

Introducción a los diccionarios

| | |
|--|----|
| ¿Qué aprenderás? | 3 |
| Introducción | 3 |
| Diccionarios | 4 |
| Lista versus Diccionario | 4 |
| ¿Para qué sirven los diccionarios? | 5 |
| Crear un diccionario | 5 |
| Acceder a un elemento dentro de un diccionario | 6 |
| La clave tiene que ser única | 6 |
| Agregando un elemento a un diccionario | 6 |
| Cambiano un elemento dentro de un diccionario | 7 |
| Eliminar elementos de un diccionario | 7 |
| Unir diccionarios | 8 |
| Cuidado con las colisiones | 8 |
| Otros Métodos para diccionarios | 9 |
| Método keys() | 9 |
| Método values() | 9 |
| Método items() | 10 |
| Método get() | 10 |
| Otras Estructuras de Datos | 10 |
| Tuplas: | 10 |
| Sets: | 11 |
| Convertir estructuras | 12 |
| Convertir un diccionario en una lista | 12 |
| Convertir una lista en un diccionario | 12 |
| Ejercicio guiado 1: | 13 |
| Efemérides | 13 |
| La función dir() | 18 |
| Otras funciones en Python | 20 |



¡Comencemos!

¿Qué aprenderás?

- Codificar un programa utilizando agregación, modificación y eliminación de elementos de una estructura de datos acorde al lenguaje Python para resolver un problema.

Introducción

Si bien las listas en Python pueden aplicarse a la resolución de diversos problemas, tienen asociado el inconveniente de que no se pueden nombrar sus componentes. Para solucionar este inconveniente y presentar más alternativas es que introduciremos los diccionarios.

Diccionarios

Los diccionarios son una estructura de datos compuesta por pares de `clave:valor`, donde cada clave se asocia con un elemento del diccionario, donde como toda estructura de datos, permiten almacenar una gran cantidad de datos en una sola variable.

Reciben este nombre porque se leen igual que uno de la vida real, ya que en un diccionario buscamos una palabra y esta nos lleva a la definición: la clave es equivalente a la palabra y el valor es equivalente a su definición.

Lista versus Diccionario

Las listas y los diccionarios son estructuras bastante similares, con la diferencia de que para acceder a los elementos de una lista, se hace a través de la posición o el índice asociado al elemento, mientras que en un diccionario, se hace por medio de la **clave** (o llave).

La otra diferencia es que en una lista, los índices se generan automáticamente para cada elemento, mientras que las claves de un diccionario no se generan automáticamente, sino que son definidas explícitamente al crear el diccionario o al asignar un nuevo elemento a la estructura.

Una última diferencia, es que la lista es un elemento ordenado donde el orden lo entrega el índice, en cambio, en un diccionario es un elemento no ordenado, no hay un elemento que va primero, segundo o último. La única manera de acceder a dicho elemento es mediante su clave.

Por último, las claves del diccionario pueden ser un string, un número, o incluso un booleano, pero con frecuencia se utilizan los strings.

```
# En una lista usamos la posición para acceder a un elemento, y los
índices se generan en forma implícita
lista = [25, 31, "hola"]
lista[2] # "hola"

# En un diccionario usamos la clave, y se deben definir de forma
explícita
diccionario = {"a": 25, "b": 31, "c": "hola"}
diccionario["c"] # "hola"
```

¿Para qué sirven los diccionarios?

Existen varios tipos de situaciones donde los diccionarios son mejores que las listas para resolver un determinado problema, principalmente cuando se requiere identificar rápidamente un elemento a través de un valor único, la **clave** y para generar conversiones.

