicos, lo que significa que se pueden agregar, modificar y eliminar pares clave-valor del diccionario en cualquier momento. Para agregar un nuevo par clave-valor a un diccionario, simplemente se asigna un valor a una nueva clave, como `mi_diccionario['nueva_clave'] = 'nuevo_valor'`. Para modificar un valor existente, se asigna un nuevo valor a la clave correspondiente, como `mi_diccionario['clave_existente'] = 'nuevo_valor'`. Para eliminar un par clave-valor, se puede usar la instrucción `del`, como `del mi_diccionario['clave']`.

Otra característica útil de los diccionarios es que permiten el almacenamiento de diferentes tipos de datos, incluyendo otros diccionarios, lo que los hace extremadamente versátiles para almacenar estructuras de datos complejas.

Tipos de conjunto: Incluyen los sets y los frozensets.

- **Set**: Es una colección no ordenada de elementos únicos. Se define con llaves {}. Por ejemplo, `mi_set = {1, 2, 3, 3}` resulta en `{1, 2, 3}` ya que los elementos repetidos se eliminan.
- **Frozenset**: Es un set inmutable. Se define con la función frozenset(). Por ejemplo, `mi_frozenset = frozenset([1, 2, 3, 3])` resulta en `frozenset({1, 2, 3})`.

Los sets y los frozensets son dos tipos de colecciones en Python que son especialmente útiles cuando se necesita almacenar elementos únicos, es decir, sin duplicados.

Al igual que los diccionarios, los sets se definen con llaves {}. Sin embargo, a diferencia de los diccionarios, no almacenan pares clave-valor, sino elementos individuales. Por ejemplo, si se define un set como `mi_set = {1, 2, 3, 3}`, el resultado será `{1, 2, 3}`, ya que los elementos duplicados son automáticamente eliminados. Los sets son mutables, lo que significa que se pueden agregar, modificar y eliminar elementos después de que el set ha sido creado. Además, los sets soportan operaciones matemáticas como la unión, intersección, diferencia y diferencia simétrica.

En un frozenset, en cambio, una vez creado, sus elementos no pueden ser modificados. Se definen con la función frozenset(), que toma un iterable como argumento. Por ejemplo, si se define un frozenset como `mi_frozenset = frozenset([1, 2, 3, 3])`, el resultado será `frozenset({1, 2, 3})`. Aunque son inmutables y, por lo tanto, no pueden ser modificados después de su creación, pueden ser usados como claves en un diccionario, a diferencia de los sets.

Ambos, sets y frozensets, son útiles en situaciones en las que se necesita garantizar que los elementos son únicos y en las que se realizan operaciones que involucran conjuntos, como la comprobación de si un elemento pertenece al conjunto, la unión de varios conjuntos, la intersección de conjuntos, entre otras.

NOTA:
Tipos Mutables: Listas - Diccionarios - Sets.
Tipos Inmutables: Tuplas - Rangos - Frozensets - Todos los tipos básicos.

PRUEBA:
En el ambiente interactivo de Python, basándonos en los fragmentos de código de los ejemplos mostrados, probemos crear y manipular las distintas estructuras de datos.

2.3 Introspección y reflexión

Son dos conceptos de la programación moderna, relacionados pero distintos.

La **introspección** se refiere a la capacidad de un lenguaje de programación de examinar el tipo o los atributos de un objeto en tiempo de ejecución.

Un buen ejemplo de introspección en Python es la función `type()`, que permite determinar el tipo de un objeto. Por ejemplo:

```
>>> x = [1, 2, 3]
>>> type(x)
<class 'list'>
>>> type(x).__name__
'list'
```

Otra función introspectiva es `dir()`, que devuelve una lista de los atributos de un objeto. Por ejemplo:

```
>>> x = [1, 2, 3]
>>> print(dir(x))
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', ... ]
```

La lista resultante está compuesta por cadenas de caracteres que representan a las propiedades y métodos de la clase del objeto referenciado.

Otra función útil es `hasattr()`, que se usa para comprobar si un objeto tiene un atributo específico:

```
>>> x = [1, 2, 3]
>>> print(hasattr(x, 'append'))
True
```

En este ejemplo, `hasattr(x, 'append')` devuelve `True`, lo que significa que la lista `x` tiene un atributo `append`.

