

Universidad Nacional de la Patagonia San Juan Bosco

Laboratorio de Investigación en Informática



Conceptos Avanzados de Programación Python

Noviembre 2023

Parte 2

Índice

1. Diccionarios	2
2. Algunos métodos y funciones con diccionarios	8
3. Conjuntos	9
4. Funciones aplicables a conjuntos	11
5. Ordenamiento de secuencias	12
6. Archivos	14
7. El cursor (los archivos no son estructuras!)	15
8. Archivos y estructuras secuenciales	17
9. Cerrar un archivo	19
10. Escribiendo en archivos	20

1. Diccionarios

El diccionario es una estructura de datos mutable muy práctica, que consiste en una *colección* (es decir, una “bolsa”) de *pares clave-valor*. Esta estructura no solo es muy útil a la hora de simplificar el desarrollo de algoritmos, sino que también es el formato genérico que se utiliza casi hegemónicamente para intercambiar información entre aplicaciones (por ejemplo en el formato JSON). Para definirlo junto con los miembros que va a contener, se encierra el listado de valores entre llaves, las parejas de clave y valor se separan con comas, y la clave y el valor se separan con ‘:’

```
>>> punto = {'x': 2, 'y': 1, 'z': 4}
>>> punto
{'x': 2, 'y': 1, 'z': 4}
```

Las claves tienen que ser valores inmutables (números, strings, tuplas), pero los valores pueden ser de cualquier tipo (inclusive diccionarios). Se puede crear un diccionario vacío con el “conjunto vacío” `{}` o bien con la función `dict()`. Una vez creado se le pueden asignar valores directamente a cada índice.

```
materias = {} # o bien materias = dict()
materias["lunes"] = [6103, 7540]
materias["martes"] = [6201]
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = [6201]
```

Para acceder al valor asociado a una determinada clave, se lo hace de la misma forma que con las listas, pero utilizando la clave elegida en lugar del índice.

```
>>> materias["lunes"]
[6103, 7540]
```

El acceso por clave falla si se provee una clave que no está en el diccionario:

```
>>> materias["domingo"]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'domingo'
```

El método `get()` si se utiliza sobre una clave inexistente produce el valor nulo.

```
>>> materias.get("domingo")
>>> # no hay tal clave: no hace nada
```

Es posible asignar una cantidad arbitraria de pares, con la única condición que las claves sean inmutables y que no se repitan.

```
>>> dicc={'número': 2, 'string': 'abc', 'tupla': (1,2,3),
'lista': ['a', 'b'], 'diccionario': {'x': 2, 'y': 3}}
>>> dicc
{'número': 2, 'string': 'abc', 'tupla': (1, 2, 3), 'lista':
['a', 'b'], 'diccionario': {'x': 2, 'y': 3}}
>>> dicc['diccionario']
{'x': 2, 'y': 3}
```

Si asignamos un valor a una clave que ya existe, el valor anterior se pierde.

```
dicc['string'] = 'def'
>>> dicc
{'número': 2, 'string': 'def', 'tupla': (1, 2, 3), 'lista':
['a', 'b'], 'diccionario': {'x': 2, 'y': 3}}
```

Si queremos modificar el contenido de algún valor de alguna clave, lo hacemos como con cualquier objeto mutable.

```
>>> dicc['string'] = dicc['string']+'hij'
>>> dicc
{'número': 2, 'string': 'defhij', 'tupla': (1, 2, 3),
'lista': ['a', 'b'], 'diccionario': {'x': 2, 'y': 3}}
```

Existen diversas formas de recorrer un diccionario. Es posible recorrer sus claves y usar esas claves para acceder a los valores.

```

for dia in materias :
    print("El {} tengo que cursar {}".format(dia, materias[dia]))
...
El lunes tengo que cursar [6103, 7540]
El martes tengo que cursar [6201]
El miércoles tengo que cursar [6103, 7540]
El jueves tengo que cursar []
El viernes tengo que cursar [6201]
>>>

```

Es posible, también, obtener los valores como tuplas donde el primer elemento es la clave y el segundo el valor.

```

for dia, codigos in materias.items() :
    print("El {} tengo que cursar {}".format(dia, codigos))
...
El lunes tengo que cursar [6103, 7540] etc...

```

El algoritmo que usa Python internamente para buscar un elemento en un diccionario es distinto al que utiliza para buscar en listas. Para buscar en las listas, se utiliza un algoritmo de comparación que tarda más a medida que la lista se hace más larga.