Crear un diccionario

En Python, un diccionario (vacío) se define con llaves: `{}`. Cada par de clave y valor se asocia mediante `:`, en la forma `clave: valor`.

```
notas = {"Camila": 7, "Antonio": 5, "Felipe": 6, "Antonia": 7}
```

Si al momento de definir un diccionario tenemos muchas llaves, podemos definirlo en múltiples líneas para facilitar la lectura de código.

```
notas = {  
    "Camila": 7,  
    "Antonio": 5,  
    "Felipe": 6,  
    "Daniela": 5,  
    "Vicente": 7,  
}
```

Acceder a un elemento dentro de un diccionario

Se accede a un elemento específico utilizando la **clave** de ese **valor**, donde necesariamente se debe utilizar la misma clave en la que está almacenado el elemento, sino podríamos recibir un valor no deseado, o incluso tener un error si la clave no existe.

```
notas["Felipe"] # 6  
notas["felipe"] # KeyError: 'felipe'
```

La clave tiene que ser única

En un diccionario, solo puede haber un valor asociado a una clave.

```
duplicados = {"clave": 1, "clave": 2}  
print(duplicados) # {"clave": 2}
```

Al usar dos veces la misma clave, Python se quedará con la última que definimos, ignorando completamente la existencia del primer valor.

Agregando un elemento a un diccionario

Tomemos un diccionario simple como el siguiente:

```
diccionario = {"llave 1": 5}
```

Para agregar un elemento a un diccionario, hace falta especificar una clave nueva, de forma que el nuevo valor ingresado sea identificado con dicha clave.

```
diccionario["llave 2"] = 9  
print(diccionario) # {"llave 1": 5, "llave 2": 9}
```



Nota: Presta atención a la sintaxis, donde para definir un diccionario usamos llaves (`{}`), pero para acceder a sus elementos usamos corchetes (`[]`).

Cambiando un elemento dentro de un diccionario

De forma similar a como agregamos un elemento a un diccionario, podemos actualizar el valor de uno. Para esto, simplemente tenemos que redefinir el valor asociado a la clave.

```
diccionario = {"llave 1": 5, "llave 2": 7}
diccionario["llave 2"] = 9
print(diccionario) # {"llave 1": 5, "llave 2": 9}
```

Eliminar elementos de un diccionario

Podemos eliminar una llave de un diccionario, junto a su valor, de dos formas: usando el método `pop` del diccionario o utilizando `del`. La principal diferencia entre ambas formas es que al utilizar `pop` obtendremos el valor del elemento eliminado.

```
diccionario = {"celular": 140000, "notebook": 489990, "tablet": 120000,
               "cargador": 12400}
```

```
del diccionario["celular"]
print(diccionario)
```

```
{'notebook': 489990, 'tablet': 120000, 'cargador': 12400}
```

```
# para usar pop se usa la clave, no la posición
eliminado = diccionario.pop("tablet")
```

```
print(eliminado)
print(diccionario)
```

```
120000
{'notebook': 489990, 'cargador': 12400}
```

Unir diccionarios

Una operación muy común es unir dos diccionarios, lo que se puede lograr a través de la siguiente expresión:

```
diccionario_a = {"nombre": "Alejandra", "apellido": "López", "edad": 33,
"altura": 1.55}
diccionario_b = { "mascota":"miti", "ejercicio":"bicicleta"}

# Union de diccionario_a y diccionario_b
diccionario_a.update(diccionario_b)

# Notar que la unión queda en el primer diccionario
print(diccionario_a)
```

```
{'nombre': 'Alejandra', 'apellido': 'López', 'edad': 33, 'altura': 1.55,
'mascota': 'miti', 'ejercicio': 'bicicleta'}
```

Cuidado con las colisiones

Cuando ambos diccionarios tienen una clave en común, el valor del segundo diccionario sobrescribe al del primero.

```
diccionario_a = {"nombre": "Alejandra", "apellido": "López", "edad": 33,
"altura": 1.55}
diccionario_b = { "mascota":"miti", "ejercicio":"bicicleta", "altura":
155}

# Union de diccionario_a y diccionario_b
diccionario_a.update(diccionario_b)

# Se sobrescribió el valor de altura por el del diccionario_b
print(diccionario_a)
```

```
{'nombre': 'Alejandra', 'apellido': 'López', 'edad': 33, 'altura': 155,
'mascota': 'miti', 'ejercicio': 'bicicleta'}
```

