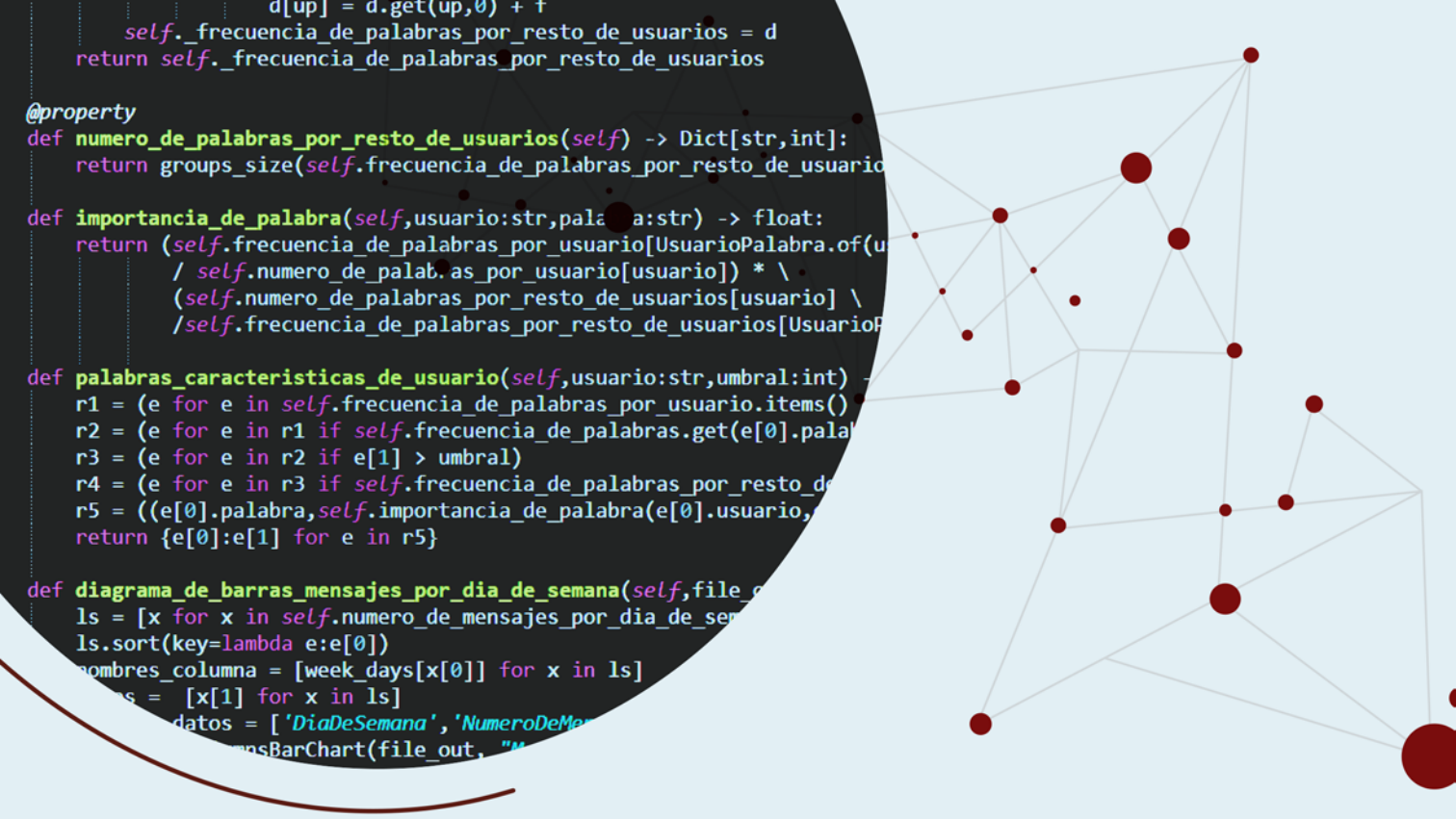


FUNDAMENTOS DE PROGRAMACIÓN: PYTHON

Miguel Toro Bonilla

Editorial Universidad de Sevilla

Fundamentos de programación: PYTHON

Fundamentos de Programación: Python Miguel Toro Bonilla

Editorial Universidad de Sevilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática

Fundamentos de Programación: Python Miguel Toro Bonilla

Miguel Toro Bonilla

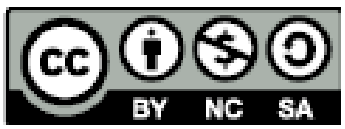
Fundamentos de programación: PYTHON

Editorial Universidad de Sevilla

Sevilla 2022

Fundamentos de Programación: Python Miguel Toro Bonilla

Colección: Manuales de Informática del
Instituto de Ingeniería Informática (I3US)
Núm.: 2



Editorial Universidad de Sevilla 2022
c/ Porvenir, 27 - 41013 Sevilla.
Tlfs.: 954 487 447; 954 487 451; Fax: 954 487 443
Correo electrónico: eus4@us.es
Web: https://editorial.us.es

DOI: https://dx.doi.org/10.12795/9788447223602

Maquetación: Miguel Toro

Diseño de cubierta y edición electrónica:
referencias.maquetacion@gmail.com

Agradecimientos

Para aprender a programar lo mejor es programar. En esta asignatura de Fundamentos de Programación vamos a aprender los conceptos básicos de la programación. Estos conceptos los vamos a concretar en dos lenguajes: Python y Java. En este volumen veremos Python.

Los lenguajes de programación tienden a ir compartiendo las mismas ideas básicas. Cada lenguaje va tomando prestadas las ideas más novedosas aportadas por otros. Aunque cada lenguaje tiene sus particularidades, podemos decir que los lenguajes de programación van convergiendo hacia elementos comunes a todos ellos. Por ello parecen sensato aprender las ideas comunes a los principales lenguajes actuales y extrapolar ideas y conceptos de unos a otros.

Vamos a abordar conceptos en Python comunes con Java que se pueden extender a otros.

El diseño de tipos ocupará un lugar central en este material. Al final se incluyen varios ejemplos que parten de un diseño de tipos.

Insistiremos en Python en un enfoque basado en iterables y reutilización de funciones.

El material en este texto procede de la experiencia de varios años de enseñanza de la asignatura de Fundamentos de Programación en la Universidad de Sevilla.

Mucho de este material está tomado de versiones anteriores de los profesores José A. Troyano, Carlos G. Vallejo, Mariano González, Daniel Mateos, Fermín Cruz, Beatriz Pontes a los que quiero agradecer sus esfuerzos y dedicación. Estas versiones anteriores han sido transformadas hasta alcanzar la forma actual. Sus defectos son responsabilidad única del autor.

En https://github.com/migueltoro/python_v1 puede encontrarse el código de los ejemplos. Próximas versiones se encontrarán en https://github.com/migueltoro/python_*.