En cambio, para buscar en diccionarios se utiliza un modelo algorítmico llamado *hashing*, que se basa en asociar la clave del elemento con su lugar de memoria mediante una ecuación matemática. Esto tiene una propiedad muy importante, y es que sin importar cuántos elementos tenga el diccionario, el tiempo de búsqueda de los datos asociados a una clave es siempre aproximadamente igual.

Este algoritmo de hashing es la razón por la cual las claves de los diccionarios deben ser inmutables, ya que la operación numérica del hashing hecha sobre las claves debe dar siempre el mismo resultado (si se utilizara un valor mutable esto no sería posible).

Esta función de hashing y el manejo asociado a los diccionarios es en realidad lo que utiliza Python para generar las asociaciones entre identificadores (nombres de variables) y sus objetos referenciados, y permite también fácilmente controlar los diferentes ámbitos de referenciamiento que se genera durante una ejecución (por ejemplo durante el llamado a funciones).

Además de modificar los valores de una clave, podemos remover el par valor-clave con el método `pop()`. Este método remueve el ítem cuya clave se pasa por parámetro, y devuelve el valor asociado. Si quisiésemos que se elimine y retorne *el último* par clave-valor, utilizamos el método `popitem()` (sin argumentos). Todos los ítems pueden ser removidos de una utilizando el método `clear()`. Si queremos eliminar el diccionario, utilizamos la función genérica `del`.

```

>>>cuadrados = {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
>>>print(cuadrados.pop(4))
16
>>>cuadrados
{1: 1, 2: 4, 3: 9, 5: 25}
>>> print(cuadrados.popitem())
(5, 25)
>>> cuadrados
{1: 1, 2: 4, 3: 9}
>>> cuadrados.clear()
>>> cuadrados
{}
>>> del(cuadrados)
>>> cuadrados
Traceback (most recent call last): ...etc...

```

Los diccionarios se combinan e interoperan con las demás estructuras secuenciales de Python de una manera natural pero no se pueden aplicar '+' ni '*' (imaginan por qué?).

```

>>> cubos = {x : x**3 for x in range(7)}
>>> cubos
{0: 0, 1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216}

>>> lista = ['a', 'b', 'c']
>>> dicc = {x:'hola '+x for x in lista}
>>> dicc
{'a': 'hola a', 'b': 'hola b', 'c': 'hola c'}

>>> tupla = ((11, "once"), (21, "veintiuno"), (19, "diecinueve"))
>>> dct = dict((y, x) for x, y in tupla)
>>> dct
{'once': 11, 'veintiuno': 21, 'diecinueve':19}

```

Puede darse el caso que tengamos que armar un diccionario a partir de dos listas y necesitemos primero combinarlas en una tupla para poder armar el diccionario.

```

>>> alumnos = ['Lucas', 'Florencia', 'Matías']
>>> notas = [8, 9, 7]
>>> tupla = tuple(zip(alumnos, notas))
# el objeto zip( ) no es asignable!
>>> tupla
(('Lucas', 8), ('Florencia', 9), ('Matías', 7))

>>> tupla = tuple(zip(alumnos, notas, ['a', 'b', 'c']))
# no está limitado a aridad 2
>>> tupla
(('Lucas', 8, 'a'), ('Florencia', 9, 'b'), ('Matías', 7, 'c'))
>>> diccio = dict(zip(alumnos, notas))

```

```
>>> diccio
{'Lucas': 8, 'Florenceia': 9, 'Matías': 7}
```

Podemos ahora pensar en un miniproyecto: dado un texto cualquiera contar la frecuencia de ocurrencia de cada una de sus palabras.