Por lo tanto, no es lo mismo hacer `diccionario_a.update(diccionario_b)` que `diccionario_b.update(diccionario_a)` (ver el valor asociado a la clave "edad"):

```
diccionario_a = {"nombre": "Alejandra", "apellido": "López", "edad": 33,
```



```
"altura": 1.55}
diccionario_b = { "mascota":"miti", "ejercicio":"bicicleta", "altura":
155}
diccionario_b.update(diccionario_a)
print(diccionario_b)
```

```
{'mascota': 'miti', 'ejercicio': 'bicicleta', 'altura': 1.55, 'nombre':
'Alejandra', 'apellido': 'López', 'edad': 33}
```

Otros Métodos para diccionarios

Cuando se tienen diccionarios muy grandes, hay 3 métodos que permiten poder explorarlos de manera mucho más sencilla y rápida:

```
computador = {'notebook': 489990, 'tablet': 120000, 'cargador': 12400}
```

- Método `keys()`

El método `.keys()` entregará una lista con todas las claves de un diccionario:

```
computador.keys()
```

```
dict_keys(['notebook', 'tablet', 'cargador'])
```

- Método `values()`

El método `.values()` entregará una lista con todos los valores de un diccionario:

```
computador.values()
```

```
dict_values([489990, 120000, 12400])
```

- Método `items()`

El método `.items()` entregará una lista con los pares clave-valor de un diccionario:

```
computador.items()
```

```
dict_items([('notebook', 489990), ('tablet', 120000), ('cargador',  
12400)])
```

- Método `get()`

El método `.get()` permitirá entregar un mensaje alternativo en caso de no encontrar alguna clave.

```
computador.get('notebook', 'No se encuentra el elemento solicitado')
```

```
'No se encuentra el elemento solicitado'
```

Otras Estructuras de Datos

En Python también existen las `tuplas` y los `sets` que son otras estructuras de datos no tan populares como las listas o los diccionarios. A pesar de no ser tan utilizadas, es bueno entender algunas propiedades que pueden ser útiles para algunos problemas específicos.

Tuplas:

Una tupla es un par ordenado inmutable, es decir, no se pueden modificar partes de ella, en caso de querer actualizarlo se debe modificar la tupla completa. Esta propiedad la hace poco amigable, pero aún así hay algunas funcionalidades útiles:

```
# definición una tupla  
tupla_ej = ("Abril", 2021)  
type(tupla_ej)
```

```
tuple
```

Una propiedad útil es lo que se llama unpacking o desempaquetamiento:

```
# cada valor de la tupla se asigna a month y year respectivamente
month, year = tupla_ej
```

```
# este es una manera alternativa de hacerlo
month, year = "Abril", 2021
```

Sets:

Es una estructura de datos que permite trabajar similar a lo que es la teoría de conjuntos. Una característica importante de este tipo de estructura es que no permite valores duplicados, por lo que si se necesita conocer sólo valores únicos, esta es la estructura de datos a utilizar.



NOTA: Esta estructura utiliza llaves `{}` al igual que los diccionarios. La principal diferencia entre los diccionarios y los sets, es que los sets solo tienen valores, no tienen claves.

```
# definición de un set
# se pueden ver que existen valores repetidos
muchos_animales = {'Gato', 'Perro', 'Tortuga',
                  'Gato', 'Perro', 'Tortuga',
                  'Gato', 'Perro', 'Tortuga',
                  'Gato', 'Perro', 'Tortuga',
                  'Hurón', 'Hamster', 'Erizo de Tierra'}
```

```
# no hay duplicados, sólo valores únicos
print(muchos_animales)
```

```
{'Erizo de Tierra', 'Gato', 'Hamster', 'Hurón', 'Perro', 'Tortuga'}
```

Este tipo de estructura suele ser bastante útil en análisis de texto, para conocer las palabras únicas que existen en él.

Convertir estructuras

Muchas veces será necesario hacer transformaciones de una estructura a otra por comodidad o porque ofrece propiedades más apropiadas al problema a resolver.

Acá algunos ejemplos:

Convertir un diccionario en una lista

Para lograr esto, se debe utilizar la función `items()`. Cada par (clave, valor) será una tupla:

```
list({"k1": 5, "k2": 7}.items()) # [('k1', 5), ('k2', 7)]
```

Convertir una lista en un diccionario

Para invertir la transformación se utiliza la función `dict`.

```
dict([('k1', 5), ('k2', 7)]) # {"k1": 5, "k2": 7}
```

Análogamente existen las funciones `tuple()` y `set()` que permitirán transformar a tuplas y/o sets respectivamente.