Un atributo puede ser una propiedad (un dato) o un método (una función asociada al objeto). Para determinar si el atributo en cuestión es un método, se cuenta con la función `callable()` ("llamable", invocable), que devuelve `True` en caso afirmativo. Pero dicha función requiere el atributo en sí, no una cadena de caracteres con su nombre. Por este motivo, antes se tiene que usar `getattr()` para obtener el atributo:

```
>>> atributo = getattr(x, 'append')
>>> callable(atributo)
True
```

Estas son sólo algunas de las funciones que Python proporciona para la introspección. Las funciones introspectivas son una poderosa herramienta que permite a los programadores escribir código más eficiente y depurarlo más fácilmente.

La **reflexión**, por otro lado, va un paso más allá y se refiere a la capacidad de un programa para manipular o cambiar el comportamiento y las propiedades de los objetos en tiempo de ejecución. Esto incluye, por ejemplo, crear nuevas instancias de un objeto, cambiar el valor de sus atributos, o invocar sus métodos. La reflexión es un concepto más avanzado y poderoso, pero también más peligroso si se usa incorrectamente.

En resumen, mientras que la introspección permite mirar un objeto y entender qué es, la reflexión permite cambiar lo que es. Se utiliza para manipular el código en tiempo de ejecución.

La función `repr()` devuelve una representación del objeto en formato cadena de caracteres, que puede ser usada para recrear el objeto utilizando la función `eval()`:

```
>>> x = [1, 2, 3]
>>> repr(x)
'[1, 2, 3]'

>>> y = eval(repr(x))
>>> print(y)
[1, 2, 3]
```

Se pueden cambiar los atributos de un objeto en tiempo de ejecución utilizando la función `setattr()`:

```
class Test:
    def __init__(self):
        self.x = 10

test = Test()
print(test.x)  # Salida: 10

setattr(test, 'x', 20)
print(test.x)  # Salida: 20
```

Se puede ejecutar un método de un objeto en tiempo de ejecución utilizando la función `getattr()`.

```
class Test:
    def saludo(self):
        return "¡Hola, mundo!"

test = Test()
metodo = getattr(test, 'saludo')
print(metodo())  # Salida: ¡Hola, mundo!
```

Se puede crear una nueva clase en tiempo de ejecución utilizando la función `type()`.

```
def saludo(self):
    return "¡Hola, mundo!"

Test = type('Test', (), {'saludo': saludo})

test = Test()
print(test.saludo())  # Salida: ¡Hola, mundo!
```

La introspección y la reflexión también facilitan el tratamiento de **iterables**. Un objeto es iterable si se puede obtener un iterador de él. Una forma sencilla de determinar si un objeto es iterable es comprobar si contiene el atributo `__iter__`. De ser así, podemos:

1. Llamar a la función `iter()` del objeto, que nos retornará un iterador.
2. Llamar a la función `next()` en el iterador, que nos dará el siguiente elemento.
3. Si el iterador está agotado (si no tiene más elementos), se levantará la excepción `StopIteration`.

Estas son sólo algunas formas en las que la reflexión permite al código inspeccionar y cambiar su propio comportamiento en tiempo de ejecución.

PRUEBA:
En el ambiente interactivo de Python, basándonos en los fragmentos de código de los ejemplos mostrados, probemos las herramientas de introspección y reflexión. Entre nuestras pruebas, intentemos recorrer un objeto iterable sin utilizar la estructura `for`.

3 Modularización de programas

La modularización es una técnica que facilita el mantenimiento y organización del código. En Python, esto se logra mediante la implementación de módulos y paquetes.

3.1 Módulos

Un módulo en Python es simplemente un archivo con extensión `.py` que contiene definiciones de funciones, clases y variables que podemos usar en otros programas. Técnicamente, cualquier programa escrito en Python puede considerarse un módulo, pero en la práctica hay una distinción empírica entre un módulo y un **programa principal**, que es el que arranca la aplicación e importa los módulos requeridos para que esta funcione.