1. Leemos el texto y lo convertimos en una lista de palabras utilizando el método split sobre el espacio.
2. Creamos un diccionario vacío (clave = palabra, valor = ocurrencias).
3. Iteramos sobre la lista:
 - Si el elemento de la lista no está en el diccionario, lo agregamos como clave nueva y con ocurrencia 1.
 - Si el elemento de la lista está, sumamos uno a sus ocurrencias.
4. Ordenamos el diccionario y generamos una salida organizada.

```
texto = input('ingresar texto (" " para terminar): ')

while texto :
    dicc = dict()
    palabras = texto.split(' ')

    for palabra in palabras :
        if palabra in dicc : dicc[palabra] += 1
        else: dicc[palabra] = 1

    dicc = dict(sorted(dicc.items())) # función genérica
    for palabra in dicc :
        print(palabra, dicc[palabra])
    texto = input('ingresar texto (" " para terminar): ')

print('Gracias por usar nuestros servicios')
```

Podemos modularizar algunas de esas tareas en funciones, lo cual nos permitiría luego un mantenimiento más rápido y una flexibilidad mayor:

```
def contar_palabras(texto) :
    diccio = dict()
    palabras = texto.split(' ')
    for palabra in palabras :
        if palabra in diccio : diccio[palabra] += 1
        else: diccio[palabra] = 1
    return(diccio)
```

```
def agregar_palabras(texto, diccionario) :
    diccio = contar_palabras(texto)
    for palabra in diccio :
        if palabra in diccionario : diccionario[palabra] += diccio[palabra]
        else: diccionario[palabra] = diccio[palabra]
    return(diccionario)
```

Al ejecutar este código sobre un texto cualquiera (extraído de un .doc, una página web, etc.) observamos un detalle que dificulta las posibles tareas subsiguientes.

```
a 4
(a 1
a, 1
...
de 3
de. 2
(de 1
...
no 1
no) 1
no. 2
```

Necesitamos entonces “limpiar” las palabras de los signos de puntuación, admiración, paréntesis, etc. Podemos definir un grupo de signos, y si algún carácter de la palabra pertenece a ese grupo lo eliminamos.

```
def QuitarSignos(palabra):
    signos=",.!:;¡!¿?()"
    if palabra[-1] in signos: palabra = palabra[:len(pal)-1]
    if palabra[0] in signos: palabra = palabra[1:]
    return palabra
```

Esta forma de proceder es ineficiente porque hay que ejecutar la función palabra por palabra (definiendo el grupo de signos para luego descartarlo en cada palabra, etc.). Resulta entonces mejor realizar esta limpieza durante el ingreso de las palabras al diccionario.

```
def contar_palabras(texto) :
    diccio = dict()
    signos=",.!:;¡!¿?()"
    palabras = texto.split(' ')
    for palabra in palabras :
        if palabra[-1] in signos: palabra = palabra[:len(pal)-1]
        if palabra[0] in signos: palabra = palabra[1:]
        if palabra in diccio : diccio[palabra] += 1
        else: diccio[palabra] = 1
    return(diccio)
```


2. Algunos métodos y funciones con diccionarios

- `{clave1:valor1, clave2:valor2}`

Se crea un nuevo diccionario con los valores asociados a las claves. Si no se ingresa ninguna pareja de clave y valor, se crea un diccionario vacío.

- `diccionario[clave]`

Accede al valor asociado con clave en el diccionario. Falla si la clave no está en el diccionario.

- `clave in diccionario`

Indica si un diccionario tiene o no una determinada clave.

- `diccionario.get(clave, valor_predeterminado)`

Devuelve el valor asociado a la clave. A diferencia del acceso directo utilizando `diccionario[clave]`, en el caso en que el valor no se encuentre devuelve el `valor_predeterminado`.

- `for clave in diccionario:`

Permite recorrer una a una todas las claves almacenadas en el diccionario.

- `diccionario.keys()`

Devuelve una secuencia desordenada, con todas las claves que se hayan ingresado al diccionario.

- `diccionario.values()`

Devuelve una secuencia desordenada, con todos los valores que se hayan ingresado al diccionario.

- `diccionario.items()`

Devuelve una secuencia desordenada con tuplas de dos elementos, en las que el primer elemento es la clave y el segundo el valor.

- `diccionario.pop(clave)`

Quita del diccionario la clave y su valor asociado, y devuelve el valor.

- `diccionario.popitem()`

Quita del diccionario el último par clave-valor (puede variar) y devuelve el valor del par.

- `diccionario.clear()`

Quita todos los pares del diccionario.