Ejercicio guiado 1:

Efemérides

Toda fecha tiene sucesos importantes ¿cómo recordarlos? Las efemérides son conjunto de acontecimientos importantes ocurridos en una misma fecha, por lo que aquí intentaremos utilizar lo aprendido para almacenar eventos importantes y consultarlos.

Supongamos los siguientes eventos:

- 1 de Enero: Año Nuevo
- 27 de Febrero: Terremoto en Chile
- 8 de Marzo: Día de la mujer
- 21 de Mayo: Glorias Navales
- 18 de Septiembre: Fiestas Patrias
- 19 de Septiembre: Glorias del Ejército
- 25 de Diciembre: Navidad

1. Crear el archivo `efemerides.py`

2. Disponibilizar estos elementos en Python

Notamos que queremos almacenar varios elementos en Python, naturalmente esto nos indica que requerimos una estructura de datos, ¿pero cuál?

Cualquiera sirve para almacenar pero los requerimientos piden también consultarlos.

Las listas y las tuplas requieren índices por lo que no serían lo más apropiado, por lo que la opción que nos queda son los Diccionarios:

```
# definimos el diccionario
efemerides = {'1 de Enero': 'Año Nuevo',
              '27 de Febrero': 'Terremoto en Chile',
              '8 de Marzo': 'Día de la Mujer',
              '21 de Mayo': 'Glorias Navales',
              '18 de Septiembre': 'Fiestas Patrias',
              '19 de Septiembre': 'Glorias del Ejercito',
              '25 de Diciembre': 'Navidad'}
```

3. Consultar la Fecha

Mediante `input()` se puede solicitar al usuario que ingrese la fecha a consultar.

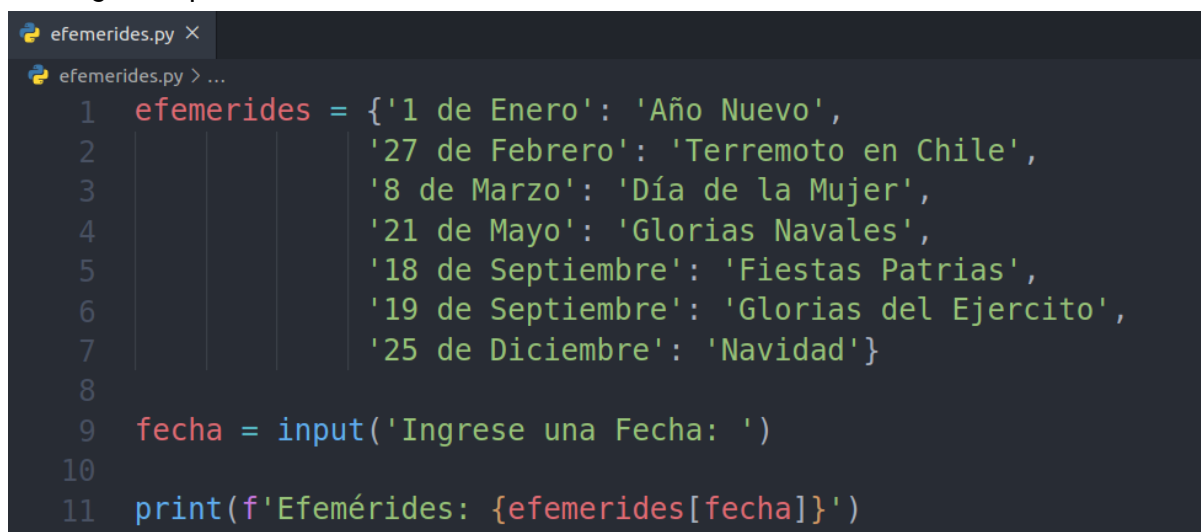
```
fecha = input('Ingrese una Fecha:')
```

4. Mostrar los resultados

Para mostrar los resultados es necesario identificar la clave del diccionario, que en este caso particular es la fecha. Combinando esto con f-strings el código queda así:

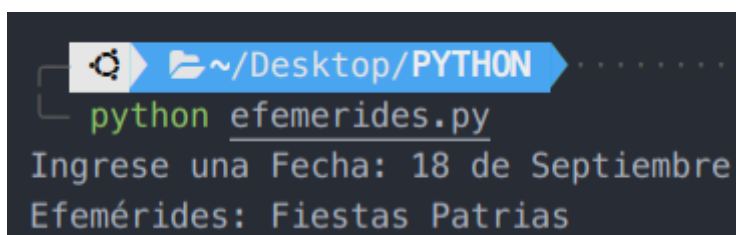
```
print(f'Efemérides: {efemerides[fecha]}')
```

El código completo se ve así:



```
efemerides.py X
efemerides.py > ...
1 efemerides = {'1 de Enero': 'Año Nuevo',
2               '27 de Febrero': 'Terremoto en Chile',
3               '8 de Marzo': 'Día de la Mujer',
4               '21 de Mayo': 'Glorias Navales',
5               '18 de Septiembre': 'Fiestas Patrias',
6               '19 de Septiembre': 'Glorias del Ejercito',
7               '25 de Diciembre': 'Navidad'}
8
9 fecha = input('Ingrese una Fecha: ')
10
11 print(f'Efemérides: {efemerides[fecha]}')
```

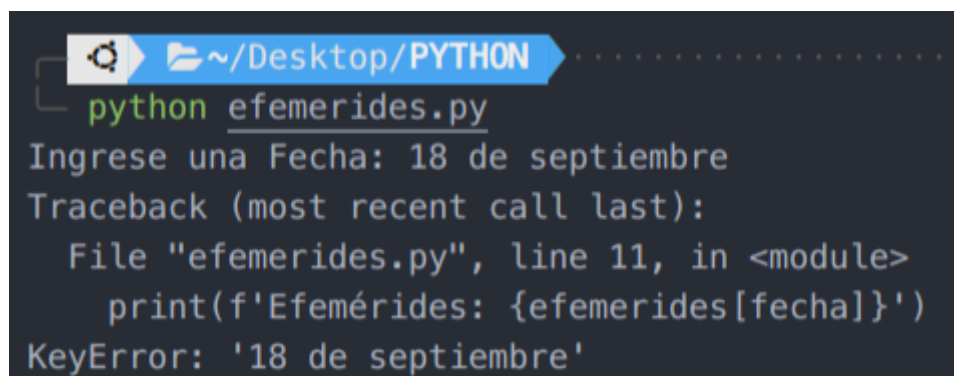
Imagen 4. Vista completa de efemerides.py
Fuente: Desafío Latam.



```
~/Desktop/PYTHON
python efemerides.py
Ingrese una Fecha: 18 de Septiembre
Efemérides: Fiestas Patrias
```

Imagen 5. Ejecución de efemerides.py
Fuente: Desafío Latam.

Pero vamos más allá. Si bien este programa funciona tiene algunos inconvenientes. Por ejemplo:

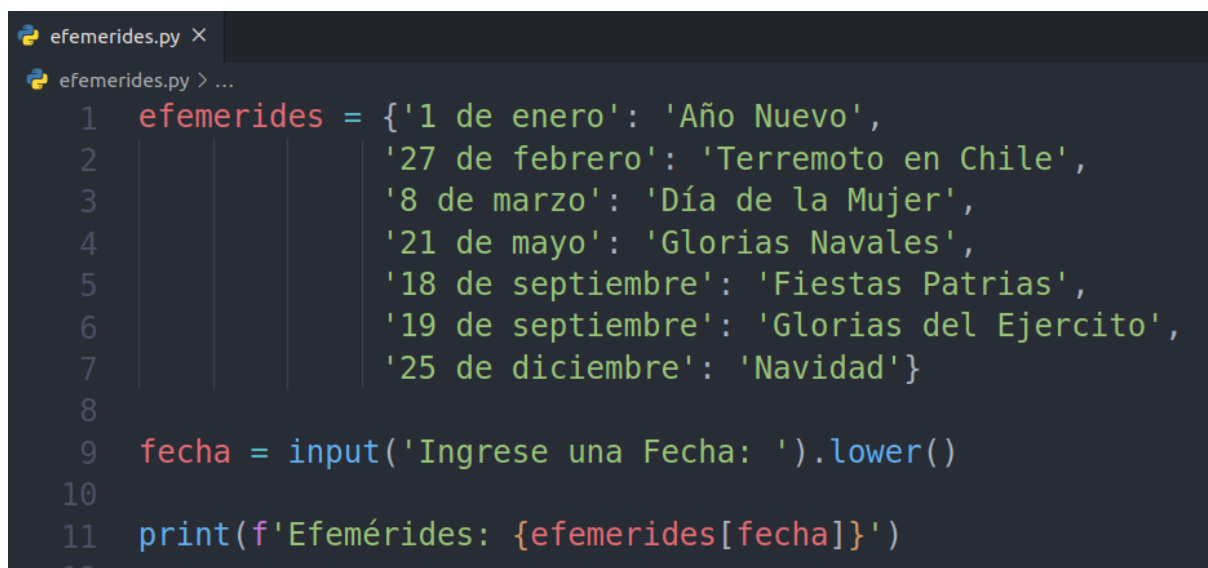


```
python efemerides.py
Ingresa una Fecha: 18 de septiembre
Traceback (most recent call last):
  File "efemerides.py", line 11, in <module>
    print(f'Efemérides: {efemerides[fecha]}')
KeyError: '18 de septiembre'
```

Imagen 6. Detectando algunos problemas en Vista completa de efemerides.py
Fuente: Desafío Latam.

Si ingresamos “18 de septiembre” (notar la minúscula), el programa falla. ¿Qué estrategia podemos aplicar para evitar este problema?

Podríamos definir todas las claves en minúsculas y aplicar `.lower()` a `input()` así:

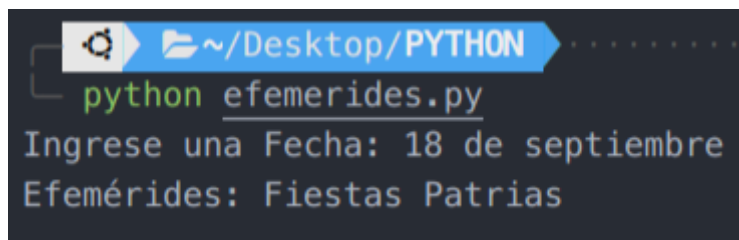


```
efemerides.py X
efemerides.py > ...
1 efemerides = {'1 de enero': 'Año Nuevo',
2               '27 de febrero': 'Terremoto en Chile',
3               '8 de marzo': 'Día de la Mujer',
4               '21 de mayo': 'Glorias Navales',
5               '18 de septiembre': 'Fiestas Patrias',
6               '19 de septiembre': 'Glorias del Ejercito',
7               '25 de diciembre': 'Navidad'}
8
9 fecha = input('Ingresa una Fecha: ').lower()
10
11 print(f'Efemérides: {efemerides[fecha]}')
```

Imagen 7. Vista completa de efemerides.py
luego de solucionar el problema.
Fuente: Desafío Latam.



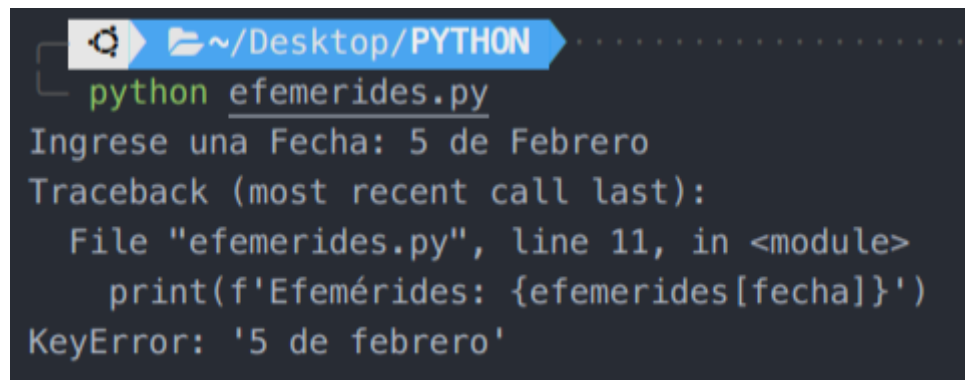
NOTA: Todas las fechas ahora están en minúsculas por lo que se soluciona el problema antes mostrado:



```
~/Desktop/PYTHON  
python efemerides.py  
Ingresa una Fecha: 18 de septiembre  
Efemérides: Fiestas Patrias
```

Imagen 8. Ejecutando el programa luego de la corrección.
Fuente: Desafío Latam.