Miguel Toro

Sevilla, septiembre de 2021

Índice

Conceptos básicos de programación: Python

Instrucciones y funciones

Un programa Python está formado por instrucciones (a veces también son llamadas sentencias). Cada instrucción se escribe normalmente en una línea. Por ejemplo, la siguiente instrucción sirve para imprimir el mensaje "Hola, mundo!" en la pantalla.

```
print("Hola, mundo!")
```

Si en un programa hay varias instrucciones, estas se ejecutan secuencialmente, una tras otra.

```
print("Hola, mundo!")
print("Adiós, mundo!")
```

La instrucción que hemos usado en los ejemplos anteriores, *print* (que significa imprimir en inglés), es una función predefinida (built-in function). Una función puede ser *llamada* (también se dice *invocada*) desde un programa escribiendo su nombre y a continuación unos paréntesis de apertura y cierre. En ocasiones, las funciones reciben parámetros, que se escriben en la llamada dentro de los paréntesis, separados por comas si hay más de uno.

Hay muchas funciones predefinidas, por ejemplo la función help nos proporciona ayuda sobre cualquier otra función.

```
help(print)    # Nos muestra información de ayuda sobre la
                 # función predefinida print
```

Por supuesto, la ayuda está en inglés. En el ejemplo anterior hemos usado el carácter almohadilla (#) para escribir un **comentario** sobre el código. Los comentarios se usan en los programas para hacer aclaraciones, explicar el funcionamiento y, en general, hacer más fácil de entender nuestros programas.

Las funciones anteriores se llaman funciones predefinidas porque ya vienen incluidas *por defecto* en Python. Los programadores pueden *definir* sus propias funciones, como se muestra a continuación.

```
'''
Esta función imprime un saludo personalizado con el nombre
indicado mediante el parámetro "nombre".
'''
def saluda(nombre:str)->None:
    print("Hola, " + nombre)
```

La función que acabamos de definir se llama *saluda* y recibe un único parámetro. Después de la primera línea (llamada *cabecera* o *prototipo* de la función) vienen una o varias instrucciones (llamadas *cuerpo* de la función), que serán ejecutadas cuando alguien llame a la función. Las instrucciones que conforman el cuerpo de la función aparecen indentadas, es decir, tienen un tabulador delante.

La cabecera de la función está compuesta por el nombre, los parámetros con su tipo y el tipo del resultado devuelto por la función. Los parámetros que aparecen en la defición de la función se llaman *parámetros formales*.

Las líneas que comienzan y acaban con tres comillas simples son *comentario de documentación* de la función y, aunque no son obligatorias, sí es recomendable incluirlas. Cuando alguien llame a la función predefinida *help* para pedir ayuda sobre nuestra función, se mostrará precisamente el texto aquí incluido.

Tras definir una función, podemos llamarla pasándole unos parámetros. Por ejemplo:

```
saluda("Juan")
```

Los parámetros que aparecen en la llamada de la función se llaman *parámetros reales*. Lo que hará la función cuando alguien la llame será ejecutar su cuerpo sustituyendo los parámetros formales por los parámetros reales. En este caso mostrar un mensaje de saludo.

El mensaje que aparecerá en la pantalla será:

```
Hola, Juan
```

Al definir una función conseguimos reutilizar la funcionalidad implementada las veces que queramos, simplemente llamando a la función en las partes del programa en que necesitemos dicha funcionalidad. Por ejemplo, ahora puedo realizar varios saludos, uno tras otro:

```
saluda("John")
saluda("Paul")
saluda("George")
saluda("Ringo")
```

La función *saluda_educado* recibe un nombre como parámetro e imprime el mismo saludo que la función saluda pero personalizado, es decir a continuación imprime el texto "Encantado de conocerle".

```
def saluda_educado(nombre:str)->None:
    saluda(nombre)
    print("Encantado de conocerle")

saluda_educado('Fermín')
```

Expresiones y tipos

Al definir la función *saluda* hemos escrito:

```
print("Hola, " + nombre)
```

Hemos usado un *operador*, el signo +, que sirve para concatenar las dos cadenas de texto: la cadena *"Hola, "* y la cadena contenida en el parámetro *nombre*.

El uso de operadores es habitual en todos los lenguajes de programación. Las *expresiones* están formadas por operadores (como el +), literales (como la cadena *"Hola, "*) y variables (como el parámetro nombre). Una expresión es siempre evaluada por Python, para obtener un resultado, antes de seguir ejecutando el resto de las instrucciones del programa.

Veamos algunos ejemplos de expresiones utilizando los *operadores aritméticos*. Algunos ejemplos de expresiones son:

```
3 + 5
3 + 5 * 8
(3 + 5) * 8
(3 + 5) * 8 / 14
3//4
4%5
```

En las expresiones lógicas se usan distintos *operadores lógicos* (como el operador *and*) *y operadores relacionales* (como los operadores > o ==). Iremos viendo más operadores de estos tipos a medida que los vayamos necesitando. Si queremos ver el valor de una expresión en pantalla lo hacemos con la fución predefina *print*.

```
print(4 > 5)
print(4 < 5 and 5 < 6)
print(3+1 == 4)
```

En las expresiones también podemos usar llamadas a funciones, siempre que sean funciones que devuelvan algún resultado. Para que una función devuelva un valor cuando sea invocada, usaremos la instrucción *return* en algún punto del cuerpo de la función (casi siempre, al final del cuerpo

de la función). Estas funciones tienen un tipo de retorno. En este caso *float*. Por ejemplo:

```
def doble(x:float)->float:
    return x * 2

doble(10)
```

Una función puede tomar varios parámetros. Definamos una función de nombre *descuento* que reciba dos parámetros: un valor y un porcentaje y devuelva el resultado de aplicar el descuento correspondiente al porcentaje sobre el valor indicado.

```
def descuento(precio:int,des:int)->float:
    return precio - precio*des/100
```

El resultado de una expresión podemos guardarlo para utilizarlo más adelante en nuestro programa. Para esto se usan las *variables*. Cada variable tiene un tipo. Ejemplos:

```
precio:int = 200
r1:float = descuento(precio, 10)
print("Precio con 10% de descuento:",r1)
r2:float = (3 + 5) * doble(8)
print(r2 / 5)
```

Hemos *asignado* a la variable *r1* el valor de la expresión que hemos escrito a la derecha del carácter igual (=), de manera que más adelante en el programa podremos utilizar ese valor mencionando la variable. A este carácter igual lo llamamos *operador de asignación*. A la línea completa formada por una variable, el operador de asignación y una expresión se denomina instrucción o *sentencia de asignación*. Cuando asignamos un valor a una variable decimos que la variable está *definida*. Es decir guarda un valor. El valor de una variable definida puede ser *usado* más tarde. Decimos que estamos usando esa variable. Una variable se puede usar solo si está definida previamente.