3. Conjuntos

Un conjunto es una estructura de datos mutable que consiste en una colección de valores inmutables que no pueden estar repetidos (esto es análogo al conjunto de claves en un diccionario).

```
>>> conjunto = {2, 'abc', (1,2,3)}
```

Observar que el conjunto no puede contener objetos mutables (listas, diccionarios, conjuntos) y tampoco elementos duplicados.

```
>>> conj = {1,2,3,4,3,2}
>>> conj
{1, 2, 3, 4}
```

Todas las estructuras secuenciales son “convertibles” a conjuntos utilizando la función genérica `set(estructura)`, y un conjunto es convertible a las demás estructuras secuenciales con las funciones correspondientes. Utilizamos `set()` (en forma literal) para crear o denotar conjuntos vacíos (dado que `{}`) representa un diccionario vacío.

```
>>> set([1, 2, 3, 3])
{1, 2, 3}
>>> set((1, 2, 3, 4, 3))
{1, 2, 3, 4}
>>> set('pepin')
{'i', 'p', 'e', 'n'}
>>> set({1, 2, 3})
{1, 2, 3}
>>> set({'a':1, 'b':2})
{'a', 'b'}
>>> conj = set()
>>> conj
set() # set() representa el conjunto vacío
```

Podemos agregar elementos a un conjunto, de a uno con el método `add()` y de a varios con el método `update()` (que recibe tuplas, listas o conjuntos, o también tuplas de tuplas, etc.).

```
>>> conj.add(2) # antes de esta sentencia conj estaba vacío
>>> conj
{2}
>>> conj.update([1,2,3,4,5])
>>> conj
{1, 2, 3, 4, 5}
>>> conj.update(['a', 'b'],{(1,2),'c'})
>>> conj
{1, 2, 3, 4, 5, 'a', (1, 2), 'c', 'b'}
# notar que el orden es 'impredecible' pero también es irrelevante
```

Para quitar elementos de un conjunto tenemos los métodos `discard()` y `remove()`, que reciben por parámetro el elemento que se desea quitar. Ambos trabajan de igual manera, salvo que `remove()` devuelve un error si se desea quitar un elemento que no se encuentra en el conjunto. Al igual que con otras estructuras compuestas, tenemos los métodos `pop()` y `clear()`.

```
>>> conj.discard(6)
>>> conj
{1, 2, 3, 4, 5, 7, 8, 9, 'a', (1, 2), 'c', 'b'}
>>> conj.remove(7)
>>> conj
{1, 2, 3, 4, 5, 8, 9, 'a', (1, 2), 'c', 'b'}
>>> conj.remove(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 7
>>> conj.discard(7)
>>> conj
{1, 2, 3, 4, 5, 8, 9, 'a', (1, 2), 'c', 'b'}
```

Es posible realizar las operaciones matemáticas usuales entre conjuntos (unión, intersección, diferencia, y “diferencia simétrica”. El operador `in` también aplica.

```
>>> A = {1, 2, 3, 4}
>>> B = {3, 4, 5, 6}
>>> A|B
{1, 2, 3, 4, 5, 6}
>>> A.union(B)
{1, 2, 3, 4, 5, 6}
>>> B.union(A)
{1, 2, 3, 4, 5, 6}
>>> A&B
{3, 4}
>>> A.intersection(B)
{3, 4}
>>> B.intersection(A)
{3, 4}
```

```
>>> A-B
{1, 2}
>>> B-A
{5, 6}
>>> A.difference(B)
{1, 2}
>>> B.difference(A)
{5, 6}
```

```
>>> A^B
{1, 2, 5, 6}
>>> A.symmetric_difference(B)
{1, 2, 5, 6}
>>> B.symmetric_difference(A)
{1, 2, 5, 6}
```

```
>>> for x in A&B : print(x)
...
3
4
```

4. Funciones aplicables a conjuntos

- `all()`

Retorna `True` si ningún elemento del conjunto evalúa a `False` o si el conjunto es vacío.

- `any()`

Retorna `False` si todos los elementos del conjunto evalúan a `False` o si el conjunto es vacío.

- `enumerate()`

Retorna objeto no asignable (como el zip) que consiste en una tupla de pares enumerados (índice, valor).

- `len()`, `max()`, `min()`, `sum()`

Retorna la longitud, mayor elemento, menor elemento, o suma entre los elementos del conjunto.

Como los conjuntos pueden ser claves adecuadas o útiles en un diccionario, pero no podrían serlo por ser mutables, existe el tipo `frozenset`, que es una versión inmutable de conjuntos (algo así como la relación que hay entre tuplas y listas).



```
>>> A = frozenset([1,2,3,4])
>>> A
frozenset({1, 2, 3, 4})

>>> A&B # B es el (no frozen) set anterior
frozenset({3, 4})

>>> A.add(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

5. Ordenamiento de secuencias

Ya vimos los ejemplos naturales de ordenamiento de secuencias de valores simples (listas, tuplas, conjuntos) con la función `sorted()`.