Por ejemplo, podríamos tener un módulo `mis_funciones.py` con la siguiente función:

```
def saludo(nombre):
    print(f'Hola, {nombre}!')  # Cadena con prefijo 'f' (formato)
```

Desde nuestro programa principal, podemos importar este módulo y usar la función `saludo`:

```
import mis_funciones
mis_funciones.saludo('Estudiantes')
```

3.2 Paquetes

Un paquete en Python es simplemente una carpeta (directorio) que contiene varios módulos y un archivo especial llamado `__init__.py`. Este archivo puede estar vacío, pero debe estar presente para que Python reconozca el directorio como un paquete. Los paquetes nos permiten organizar módulos relacionados en una estructura de directorios (tipo "árbol").

A continuación vemos un ejemplo de cómo podría verse la estructura de directorios de un paquete para Python:

```
mi_paquete/
    __init__.py
    modulo1.py
    modulo2.py
```

```
sub_paquete/
    __init__.py
    modulo3.py
```

En este ejemplo, `mi_paquete` es un paquete que contiene dos módulos (`modulo1.py` y `modulo2.py`) y un subpaquete llamado `sub_paquete` . El subpaquete también contiene un módulo (`modulo3.py`).

El archivo `__init__.py` en cada directorio es lo que indica a Python que el directorio debe tratarse como un paquete o subpaquete. Estos archivos pueden estar vacíos o incluir código que se ejecuta cuando se importa el paquete.

Para importar un módulo desde el paquete, se podría escribir algo como esto:

```
from mi_paquete import modulo1
modulo1.mi_funcion()
```

O, para importar desde el subpaquete, se podría escribir:

```
from mi_paquete.sub_paquete import modulo3
modulo3.mi_funcion()
```

Esto es solo un ejemplo básico, los paquetes en Python pueden tener tantos subpaquetes y módulos como se necesite, lo que permite crear una estructura de directorios muy organizada para nuestro código.

3.3 El comando import

Como se vio anteriormente, el comando `import` se utiliza para incluir (importar) un módulo o paquete en nuestro programa. Existen varias maneras de utilizar este comando:

- `import nombre_del_modulo` : importa el módulo completo. Para usar una función o clase del módulo, debemos precederla con `nombre_del_modulo.` .
- `from nombre_del_modulo import nombre_de_la_funcion` : importa solo la función especificada del módulo. Podemos usar la función directamente sin precederla con el nombre del módulo.
- `import nombre_del_modulo as otro_nombre` : importa el módulo con un alias. Podemos usar el alias para referirnos al módulo en nuestro código.

4 Manejo de excepciones

En programación, las excepciones son errores que pueden ocurrir durante la ejecución de un programa. Cuando se produce un error que el programador no previó, el programa no sabe cómo manejarlo y en consecuencia se genera una excepción que interrumpe el flujo normal del programa.

La manera tradicional de prevenir errores es mediante verificaciones previas basadas en el uso de estructuras condicionales. En Python, estas excepciones también pueden ser capturadas y manejadas mediante el uso de bloques de código `try` / `except` , permitiendo así que el programa continúe ejecutándose a pesar de los errores.

Existen dos actitudes (o filosofías, o estilos de codificación) frente al manejo de excepciones. Se denominan, como muchas cosas en informática, por sus propias siglas en inglés: **LBYL** (*Look Before You Leap*, "mira antes de saltar") y **EAFP** (*Easier to Ask Forgiveness than Permission*, "es más fácil pedir perdón que permiso").

4.1 LBYL: Hay que fijarse antes de saltar

Una manera de evitar errores sería verificar primero si algo tendrá éxito, y continuar sólo si se sabe que funcionará. De esto trata lo de mirar antes de saltar. Este estilo de codificación prueba explícitamente las condiciones previas antes de realizar llamadas o búsquedas. Se caracteriza por la presencia de muchas declaraciones `if` , comúnmente anidadas.

Para comprender la esencia de LBYL, un ejemplo clásico es el de manejo de claves faltantes en un diccionario. Supongamos que necesitamos procesar el diccionario clave por clave. Sabemos de antemano que el diccionario puede incluir algunas claves específicas, pero también es posible que algunas no estén presentes. ¿Cómo podemos lidiar con las que faltan sin obtener una excepción `KeyError` que rompa nuestro código?

Esta sería la manera de resolver el problema usando una sentencia condicional:

```
if "clave_posible" in mi_diccionario:
    valor = mi_diccionario["clave_posible"]
else:
    # Acá se maneja la posibilidad de que no exista la clave...
```

Esta forma de abordar el problema se conoce como "mirar antes de saltar" porque se basa en verificar la condición previa antes de realizar la acción deseada. LBYL es un estilo de programación tradicional en el que el programador se asegura de que un fragmento de código funcione antes de ejecutarlo. Si se sigue este estilo, se terminará con muchas declaraciones `if` en todo el código.