En este ejemplo hemos *asignado* a la variable *r1* el valor de la expresión *descuento(precio, 10)* que en este caso es simplemente la llamada a una función que hemos definido previamente. Hemos usado el valor de *r1*

como parámetro de la función print. Podemos observar que la función print puede tener varios parámetros. En ese caso se imprimen en pantalla uno tras otro.

Hemos definido la variable *r2* con el valor de la expresión *(3 + 5) * doble(8).* Hemos usado su valor en la expresión posterior *r2 / 5*.

Tipos predefinidos

En los ejemplos anteriores hemos guardado valores de distintos tipos en las variables: tipo cadena de caracteres, tipo entero y tipo real. Cada uno de estos son tipos predefinidos en Python (built-in types). Hablamos de predefinidos porque Python también permite al programador crear sus propios tipos mediante clases. Los valores que hemos escrito para inicializar cada una de las variables se llaman literales.

Un tipo de datos está definido por un conjunto de posibles valores (lo que en matemáticas se conoce como dominio) y un conjunto de operaciones asociadas. Por ejemplo, el tipo número entero (int) se corresponde con los valores 0, -1, 1, -2, 2 ..., y con las operaciones aritméticas (suma, resta, multiplicación, ...).

Un literal (es decir, un valor concreto de un tipo) tiene asociado un tipo determinado, dependiendo de como está escrito dicho literal. Por otra parte para saber el tipo asociado a una variable, debemos fijarnos en el valor que ha sido almacenado en la misma.

Una función recibe uno o varios valores de un tipo determinado, sus parámetros, y puede devolver un valor de este u otro tipo. Las llamadas a las funciones pueden estar representadas por un operador o por un identificador más unos parámetros reales, como veremos más adelante.

En las siguientes secciones se muestran distintos tipos predefinidos, la manera en que se escriben sus literales y sus operaciones asociadas más importantes.

Tipo lógico

El tipo lógico *(bool)* únicamente incluye dos valores en su dominio: verdadero *(True)* y falso *(False).* Estas dos palabras son precisamente los únicos *literales lógicos* permitidos en Python. El tipo lógico sirve para representar resultados de expresiones lógicas, por ejemplo, si un peso es o no mayor a un umbral, si un año es o no bisiesto, o si el personaje de un videojuego tiene o no una determinada habilidad.

Los operadores lógicos son solo tres: *and, or* y *not*, tal como se muestra en los siguientes ejemplos.

```
# Disyunción (también llamado "o lógico" y "sumador lógico")
False or True

# Conjunción (también llamado "y lógico" y "multiplicador
lógico")
False and True

# Negación
not False

print("Disyunción:", False or False)
print("Conjunción:", False and False)
print("Negación:", not False)
```

Tipo cadena

El tipo cadena de caracteres (*str*), o como se suele abreviar, tipo cadena, nos permite trabajar con cadenas de caracteres. Los *literales cadenas de caracteres* se escriben utilizando unas comillas simples o dobles para rodear al texto que queremos representar. Por ejemplo:

```
"Este es un literal cadena"
'Este es otro literal cadena'
```

Si usamos comillas simples, dentro del texto podemos emplear las comillas dobles sin problema. Igualmente, si usamos las comillas dobles para rodear al texto, dentro del mismo podemos usar las comillas simples. Por ejemplo:

```
"En este ejemplo usamos las 'comillas simples' dentro de un  \
      texto"
'En este ejemplo usamos las "comillas dobles" dentro de un
texto'
```

En ocasiones queremos hacer referencia a caracteres especiales, como el tabulador o el salto de línea. En dichos casos, debemos usar el *carácter de escape*, que es la barra invertida \. Este carácter al final de una línea indica que esta no ha acabado, que continua en la siguiente como en el ejemploo anterior.

El tabulador se escribe como *t* y el salto de línea se escribe como *n*. Por ejemplo:

```
"Este texto tiene dos líneas.\nEsta es la segunda línea."
```

También es posible utilizar tres comillas, simples o dobles, como delimitadores del texto, en cuyo caso podemos escribir texto de varias líneas, sin necesidad de usar *n*:

```
"""Este texto tiene dos líneas.
Esta es la segunda línea."""
```

Veremos más adelante otras operaciones sobre cadenas de caracteres. Por ahora, nos basta con ver las operaciones que podemos realizar mediante operadores o funciones predefinidas:

```
texto:str = "Este es un texto de prueba."
```

Tamaño de una cadena, función predefinida len

```
print("Número de caracteres del texto:", len(texto))
```

El operador de acceso, [], permite obtener un único carácter. El primer carácter se referencia mediante un cero.

```
print(texto[0])
print(texto[1])
print(texto[26])
```

Otra forma de acceder al último carácter de la cadena

```
print(texto[-1])
```

Si se intenta acceder a una casilla que no existe se produce un error.

```
print(texto[27])
```

Una casilla no existe si su índice está fuera del rango permitido. Un indice *i* está en el rango permitido si *0<=i<len(texto)*.

Python nos permite usar el operador + entre dos cadenas, y el operador * entre una cadena y un número entero. También es posible usar los *operadores relacionales* entre cadenas, de manera que se utiliza el orden alfabético para decidir el resultado de las operaciones.

```
print(texto + " ¡Genial!")
print(texto * 4)
print("Ana" < "María")
```

Las cadenas tienen, además otros métodos. Por ejemplo el método *split* que divide una cadena en partes separadas por un delimitador.

```
print('../../../resources/datos_2.txt'.split('/'))
```

El resultado de la sentencia anterior es:

```
['..', '..', '..', 'resources', 'datos_2.txt']
```

Si quisieramos partir por varios delimitadores disponemos de la función *split* del módulo *re*

```
print(re.split('[ ,]','En un lugar de la Mancha, de cuyo \
nombre no quiero acordarme'))
```

Cuyo resultado es

```
['En', 'un', 'lugar', 'de', 'la', 'Mancha', '', 'de', 'cuyo',
'nombre', 'no', 'quiero', 'acordarme']
```

Las cadenas tienen los métodos: *strip(), rstrip(), lstrip()*. Estos métodos devuelven una nueva cadena eliminando los espacios en blanco del principio y del final, sólo del final o sólo del principio respectivamente. Los espacios en blanco eliminados incluyen también a los tabuladores, \t, y saltos de línea, \n, return, \r, entre otros.

En una cadena puede ser reemplazada una subcadena por otra con el método *replace*.

```
txt = "I like bananas"
x = txt.replace("bananas", "apples")
```

Con la llamada anterior se reemplazan todas las ocurrencias de una subcadena por otra. Con un parámetro adicional podemos solamente reemplazar la n primeras ocurrencias.

```
x = txt.replace("bananas", "apples",2)
```

El método *format* de las cadenas devuelve una versión *formateada* de la misma al combinarla con algunos parámetros. Entre otras cosas, nos permite intercalar en una cadena los resultados de diversas expresiones, eligiendo el orden o el formato en que se representan dichos resultados. Esta flexibilidad hace de *format* una función adecuada, que podemos usar junto a *print*, para mostrar mensajes más o menos complejos.