```
>>> T = (5, 6, 4, 3, 9)
>>> sorted(T)
[3, 4, 5, 6, 9]
>>> C = {'f', 'g', 'a', 'z'}
>>> sorted(C)
['a', 'f', 'g', 'z']
```

Eventualmente nos sirve para secuencias de tuplas, ordenando por el primer elemento.

```
>>> S = tuple(zip(T,C))
>>> S
((5, 'a'), (6, 'g'), (4, 'z'), (3, 'f'))
>>> sorted(S)
[(3, 'f'), (4, 'z'), (5, 'a'), (6, 'g')]
```

Para poder ordenar estas secuencias, sus elementos deben poder ser comparables entre sí a través del operador de comparación “<” (tipos numéricos, strings y tuplas), siempre entre elementos del mismo tipo de dato. Si en algún caso quisiésemos que esa comparación se realice de una manera diferente, podemos utilizar el parámetro opcional `key`, en el cual se indica la manera de comparar los elementos de la secuencia. Lo ideal es indicar a través de este parámetro cuál será la función para comparar.

Recordamos nuestro ejemplo de estructura de datos:

```
>>> Lista = [['241251', 'Adobatrck, Gobtrya Itun', '49114401'],
['259497', 'Anotrosso, Sobasskyan', '51953065'],
['249120', 'Cassktro, Fahwndo Nyhoáass', '49768958'],
... etc.
```

Definimos entonces funciones para ordenar por legajo, nombre o dni:

```
>>> def legajo(L) : return L[0]
...
>>> def nombre(L) : return L[1]
...
>>> def dni(L) : return L[2]
```

Podemos ahora obtener los listados ordenados por el campo deseado:

```
>>> for x in sorted(Lista, key=legajo) : print(x)
...
['230823', 'Otrkyz, Ronado Yaytr', '45747617']
['235509', 'Mayo Patraz, Danya', '47955159']
['241251', 'Adobatrck, Gobtrya Itun', '49114401']
['249120', 'Cassktro, Fahwndo Nyhoáass', '49768958']
['249593', 'Ctryhho, Nyhoáass', '50306810']
['250110', 'Shhnyk, Lwhass Gobtrya', '49970445']
['253198', 'Pahhaho, Mayho Aaxyss', '50504008']
...
etc...
```

Idem con los otros dos criterios de ordenación.

Funciones Lambda

Muchas veces necesitamos definir estos criterios en forma rápida y flexible (en realidad muchas veces necesitamos definir funciones simples en condiciones similares). Para esto tenemos las “funciones lambda”.

```
>>> for x in sorted(Lista, key=lambda f: Lista[1]) : print(x)
...
['219783', 'Adobatrck, Gobtrya Itun', '49114401']
['259497', 'Anotrosso, Sobasskyan', '51953065']
['249120', 'Cassktro, Fahwndo Nyhoáass', '49768958']
['249593', 'Ctryhho, Nyhoáass', '50306810']
['253509', 'Gogg Aatr, Ftranhyssho Agwsskyn', '50695238']
['260755', 'Gonaz Latra, Yania Vatrónyha', '51975758']
etc...
```

6. Archivos

Los archivos requieren un mecanismo un poco más complejo que otras estructuras:

1. Primero necesitamos asociar un identificador al archivo (este identificador es como cualquier otra variable, pero se denomina *handler* por razones que se van a ir aclarando).
2. Esta asociación se hace con la función predefinida `open`, a la cual hay que indicarle por parámetro el nombre del archivo que queremos gestionar, y el modo de acceso).
3. Si luego de esta asociación, el archivo (existente en el *path* desde el cual se ejecuta el script o el intérprete) fue accedido por el handler en modo lectura, entonces el handler nos permite acceder a una copia (*buffer*) del archivo como si fuese una secuencia de strings.