Ahora bien, vamos aún más allá. Siempre existe la posibilidad de que un usuario ingrese información no disponible en nuestro programa, como por ejemplo:



```
~/Desktop/PYTHON  
python efemerides.py  
Ingresa una Fecha: 5 de Febrero  
Traceback (most recent call last):  
  File "efemerides.py", line 11, in <module>  
    print(f'Efemérides: {efemerides[fecha]}')  
KeyError: '5 de febrero'
```

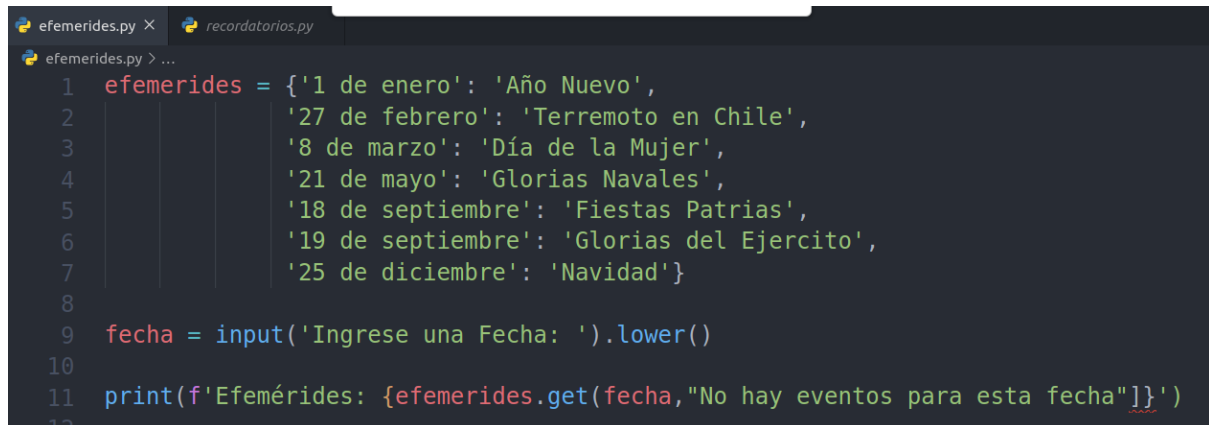
Imagen 9. Detectando oportunidades de mejora
Fuente: Desafío Latam.

Al ingresar una fecha que no tiene información nuestro programa fallará, ya que no es capaz de encontrar la clave especificada, para evitar aquello podemos utilizar `.get()`. Lo bueno que tiene este método, es que permite añadir una salida alternativa en caso de no encontrar un elemento.

```
# en caso de no encontrar la fecha solicitada devolverá el texto  
indicado  
print(f'Efemérides: {efemerides.get(fecha,"No hay eventos para esta  
fecha")}')')
```



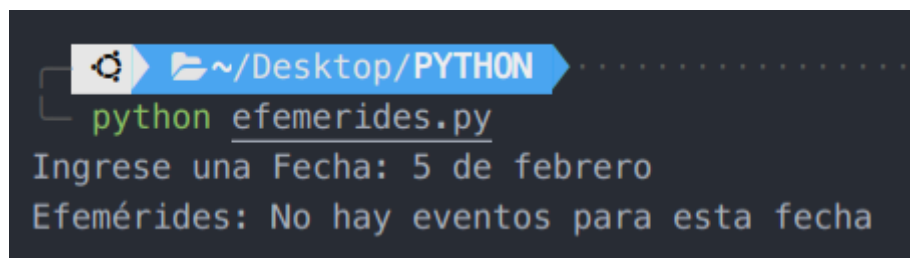
NOTA: Fijarse en el uso de los dos tipos de comillas. Esto, porque la comilla simple (') genera el **f-string** mostrado en pantalla por `print()`, mientras que la comilla doble (") genera el string alternativo en caso de no encontrar una clave. Considera que, de no respetar esto, Python arrojará error.



```
efemerides.py x recordatorios.py
efemerides.py > ...
1 efemerides = {'1 de enero': 'Año Nuevo',
2               '27 de febrero': 'Terremoto en Chile',
3               '8 de marzo': 'Día de la Mujer',
4               '21 de mayo': 'Glorias Navales',
5               '18 de septiembre': 'Fiestas Patrias',
6               '19 de septiembre': 'Glorias del Ejercito',
7               '25 de diciembre': 'Navidad'}
8
9 fecha = input('Ingrese una Fecha: ').lower()
10
11 print(f'Efemérides: {efemerides.get(fecha, "No hay eventos para esta fecha")}')
12
```