El método *format* se aplica sobre una cadena con texto que tiene intercaladas parejas de llaves que pueden incluir un número. Devolverá una cadena en la que se sustituirán las llaves por los resultados de evaluar las expresiones que reciban como parámetros.

Dentro de las llaves apararece un número que se refiere al parámetro en esa posición. También se puede indicar el tipo de dato que se espera (entero *d*, real *f*, cadena *s*), el número de decimales si es un real, la anchura del campo de salida si es un entero o una cadena.

```
a:int = 2
b:float = 4.567
i: int = 5
print('El resultado de {0} multiplicado por {1:.1f} \
       es {2}'.format(a, b, a*b))
print('El resultado de {0} entre {1} es {2}, y el de {1} \
       entre {0} es {3}'.format(a, b, a/b, b/a))
print('El resultado de {0} entre {1} es {2:.2f}, y el \
       de {1} entre {0} es{3}'.format(a, b, a/b, b/a))
print('{0} {1:2d} {2:3d}'.format(i, i*i, i*i*i))
print('{0:03d} {1:03d} {2:03d}'.format(i, i*i, i*i*i))
```

También podemos indicar como completar el espacio reservado para un dato. Así *{0:03d}* indica que el parámetro en posición cero, el primero, será entero, se ubicará en un espacio de tres caracteres que se completará con ceros si el entero es más pequeño. Con *{2:.2f}* que el parámetro en posición dos, el tercero, es un número real y se imprimirá con dos decimales.

En el lugar de un número para referenciar un parámetro en la posición correspondiente podmeos usar un identificador.

```
print('Si x es igual a {x:d} e y es igual a {y:.2f}, \
         entonces la inecuación {t:s} es {inecuacion}' \
         .format(x=a, y=b, t = 'x < (y * 2)',
          inecuacion = a<(b*2)))
```

Python ofrece la posibilidad de usar *f-string*. Estas son cadenas de caracteres con el prefijo f o F. Estas cadenas pueden incluir pares de llaves con expresiones cuyo valor puede ser sustituido en tiempo de ejecución. Dentro de los pares de llaves también podemos especificar detalles del tipo de dato que se espera y la anchura del campo entre otros.

```
nombre: str = 'Juan'
telefono: int = 678900123
altura: float = 182.3
print(f'{nombre*2:10s} ==> {telefono:10d}  => {altura:.2f}')
```

Otra alternativa para formatear cadenas es el operador %. Este operador compone una cadena de texto donde aparecen %, posiblemente seguidos por especificadores de tipo como antes, con una tupla de parámetros. El resultado es la sustitución de cada % junto con la especificación de tipo por el valor del parámetro correspondiente.

```
print('fruit:%10s, price:%8.2f' % ('apple', 6.056))
```

Tipos numéricos

Existen tres tipos que permiten trabajar con números en Python: enteros (*int*), reales (*float*) y complejos (*complex*). También podemos usar el tipo *Fraction*.

Los *literales enteros* se escriben tal como estamos acostumbrados, mediante una secuencia de dígitos. Por ejemplo:

```
2018
```

Si escribimos el punto decimal (.), entonces diremos que se trata de un *literal real*:

```
3.14159
```

Las operaciones disponibles sobre valores de estos tipos incluyen a las *operaciones aritméticas* (suma, resta, multiplicación...), las *operaciones relacionales* (mayor que, menor que...), y algunas otras como el valor absoluto. Algunas operaciones se representan mediante un operador (por ejemplo, se usa el operador + para la suma), mientras que otras se representan mediante una llamada a función (por ejemplo, se usa la función predefinida *abs* para obtener el valor absoluto de un número).

A continuación, se muestran ejemplos que deben ser autoexplicativos. Empezamos por las *operaciones aritméticas*, que son aquellas en las que tanto los operandos como el resultado son numéricos:

```
print(3 + 6) # suma
print(3 - 4) # resta
print(3 * 4) # producto
print(3 / 4) # división
print(3 // 4) # división entera: sin decimales
print(3 % 4) # resto de la división entera
print(- 3) # opuesto
print(abs(-3)) # valor absoluto
print(3 ** 4) # potencia
```

Las funciones matemáticas usuales se importan del módulo math. Existen más funciones que se pueden encontrar en la documentación de Python. Con ellas podemos diseñar nuevas funciones. Ejemplos de funciones

```
from math import pi, sqrt, sin, cos, atan2, degrees, sqrt,acos

def area_circulo(radio:float) -> float:
    return pi*radio**2

def longitud_circunferencia(radio:float) -> float:
    return 2*pi*radio

print(sqrt(3.4))
print("El área del círculo es {0:.2f}" \
      .format(area_circulo(1.5)))
print("El área del círculo es {0:.2f}" \
      .format(longitud_circunferencia (2.5)))
```

En los ejemplos anteriores, hemos utilizado *literales* de tipo *cadena de caracteres* (por ejemplo, *"Hola, mundo!"*) y literales de tipo *entero* (por ejemplo, *3*, *5* y *8*). Hay más tipos de literales, por ejemplo literales de tipo *real* o de tipo *lógico*. El tipo cadena de caracteres en Python es *str*, el entero *int*, el real *float* y el lógico *bool*.

```
umbral:float = 100.0
area_superior_umbral:bool = area > umbral
print("¿Es el área superior al umbral?{0:s}" \
             .format(str(area_superior_umbral)))
```

Los valores de los tipos de datos deben convertirse a cadenas de caracteres para poder imprimirse en la pantalla. En general esta conversión se hace de manera automática pero cuando queremos hacerlo explícitamente usamos la función *str* aplicada al valor que queramos

convertir. Cuando llamamos a la función *print* se aplica automáticamente la función *str* al parámetro.

La operación contraria, convertir una cadena de caracteres en un valor de un tipo, se denomina *parsing.* Cada tipo viene dotado de una operación de este tipo. Para los tipos anteriores las operaciones de parsing son los nombres de los tipos: *int*, *float*. Esta operación de parsing la debemos hacer cuando leemos datos de un fichero o de la consola. La función *bool* aplicada a una cadena resulta en *True* si la cadena no está vacía y *False* en caso contrario.