```
>>> archi = open('mastermind.py', 'r')
>>> for L in archi : print(L)
...
import random

print("Bienvenidx al Mastermind!")

print("TenÃ@s que adivinar un nÃºmero de cuatro dÃ¡gitos distintos")

(sigue la escritura del resto del archivo)
```

En este ejemplo, hemos asumido que el archivo `'mastermind.py'` estaba “en el path”, es decir en la misma carpeta o folder desde la cual estamos corriendo el intérprete. Lo abrimos con `'r'` para indicar solo lectura (read).

Podemos abrir un archivo cualquiera indicando el path absoluto:
`c:\carpeta\...\archivo.py.`

Nos irrita un poco que los acentos salgan mal. Esto se debe al *encoding* utilizado por el notepad para los caracteres que no están entre los primeros 127 de ASCII.

UTF-8 (8-bit Unicode Transformation Format) es un encoding que utiliza símbolos de longitud variable. Está definido como estándar de la Internet Engineering Task Force (IETF). Sus características principales son:

- Es capaz de representar cualquier carácter de cualquier lenguaje que sea representable en Unicode, incluyendo los ASCII de 7 bits en forma transparente.
- Usa símbolos de longitud variable (de 1 a 4 bytes por carácter).
- Los conjuntos de valores que puede tomar cada byte de un carácter multibyte, son disjuntos, por lo que no es posible confundirlos entre sí.

Estas características lo hacen atractivo en la codificación de correos electrónicos y páginas web. El IETF requiere que todos los protocolos de Internet indiquen qué codificación utilizan para los textos y que UTF-8 sea una de las codificaciones contempladas. El Internet Mail Consortium (IMC) recomienda que todos los programas de correo electrónico sean capaces de crear y mostrar mensajes codificados utilizando UTF-8.

Intentamos entonces avisarle al handler que utilice UTF-8 como encoding:

```
>>> archi = open('mastermind.py', 'r', encoding = 'utf-8')
>>> for l in archi : print(l)
...
import random

print("Bienvenidx al Mastermind!")

print("Tenés que adivinar un número de cuatro dígitos distintos")

intentos = 0 (sigue la escritura del resto del archivo)
```

Es entonces importante tener en cuenta el formato del archivo a abrir (y averiguar qué otros encodings existen) en caso de no estar trabajando con archivos ASCII “planos”. En esa llamada a open hemos utilizado dos parámetros “por posición” y uno “por nombre de parámetro”.

7. El cursor (los archivos no son estructuras!)

Si bien el ejemplo anterior puede hacernos creer que un archivo es una secuencia de strings, tenemos que observar varios aspectos. Hagamos varias pruebas:

```
>>> archi = open('mastermind.py', 'r', encoding = 'utf-8')
>>> archi
<_io.TextIOWrapper name='mastermind.py' mode='r' encoding='utf-8'>
```


Esto quiere decir que el archivo es un “objeto no legible” (como son los `zip`).

```
>>> archi = open('mastermind.py', 'r', encoding = 'utf-8')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'mastermind.py'
>>> archi
<_io.TextIOWrapper name='mastermind.py' mode='r' encoding='utf-8'>
```

Es decir, si el handler falló, no cambia su estado.

```
>>> archi = open('mastermind.py', 'r', encoding = 'utf-8')
>>> for L in archi : print(L)
...
import random

print("Bienvenidx al Mastermind!")

print("Tenés que adivinar un número de cuatro dígitos distintos")

intentos = 0 (sigue la escritura del resto del archivo)

>>> for L in archi : print(L)
...
>>> #!!!
```

No es que el archivo “se borró” (o el buffer en memoria local), sino que el `for` en este caso recorre un *cursor*, el cual al final de la recorrida queda al final del archivo. Podemos “resetear” el cursor (a la línea deseada) utilizando el método `seek()`. El parámetro indicado es la cantidad de *caracteres* desde el comienzo del archivo a la que queremos que el cursor pase a señalar. Si queremos posicionar el cursor en un lugar dado del archivo, indicamos `archivo.seek(desplazamiento, lugar)`, dónde `desplazamiento` indica cuántos caracteres desde `lugar` nos queremos desplazar, y `lugar` puede ser 0 (comienzo), 1 (posición actual), o 2 (final). Para ejemplificar pensemos en el siguiente archivo:

```
0                                     (o sea '0/n') etc.
1
2
...
9
>>> archi = open('ejemplo.py', 'r')
>>> for L in archi : print(L)
...
0
1
2
...
9
```

Luego de esta ejecución el cursor quedó al final del archivo. Lo reseteamos a una posición arbitraria (tres caracteres desde el comienzo implican haber leído `'0/n'`).