4.2 EAFP: Mejor disculparse que pedir permiso

> "Recuerda siempre que es mucho más fácil disculparse que obtener permiso. En este mundo de las computadoras, lo mejor que se puede hacer es hacerlo."
> – *Contraalmirante Grace Murray Hopper, científica de la computación y pionera de la programación*

La Dra. Grace no se refería específicamente a la programación cuando dijo esa frase, pero la técnica EAFP es una expresión concreta de ese consejo aplicado a la escritura de código. Sugiere que de inmediato se debe hacer lo que se espera que funcione. Si no funciona y ocurre una excepción, simplemente hay que detectarla y manejarla adecuadamente.

Este estilo de codificación supone la existencia de claves o atributos válidos y detecta excepciones si la suposición resulta falsa. Este estilo limpio y rápido se caracteriza por la presencia de muchas declaraciones de prueba (`try`) y excepción (`except`).

Así se reescribiría al estilo EAFP el ejemplo de la sección anterior:

```
try:
    valor = mi_diccionario["clave_posible"]
except KeyError:
    # Acá se maneja la posibilidad de que no exista la clave...
```

En esta variación, no se verifica si la clave está presente antes de usarla. En su lugar, se intenta acceder a la clave deseada. Si por alguna razón la clave no está presente, simplemente se detecta el `KeyError` en la cláusula `except` y se lo maneja adecuadamente.

Cualquiera de los estilos puede ser la solución adecuada según el problema específico. Algo que puede ayudar a decidir qué estilo usar es responder a la pregunta: ¿Qué es más conveniente en esta situación, evitar que ocurran errores o manejarlos después de que ocurran?

5 Ejercicios

1. Escriba una función que reciba como parámetro un objeto y retorne dos listas: una con las propiedades y otra con los métodos.

```
# Ejemplo de llamada a la función "filtrar" (puede usarse otro nombre)
propiedades, metodos = filtrar(objeto)
```

2. Escriba una función que reciba como parámetro un objeto, determine el tipo de dato y en base a las tablas "Tipo/Clase" mostradas anteriormente devuelva la cadena correspondiente a la columna "Tipo".

```
# Ejemplo de llamada a la función "tipo" (puede llamarse de otra manera)
nombre = tipo("Carlos") # "nombre" debe contener "Alfanumérico (cadena)"
```

3. Guarde las funciones en un archivo llamado `actividad02.py` y envíelo por correo a czayas@pol.una.py, poniendo como asunto "Actividad 02".

NOTA:
Utilice un **analizador de código** durante la elaboración de los ejercicios. Es una herramienta que analiza el código fuente sin ejecutarlo, con el fin de identificar prácticas de codificación deficientes. Ofrece retroalimentación a los programadores durante la fase de desarrollo, incluso sobre fallas de seguridad que podrían introducirse en el código.
Hay entornos de desarrollo que integran el análisis de código al editor de programación, funcionando de manera similar a las facilidades de autocompletado y resaltado de sintaxis. Uno de los más conocidos para Python es PyCharm que es un ambiente sumamente completo y profesional, pero existen alternativas más sencillas que consumen menos recursos.

6 Recursos

Los siguientes recursos están disponibles online de manera gratuita:

6.1 Documentación

- El Tutorial de Python 3 - PDF
- Aprenda a Pensar como un Programador con Python - PDF
- Inmersión en Python 3 - PDF
- PEP 8: Guía de estilo para el código Python - PDF

6.2 Artículos

- El Pythonista: Guía de estilos en Python - Link
- El Pythonista: Entornos de desarrollo para Python - Link

6.3 Software

- Mu: Mini entorno de desarrollo para Python - Link
- Mu: Versión portable para Windows - Link
- Thonny: Entorno de desarrollo con herramientas didácticas - Link
- Thonny: Versión portable para Windows - Link

NOTA:
Los enlaces fueron verificados a la fecha de revisión de este documento. Sin embargo, esto no garantiza su validez al momento de leer esto. Favor enviar sugerencias y correcciones a czayas@pol.una.py