Los tipos de datos numéricos también se puede convertir entre sí ya sea explícita o implícitamente. Cuando operamos un entero con un real se convierten ambos a reales y luego se operan. Para convertirlos explícitamente se usa *int* o *float*.

```
a: float = float('3.45')
b: int = int('45')
print('34'+str(a+b))
print(a/b)
print(int(a)//b)
print(bool(""))
```

Hay otros tipos que son usados en matemáticas como el tipo *Fraction* y el tipo *Complex.*

El tipo de datos Fraction representa una fracción y Complex un número complejo. Los valores de Fraction y Complex pueden ser operados con los operadores aritméticos usuales: +, -, *, / ... Los valores de Fraction se pueden comparar con los operares <, >, <=, >=. Los valores de complex no. Fraction tiene, entre otras, las propiedades numerator, denominator. Complex tiene las propiedades real, imag que representan respectivamente la parte real y la imaginaria. El ángulo y el módulo pueden ser calculados mediante las funciones *phase, abs*.

Las fracciones se representan como cadenas de caracteres en la forma *numerador/denominador*.

```
from fractions import Fraction
from cmath import phase

f1: Fraction = Fraction(2,3)
f2: Fraction = Fraction(2,4)
f3: Fraction = Fraction('5/7')
print('result = {}'.format(f1+f2+f3))
print(f1<f2)
```

Los números complejos en la forma *real+imagj*.

```
c1: complex = 3+4j
c2: complex = complex(3,5);
c3: complex = complex('6+7j')
print(c1*c2+c3)
print(c1.real)
print(c1.imag)
print(phase(c1)) #argumento
print(abs(c1))  #modulo
print('result = {}'.format(c1))
```

Tipos para manejar fechas

Otros tipos de datos que usaremos a menudo son las fechas y las horas. Estos son los tipos *datetime, date, time y timedelta*. Los valores de datetime tienen entre otras las propiedades hour, month, year y se pueden comparar con los operadores relacionales.

Mediante la función *now* podemos obtener la fecha y hora actual. Los valores del tipo datetime aceptan el operador + y - para sumar o restar días, etc.

El tipo *datetime* representa fechas, horas, minutos, etc., *date* fechas y *time* horas, minutos y segundos.

La conversión a cadenas de caracteres se hace con la función *strptime(valor,formato)* indicando el formato adecuado. El parsing de los valores de datetime se hace con el método *strftime(formato)* usando también el formato adecuado.

En ambos casos el formato puede contener los símbolos: *%Y:* Año *(4 dígitos), %m:* Mes*, %d:* Dia del mes*, %H:* Hora (24 horas*), %M:* Minutos, *%S:* Segundos, *%a:* Dia de la semana abreviado, %A: día de la semana completo, *%b*: Nombre del mes abreviado, *%B:* Nombre del mes completo y otros que se pueden encontrar en la documentación. Podemos ajustar el formato para que los nombres de meses y días de la semana aparezcan en español con *setLocale*.

A partir de un valor de tipo *datetime* podemos obtener otro de tipo *date* (solo la fecha) con el método *date()* u otro de tipo *time* (solo la hora, minutos, ...) con el método *time()*. Con el método *combine* podemos obtener un valor de tipo *datetime* a partir de un *date* y un *time*.

```
from datetime import datetime, date, time, timedelta

actual:datetime = datetime.now()
hora: int = actual.hour
print(actual)
mes:int = actual.month
print(mes)
anyo:int = actual.year
print(anyo)
otro: datetime = datetime(2022,3,27)
print(otro)
print(otro > actual)
```

El tipo *timedelta* representa diferencias de fechas. Podemos construirlo directamente a partir de un número de días, horas, etc. u obtenerlo de la diferencia de dos fechas. Puede operarse con los operadores numéricos usuales y sumarse o restarse a un *datetime*.

```
d1:datetime = datetime.now()
t:timedelta = timedelta(days=43,hours=23,seconds=5)
d2:datetime = d1+t
print(d1.year)
print(d1.month)
print(d1.hour)
print(d1.second)

print((d2-d1).seconds)
d3: date = actual.date()
d4: time = actual.time()
print(d3)
print(d4)
```

En el ejemplo anterior, cuando escribimos *actual.month* estamos accediendo a una *propiedad*, llamada *month*, del objeto *actual*.

```
dt_str = '2018-06-29 17:08:00'
dt_obj = datetime.strptime(dt_str, '%Y-%m-%d %H:%M:%S')
print(dt_obj.strftime('%d/m/%Y -- %H:%M:%S'))
print(dt_obj.strftime('%A %d de %B del %Y -- %H:%M:%S'))

import locale

locale.setlocale(locale.LC_TIME, 'es_ES')
print(dt_obj.strftime('%A %d de %B del %Y -- %H:%M:%S'))

print(str_date(d1))
d3 = parse_date('2/2/2002')
t1 = parse_time('00:00:30','%H:%M:%S')
print(str_time(t1))
t2 = parse_time('00:00:40','%H:%M:%S')
```

Listas y tuplas

Cuando necesitamos almacenar varios datos en una sola variable, utilizamos literales de tipo *lista*:

```
temperaturas:list[float] = [28.5, 27.8, 29.5, 32.1, 30.7, \
        25.5, 26.0]
print(temperaturas)
heroes:list[str] = ["Spiderman", "Iron Man", "Lobezno",  \
             "Capitán América", "Hulk"]
print(heroes)
```

O, a veces, de tipo *tupla*, que se escriben igual pero usando paréntesis en lugar de corchetes:

```
usuario: tuple[str,str] = ("Mark", "Lenders")
print(usuario)
```

Ambos, listas y tuplas, se parecen mucho (ya veremos sus diferencias más adelante). Podemos acceder a un elemento concreto, indicando el *índice* de dicho elemento (es decir, la posición que ocupa dentro de la lista o tupla, contando desde la izquierda). Debemos tener en cuenta que la primera posición corresponde al índice 0 y podemos asignar un valor a una casilla cualquiera de una lista indicando su índice.

```
print(temperaturas[0])
print(heroes[4])
print(usuario[1])
temperaturas[1] = 27.0
print(temperaturas)
temperaturas[0] = temperaturas[0] + 0.5
print(temperaturas)
```

También podemos conocer en cualquier momento el número total de elementos dentro de una lista o una tupla mediante la función predefinida *len*:

```
len(temperaturas)
len(usuario)
```

Las tuplas son *inmutables*. No podemos asignar valores a las posiciones de una tupla. Al ejecutar el código siguiente se produce un error.

```
usuario[0] = "Marcos" # Cambiemos el nombre del usuario
```

Decimos que las tuplas son un *tipo inmutable*, lo que significa que una vez que hemos guardado un valor en una variable de este tipo, ya no podremos cambiarlo.