```
>>>archi.seek(3,0)
3
>>> for L in archi : print(L)
...
1
2
...
9
```

8. Archivos y estructuras secuenciales

Si queremos copiar el archivo en una estructura, tenemos varios caminos:

1. Copiarlo en una lista utilizando la asignación

```
>>> data = [L for L in archi] # copiará desde el cursor hasta el final
```

Este mecanismo es equivalente al uso del método `readlines()`.

```
>>> data = archi.readlines() # copiará desde el cursor hasta el final.
```

El uso de `readlines()` es más flexible, dado que un valor *no nulo* indica copiar solo una línea y deja el cursor al final de la última línea que copió.

```
>>> archi.seek(0,0)
0
>>> data = archi.readlines(2)
>>> data
['import random\n']
>>> data = archi.readlines(2)
>>> data
['print("Bienvenidx al Mastermind!")\n']
>>> data = archi.readlines(2) + archi.readlines(2)
>>> data
['print("Tenés que adivinar un número de cuatro dígitos distintos")\n',
 'intentos = 0\n']
>>>
```

2. Copiarlo en alguna otra estructura secuencial

```
>>> conj = set(archi) # con el cursor en 0
>>> conj
{'# para determinar si ganaste\n', ' while candidato in codigo:
candidato = random.choice(digitos)\n', 'while not ganaste:\n',
'ganaste = False\n', ' propuesta ... }

>>> tupli = tuple(archi) # con el cursor en 0
>>> tupli
('import random\n', 'print("Bienvenidx al Mastermind!")\n', 'print("Tenés
que adivinar un número de cuatro dígitos distintos")\n', 'intentos = 0\n',
'ganaste ... )
```

3. Convertirlo en un diccionario, de manera que (por ejemplo) cada línea tenga su número como clave. Aquí nos sirve el iterador `enumerate()` que se asocia a una variable y le asigna valores enteros como un contador dentro de una estructura secuencial:

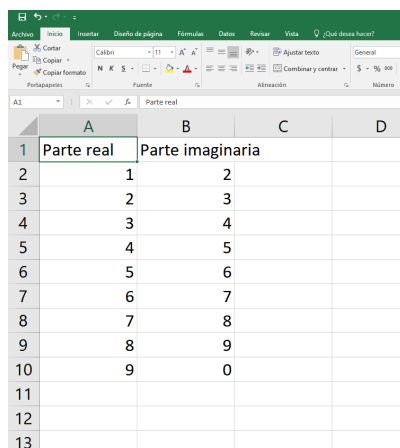
```
>>> dicci = {I+1:L for I,L in enumerate(archi)}
>>> dicci
{1: 'import random\n', 2: 'print("Bienvenidx al Mastermind!")\n', 3:
'print("Tenés que adivinar un número de cuatro dígitos distintos")\n', 4:
'intentos = 0\n', ... }
```

Podemos eliminar los `\n` al final de cada línea (indican a la consola exactamente eso) utilizando el método `.rstrip()` sobre strings.

```
>>> dicci= {I+1:L.rstrip('\n') for I,L in enumerate(archi)}
>>> dicci
{1: 'import random', 2: 'print("Bienvenidx al Mastermind!")', 3:
'print("Tenés ... }'
```

Podríamos haberlo hecho sobre el diccionario anterior por ‘dictionary comprehension’?

Ejemplo: Tenemos el siguiente archivo excel guardado en formato CSV.



	A	B	C	D
1	Parte real	Parte imaginaria		
2	1	2		
3	2	3		
4	3	4		
5	4	5		
6	5	6		
7	6	7		
8	7	8		
9	8	9		
10	9	0		
11				
12				
13				

Viéndolo con un editor de texto tendríamos lo siguiente:

```
Parte real,Parte imaginaria
1,2
2,3
3,4
4,5
5,6
6,7
7,8
8,9
9,0
```

Nuestro objetivo es poder abrirlo, leer los pares de valores real-imaginario, y generar complejos para cada línea:

```
>>> archi = open('complejos.csv', 'r')
>>> archi.readline()
'Parte real,Parte imaginaria'
>>> for L in archi :
...     tupli = tuple(L)
...     c = complex(int(tupli[0]), int(tupli[2]))
...     print(c)
...
(1+2j)
(2+3j)
etc.
(9+0j)
```

9. Cerrar un archivo

Antes de finalizar la ejecución de un programa/script, es necesario *cerrar* los archivos que se abrieron, dado que ese comando copia la información existente en los buffers asociados a los archivos originales (salvo que explícitamente se hubiesen guardado o “salvado”). Para ello se ejecuta el método `.close()`.