Imagen 10. Vista completa de efemerides.py
luego de agregar el método get().

Fuente: Desafío Latam.



```
~/Desktop/PYTHON
python efemerides.py
Ingrese una Fecha: 5 de febrero
Efemérides: No hay eventos para esta fecha
```

Imagen 11. Ejecutando el programa sin error.

Fuente: Desafío Latam.

La función dir()

La función dir corresponde a una función especial en Python que permitirá determinar qué métodos están disponibles en cada tipo de dato y/o estructura. Esta función puede ser de utilidad cuando olvidemos algunas de las funcionalidades que cada tipo de objeto ofrece en Python.

```
var = 'string'  
dir(var)
```

```
['__add__',  
 '__class__',  
 '__contains__',  
 '__delattr__',  
 '__dir__',  
 '__doc__',  
 '__eq__',  
 '__format__',  
 '__ge__',  
 '__getattr__',  
 '__getitem__',  
 '__getnewargs__',  
 '__gt__',  
 '__hash__',  
 '__init__',  
 '__init_subclass__',  
 '__iter__',  
 '__le__',  
 '__len__',  
 '__lt__',  
 '__mod__',  
 '__mul__',  
 '__ne__',  
 '__new__',  
 '__reduce__',  
 '__reduce_ex__',  
 '__repr__',  
 '__rmod__',  
 '__rmul__',  
 '__setattr__',  
 '__sizeof__',  
 '__str__',
```

```
'__subclasshook__',  
'capitalize',  
'casefold',  
'center',  
'count',  
'encode',  
'endswith',  
'expandtabs',  
'find',  
'format',  
'format_map',  
'index',  
'isalnum',  
'isalpha',  
'isascii',  
'isdecimal',  
'isdigit',  
'isidentifier',  
'islower',  
'isnumeric',  
'isprintable',  
'isspace',  
'istitle',  
'isupper',  
'join',  
'ljust',  
'lower',  
'lstrip',  
'maketrans',  
'partition',  
'replace',  
'rfind',  
'rindex',  
'rjust',  
'rpartition',  
'rsplit',  
'rstrip',  
'split',  
'splitlines',  
'startswith',  
'strip',  
'swapcase',  
'title',  
'translate',
```

```
'upper',  
'zfill']
```



NOTA: Para este caso hemos utilizado la función `dir()` con un elemento que sabemos es de tipo string. Cabe destacar que aparecen todos y cada uno de los métodos que esta función como por ejemplo `join()`, `count()`, `lower()`, `upper()`, etc. Obviamente aprender todos los métodos escapa del alcance del curso pero, en caso de querer profundizar, es posible encontrar todo esto en la documentación oficial de Python.

Otras funciones en Python

Existen además un par de funciones especiales que pueden ser útiles cuando se trata con estructuras de datos. Estas funciones pueden aplicarse a cualquier tipo de estructura de datos en Python. Es más, existe una que ya hemos revisado, que es la función `len()`. Esta función revisa todos los elementos de una estructura y nos indica cuántos elementos existen.

Otras funciones que pueden resultar útiles son:

`sum()`

```
var = [1,2,3]  
print(sum(var))
```

6

`max()`

```
var = [1,2,3]  
print(max(var))
```

3

min()

```
var = [1,2,3]  
print(min(var))
```

1