Los elementos de la lista *temperaturas* son números reales. Pero una lista puede contener elementos de cualquier tipo. Decimos que la lista es un tipo genérico. Podemos tener una lista de tuplas.

```
usuarios:list[tuple[str,str]] = [('Mark', \
      'Lenders'),('Oliver','Atom'),('Benji', 'Price')]
print(usuarios)
print(usuarios[2][0])
```

Si queremos añadir un elemento a la lista *temperaturas*, podemos usar el método *append* de las listas.

```
temperaturas.append(29.2)  # añade el valor 29.2 al final \
      de la lista
print(temperaturas)
print(len(temperaturas))
```

Cuando una variable (*temperaturas*) puede ser usada como en el ejemplo anterior, escribiendo un punto (.) y a continuación invocando a una función (*append*), decimos que la variable es un *objeto*, y a la función en cuestión la llamamos *método*. La principal diferencia entre un método y una función es que el primero sólo se puede llamar si tenemos previamente un objeto del tipo adecuado. Por ejemplo, el método *append* sólo puede ser invocado sobre un objeto de tipo lista.

Si intentamos llamar a *append* como si fuese una función, sin estar aplicada a un objeto, veremos que aparece un error.

```
append(29.2)
```

Podemos ver todos los métodos que tiene un objeto mediante la función predefinida *dir*:

```
dir(temperaturas)
```

Y si quieres una descripción más detallada de cada método, puedes usar la función *help*, que ya hemos usado anteriormente.

```
help(temperaturas)
```

Diccionarios y conjuntos

Existen más tipos, como las listas y las tuplas, que permiten almacenar agregados de datos. Uno el tipo *conjunto* y otro el tipo *diccionario*. Un conjunto es un agregado de elementos que no se repiten y tampoco se puede indexar. Es decir buscar el valor correspondiente a un índice entero que se mueve en un rango. Escribimos sus valores entre dos llaves.

En un diccionario, cada valor almacenado se asocia a una clave, de manera que para acceder a los *valores* se utilizan dichas *claves* (de forma parecida a como para acceder a los valores de una lista se utilizan los índices).

Se puede pensar un diccionario como si se tratara de una tabla con dos columnas. Por ejemplo, un diccionario podría almacenar las temperaturas medias de las capitales andaluzas, indexando dichos valores mediante los nombres de las capitales:

Clave	Valor
"Almería"	19.9
"Cádiz"	19.1
"Córdoba"	19.1
"Granada"	16.6
"Jaén"	18.2
"Huelva"	19.0
"Málaga"	19.8
"Sevilla"	20.0

Lo cual se escribiría en Python de la siguiente manera:

```
datos: set[float] = \
             {28.5, 27.8, 29.5, 32.1, 30.7, 25.5, 26.0, 26.0}
print(datos)
temperatura_media:Dict[str,float] = {"Almería": 19.9, \
       "Cádiz": 19.1, "Córdoba": 19.1, "Granada": 16.6,\
       "Jaén": 18.2, "Huelva": 19.0, "Málaga": 19.8, \
       "Sevilla": 20.0}
print(temperatura_media)
```

Ahora podemos trabajar con los datos del diccionario de forma parecida a las listas, aunque usando las claves para acceder a los valores, en lugar de los índices. Las claves de un diccionario forman un conjunto.

```
print("Temperatura anterior:  \
       {}".format(temperatura_media['Sevilla']))
temperatura_media['Sevilla'] += 1.0
print("Temperatura actual: \
       {}".format(temperatura_media['Sevilla']))
```

Clases y objetos

El tipo List ya viene predefinido en Python. Sus valores son objetos y estos objetos tienen métodos que se llaman con el operador **.** que separa un objeto y un método.

Si queremos definir un tipo nuevo debemos diseñar una clase. En esta se especifican las propiedades del objeto, sus métodos y el constructor.

Veamos como diseñar el tipo *Vector2D* cuyas propiedades son *x, y, modulo, ángulo, multiply_escalar* con el significado que usamos en matemáticas.

```
from dataclasses import dataclass

@dataclass(frozen=True,order=True)
class Vector2D:
    x: float
    y: float

    @property
    def modulo(self) -> float:
        return sqrt(self.x*self.x+self.y*self.y)

    @property
    def angulo(self) -> float:
        return atan2(self.y,self.x)

    def multiply_escalar(self,v:Vector2D) -> float:
        return self.x*v.x+self.y*v.y
```

Como podemos ver una clase está definida con la palabra reservada *class*. Algunas propiedades como *x, y* se escriben detrás del nombre de la clase. Estas las llamamos propiedades básicas. Otras propiedades *modulo, angulo* se calculan a partir de las básicas y se implementan como un método con el decorador *property*. La clase puede tener, además, otros métodos con parámetros como *multiply_escalar*.

En una clase las variables básicas y los parámetros de los métodos tienen tipos proporcionados por Python o definidos previamente. Los métodos pueden devolver un resultado de un tipo dado.

Las clases pueden tener un decorador *@dataclass(frozen=True,order=True)*. Los detalles de este decorador los veremos más adelante pero esencialmente queremos decir es que el *tipo es inmutable*, tiene definido un *orden natural* y un *constructor* que es un mecanismo para crear objetos. Cuando indicamos que un tipo tiene *orden natural* queremos decir que sus valores pueden ser comparados con los operadores relacionales. Que el tipo es *inmutable* quiere decir que no se pueden modificar las propiedades de un objeto una vez creado.

Creamos nuevos objetos con el *constructor* que es el nombre de la clase seguido de los valores de las propiedades básicas.

Una vez definida la clase podemos crear objetos y construir expresiones a partir de sus propiedades y métodos. Los valores de las propiedades, básicas o derivadas, de un objeto se obtienen concatenando el identificador del objeto, el operador **.** y la propiedad. El resto de los métodos llevarán, adicionalmente, los parámetros reales.

```
v1: Vector2D = Vector2D(3.4,5.6)
v2: Vector2D = Vector2D(-3.4,7.6)
print(v1.modulo)
print(v1.angulo)
print(v1.multiply_escalar(v2))
print(v1.x)
```

Internamente, en la definición de la clase, para acceder a las propiedades y métodos usamos *self* junto con la propiedad o el método separada por **.**

Control del flujo de ejecución

Podemos utilizar la instrucción *for* para recorrer los elementos de una lista o una tupla. Llamamos a esto un *bucle*.

```
for t in temperaturas:
    print("Temperatura:", t)
```

Se puede leer así: para cada valor t en la lista "temperaturas" imprimir t.

En el código anterior, la sentencia que aparece indentada se ejecuta una vez para cada elemento de la lista *temperaturas*. En cada ejecución, la variable *t* almacena un elemento distinto de la lista, comenzando por el primero y pasando por cada elemento secuencialmente.

Hemos usado la función predefinida *print* con varios parámetros, en cuyo caso mostrará cada uno de ellos por pantalla.