```
>>> archi.close()
>>> archi
<_io.TextIOWrapper name='mastermind.py' mode='r' encoding='utf-8'>
>>> archi.seek(0,0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

El handle sigue existiendo pero no nos permite seguir operando. Una forma más prolija es utilizar la estructura `with`, que nos permite generar contextos dentro de nuestro programa/script. En este caso:

```

with open("mastermind.py", encoding = 'utf-8') as archi :
    i = 1 #
    for linea in archi :
        linea = linea.rstrip("\n")
        print("{}: {}".format(i, linea))
        i += 1

# Cuando la ejecución sale del bloque 'with', el archivo se cierra automáticamente.

```

10. Escribiendo en archivos

Para escribir en un archivo tenemos que abrirlo en el modo de sólo escritura ('w'). En este caso el archivo existente es vaciado si existe, y se lo crea si no existe. Hay un modo sólo escritura posicionándose al final del archivo ('a') o append. En este caso se crea el archivo, si no existe, pero en caso de que exista se posiciona al final, manteniendo el contenido original.

En cualquiera de los modos se puede agregar un + para pasar a un modo lectura-escritura. El comportamiento de r+ y de w+ no es el mismo, ya que en el primer caso se mantiene el archivo completo, y en el segundo el archivo se vacía. Si un archivo no existe y se lo intenta abrir en modo lectura, se generará un error; en cambio si se lo abre para escritura, se crea el archivo al momento de abrirlo. En caso de que no se especifique el modo, los archivos serán abiertos en modo sólo lectura.

De la misma forma que para la lectura, existen dos formas distintas de escribir a un archivo, mediante cadenas (archivo.write(cadena)) o mediante listas de cadenas (archivo.writelines(lista_de_cadenas)). Así como la función readline devuelve las líneas con los caracteres de fin de línea (\n), será necesario agregar los caracteres de fin de línea a las cadenas que se vayan a escribir en el archivo.

```

with open("saludito.py", "w") as saludo :
    saludo.write("print('Hola Mundo')\n")
    saludo.write("# Je! Este programa fue generado por otro programa!\n")

```

Podemos guardar cualquier estructura de Python en un archivo, solo que si la estructura no es un string la tenemos que convertir a tal formato utilizando la función str() que ya conocemos. También podemos interrogar la posición dentro del archivo en la que estamos con el método tell().

Abrir un archivo en modo **'a'** puede parecer raro, pero es bastante útil. Uno de sus usos es para logs, que nos permite ver en su secuencia cronológica los distintos eventos que se fueron sucediendo (por ejemplo el muro de una red social, los comentarios a lectores en un medio digital, etc.). En los log automáticos que generan los sistemas (operativo, navegador, correo, etc.) ver la secuencia cronológica permite encontrar los pasos (no siempre evidentes) que ha ejecutado nuestro programa y encontrar un error en alguno de ellos.

Método sobre un archivo	Descripción
<code>close()</code>	Cierra un archivo abierto (no hace nada si el archivo no está abierto, da error si no hay un handler).
<code>fileno()</code>	Devuelve un entero denominado <i>file descriptor</i> del archivo.
<code>flush()</code>	Vacía el buffer de escritura del archivo.
<code>read(n)</code>	Lee como máximo n caracteres del archivo. Si n es nulo o vacío, lee hasta el final del archivo.
<code>readline(n=-1)</code>	Lee y retorna una línea del archivo si el parámetro no se especifica o es -1. Lee como máximo n bytes si se especifica el parámetro.
<code>readlines(n=-1)</code>	Lee y retorna una lista de líneas hasta el final del archivo si el parámetro no se especifica o es -1. Lee como máximo n bytes si se especifica el parámetro.
<code>seek(offset, dónde)</code>	Cambia la posición del cursor a offset bytes, en referencia a dónde (comienzo, posición actual del cursor, final).
<code>tell()</code>	Retorna la posición (en bytes) del cursor.
<code>writable()</code>	Retorna True si el buffer permite ser accedido para escritura.
<code>write(s)</code>	Escribe el string s en el archivo y retorna la cantidad de bytes escrita.
<code>writelines(líneas)</code>	Escribe una lista líneas de líneas en el archivo.