Veamos otro ejemplo de bucle donde conseguimos sumar los elementos de una lista:

```
suma = 0
for temperatura in temperaturas:
    suma = suma + temperatura
print("La temperatura media en la semana ha sido {} grados” \
       .format(suma/len(temperaturas))
```

Aunque realmente no hace falta hacer un bucle para sumar los elementos de una lista, ya que esto mismo podemos hacerlo en Python con la función predefinida *sum*.

```
print("La temperatura media en la semana ha sido", \
       sum(temperaturas) / len(temperaturas), "grados.")
```

La instrucción *for* es una instrucción de *control del flujo de ejecución*. Permite que los elementos de una secuencia se ejecuten uno tras otro. Otra instrucción que nos sirve para definir el flujo de ejecución es la instrucción *if*. Veamos un ejemplo:

```
def saluda_hora(nombre:str, hora:int)->None:
    if hora < 12:
       print("Buenos días, " + nombre)
    elif hora < 21:
        print("Buenas tardes, " + nombre)
    else:
        print("Buenas noches, " + nombre)

saluda_hora("Antonio", 11)
saluda_hora("Antonio", 16)
saluda_hora("Antonio", 23)
```

En la función saluda_hora según sea el valor de parámetro hora se imprime un saludo u otro. Si hora es menor que 12 se imprime uno, si es menor que 21 otro diferente y uno distinto en otro caso.

Las *palabras clave* que se usan son *if, elif* y *else*. A continuación de las dos primeras se escribe una expresión lógica (es decir, una expresión cuyo resultado es de tipo lógico: verdadero o falso). Las instrucciones que aparecen tras la palabra *if* sólo se ejecutan si la expresión correspondiente se evalúa como verdadera. Las instrucciones que aparecen tras la palabra *elif* sólo se ejecutan si no se han ejecutado las anteriores, y si la expresión correspondiente se evalúa como verdadera.

Las instrucciones que aparecen tras la palabra *else* sólo se ejecutan si no se han ejecutado ninguna de las anteriores.

Veamos un ejemplo

1. *Escribe una función con un bucle for para sumar los elementos de una lista de enteros que sean pares*

```
r : list[int] = [1,-4,5,78]

def suma(ls:List[int])->int:
    s:int = 0
    for e in ls:
        if e%2 == 0:
            s = s+e
    return s

suma(r)
```

Módulos y paquetes

Además de las funciones predefinidas, hay muchas otras cosas que se pueden hacer con Python sin necesidad de implementar nosotros ninguna función ni tener que descargar o instalar nada en el ordenador. Eso sí, para poder hacer uso de estas funcionalidades incluidas en Python, es necesario *importar* el módulo o módulos donde están implementadas. Un *módulo* es cada uno de los ficheros con extensión .py en los que están escritos los programas en Python. Los módulos están agrupados en *paquetes*, que son parecidos a las carpetas del explorador de archivos de Windows.

Por ejemplo, si queremos generar números aleatorios, podemos usar el módulo *random*. Antes de hacerlo, es necesario importarlo. Por ejemplo:

```
import random

print(random.randint(1, 10))
```

Para llamar a la función randint, que está definida dentro del módulo random, debemos escribir *random.randint(...)*. Si vamos a usar muy a menudo la función en nuestro código, también podemos importarla así:

```
from random import randint

print(randint(1, 10))
```

De esta manera ya no es necesario indicar la ubicación de la función cada vez que la llamemos. Aunque es más cómodo, no es del todo recomendable, como veremos más adelante.

¿Y si queremos generar 20 números aleatorios entre 1 y 10? Podemos hacerlo con la instrucción *for*, la misma que utilizamos antes para recorrer los elementos de una lista y la función *range*.

```
numeros = []
for i in range(20):
    numeros.append(randint(1, 10))
print(numeros)
```

La función predefinida *range(20)* devuelve los números enteros del 0 al 19. En el siguiente ejemplo, usamos el módulo datetime, que nos permite, entre otras cosas, saber la hora actual en que se está ejecutando el programa.

```
from datetime import datetime
hora_actual = datetime.now()

saluda_hora("Juan", hora_actual.hour)
```

Ficheros

Es muy habitual trabajar con **ficheros** (o "archivos", que es un sinónimo). Aunque los ficheros pueden contener todo tipo de datos, nosotros trabajaremos con ficheros de texto. Al igual que se hace con un documento en el ordenador, para trabajar con un fichero en Python primero tenemos que abrirlo; después podemos leer o escribir datos en

el mismo, y finalmente hemos de cerrarlo. Aquí tienes un pequeño ejemplo que abre un fichero de texto y muestra su contenido en pantalla:

```
with open("prueba.txt") as f:
    for linea in f:
        print(linea)
```

Hay que asegurarse que el archivo *prueba.txt* está en la misma carpeta desde la que estás ejecutando este notebook. Si no está se ocasiona un error al ser ejecutado.

También es posible que al ejecutar Python te devuelva un *error de codificación*, o que te muestre el contenido del fichero de texto con algunos errores (algunos caracteres aparecen de manera extraña). Esto ocurre porque existen múltiples maneras de representar los caracteres de un fichero de texto en notación binaria. Es lo que se conoce como *codificación de caracteres* o *enconding* en inglés. Cada sistema operativo tiene su codificación por defecto, de manera que las líneas que acabamos de escribir para leer un fichero de texto dan por hecho que el fichero de texto está expresado en dicha codificación por defecto (la cuál puede no ser la correcta en tu caso). Para evitar estos problemas, cuando trabajemos con ficheros de texto, podemos indicar el tipo de codificación. La más usual es *UTF-8*. Esto se consigue añadiendo un parámetro a la función open:

```
with open("prueba.txt", encoding='utf-8') as f:
    for linea in f:
        print(linea)
```

Veremos más adelante como averiguar el encoding de un fichero podemos hacerlo con la función.

Función principal

Cuando escribimos nuestro código en un fichero de texto con extensión ".py", estamos definiendo un módulo. Otros programadores podrán importar nuestro módulo, y hacer uso en sus programas de las funciones

que hayamos definido en el mismo, o de las variables y objetos que aparezcan definidos sin indentar.

Sin embargo, algunas veces queremos que el código que hemos escrito sea ejecutado por un usuario; es decir, estamos definiendo un *programa ejecutable*. En este caso, el usuario puede ejecutar el programa con el siguiente comando, escrito desde el *terminal* o *ventana de comandos* del sistema operativo:

```
python nombre_fichero.py
```

Cuando un programa es ejecutable, debe existir un *punto de entrada*. Un punto de entrada es un conjunto de instrucciones que se ejecutarán cuando se lance el programa. En Python, estas instrucciones se deben escribir dentro de un bloque *if* como el que se muestra a continuación:

```
if __name__ == '__main__':
    hora_actual = datetime.now()
    saluda_hora("Fermín", hora_actual.hour)
```

A veces, el conjunto de instrucciones que definen el punto de entrada se escribe dentro de una función, a la que se le suele dar el nombre main. Esto en Python no es obligatorio, aunque sí lo es en otros lenguajes de programación. Si queremos hacerlo así, entonces el programa ejecutable anterior quedaría como sigue (se muestra a continuación el programa completo, con las sentencias import y las definiciones de funciones necesarias):

```
from datetime import datetime

def main():
    hora_actual = datetime.now()
    saluda_hora("Fermín", hora_actual.hour)

if __name__ == '__main__':
    main()
```

Variables

Una variable es un elemento de un programa que permite almacenar un valor en un momento de la ejecución, y utilizarlo en un momento posterior. Cada variable tiene un nombre, su *identificador*, un *tipo* y podemos darle algún valor inicial.

Cuando solo indicamos el nombre y el tipo decimos que la variable está *declarada*. Si, además, tiene un valor inicial decimos que está *definida*.

Es habitual confundirse al escribir el nombre de una variable existente en el programa o acceder a una variable que no ha sido declarada. Si esto ocurre Python produce un error.

```
distancia:float
nombre:str = "Augustino"
edad:int = 19
peso:float = 69.4
altura: float = 1.70
```

Podemos obtener el valor asignado previamente a una variable. Decimos que la estamos *usando*. Una variable que no está definida no se puede usar.

```
print(nombre)
print(edad)
print(peso)
print(altura)
print(distancia)
```

Aunque no es obligatorio, si en algún momento no necesitamos más una variable, podemos eliminarla de la memoria:

```
del(edad)
print(edad)
```

Normas para la construcción de nombres de variables

Podemos usar los nombres que queramos para nuestras variables, siempre que cumplamos las siguientes reglas:

- Sólo podemos usar letras, números y la barra baja (_). No se pueden usar espacios.
- El identificador debe comenzar por una letra o por la barra baja.
- No se pueden usar determinadas palabras clave (keywords) que Python usa como instrucciones (por ejemplo, def o if) o como literales (por ejemplo, True). Aunque Python lo permite, tampoco es apropiado usar nombres de funciones predefinidas (por ejemplo, print).
- Los nombres tienen que ser descriptivos de lo que representa la variable.

Aquí podemos ver algunos ejemplos de nombres incorrectos de variables; podemos observar los errores generados por Python.

```
4edad = 10
if = 20
True = 15
```

Se puede consultar todas las palabras claves (*keywords*) existentes en Python de la siguiente forma:

```
import keyword
print(keyword.kwlist)
```

Asignaciones

Si escribimos una instrucción formada por el nombre de una variable, que ya hemos definido anteriormente, el signo igual y una expresión, estaremos sustituyendo el valor almacenado en la variable en cuestión por el valor de la expresión. Llamamos a esta instrucción *asignación*. Si lo estimamos oportuno podemos indicar el tipo de la variable.

Por ejemplo, podemos hacer:

```
nombre: str = "Bonifacio" # el valor anterior de la variable \
                    se pierde
print(nombre)
```

En Python es posible hacer *asignaciones múltiples*, lo que permite asignar valores a varias variables en una única instrucción:

```
edad, peso = 21, 73.2
print(edad)
print(peso)
```

Las asignaciones múltiples se pueden usar para intercambiar los valores de dos variables. Por ejemplo:

```
peso, altura = altura, peso
print(peso)
print(altura)
```

Puedes consultar todas las palabras claves (*keywords*) existentes en Python de la siguiente forma:

```
import keyword
print(keyword.kwlist)
```

Expresiones

Aunque en los ejemplos anteriores hemos inicializado las variables utilizando un literal de algún tipo, esta es sólo una de las *expresiones* que podemos emplear. Una expresión puede ser cualquier de las siguientes cosas:

- Un literal.
- Una variable.
- Un operador junto a sus operandos, cada uno de los cuales puede ser a su vez una expresión.
- Una llamada a una función o a un método, siempre que devuelvan algo; cada uno de los parámetros de la invocación a la función o al método es a su vez una expresión.
- Unos paréntesis envolviendo a otra expresión.
- Un objeto, o el nombre de un tipo, el operador . y las llamadas a uno de sus métodos o propiedades

La definición anterior es recursiva: por ejemplo, los operandos de una operación pueden ser a su vez expresiones. Esto hace que podamos tener expresiones tan largas como quieras imaginar (aunque por regla general intentaremos que no sean *demasiado* largas, pues eso las hace más difíciles de leer y entender).

Una asignación está bien formada si la expresión lo está. En Python una variable puede cambiar de tipo. Por lo tanto después de una asignación la variable contendrá un valor del tipo de la expresión.

Si se ejecuta cada trozo de código mostrado, obtendrás el *resultado* de la expresión. Decimos que la expresión es del *tipo* correspondiente al resultado de esta. Podemos llamar a la función predefinida *type* pasándole como parámetro cada una de las expresiones siguientes: al ejecutar, obtendrás el tipo de la expresión. Una expresión puede ser:

Un literal

```
print(39)
```

Un operador junto a sus operandos

```
print(edad + 18)
```

Cada operando es a su vez una expresión, que puede estar formada por otros operadores y operandos

```
print(edad + 18 < 30)
print(type(edad + 18 < 30))
```

Una llamada a función donde el parámetro, a su vez, es una expresión

```
print(len(nombre * 2))
```

Podemos usar paréntesis para indicar el orden de ejecución de las operaciones en una expresión.

```
print(((len(nombre) - len(apellido)) < 2) and \
      ((edad % 2) != 0))
```

Igualmente podemos usar expresiones en los parámetros de las llamadas a funciones o a métodos. Se evalúan las expresiones antes de proceder a ejecutar la función o método:

```
print("El nombre completo del jugador es " + \
      nombre_completo + ".")
```

Prioridad de las operaciones

En uno de los ejemplos de expresiones hemos utilizado los paréntesis para indicarle a Python en qué orden debe evaluar la expresión. Pero ¿qué ocurre si no empleamos paréntesis y la expresión contiene varios operadores y/o llamadas a funciones?

En este caso, Python decide el orden según la *prioridad de las operaciones*. En el caso de los operadores aritméticos la suma tiene menor prioridad que la multiplicación; por tanto, en la expresión *3 + 5 * 8* primero se evalúa *5 * 8* y posteriormente se evalúa *3 + 40*.

En el caso de los operadores relacionales, todos tienen la misma prioridad. Si tenemos expresiones en las que aparezcan operadores de los tres tipos, en primer lugar se evalúan los operadores aritméticos, después los relacionales, y por último los lógicos. Trata de entender cómo se evalúa la siguiente expresión:

```
import math

print(3 + 9 > 9 and 8 > 3)

resultado = 5 + math.sqrt(10 * 10) < 20 - 2
print(resultado)
```

El orden de evaluación de la expresión *5 + math.sqrt(10 * 10) < 20 - 2* es el siguiente:

- Se evalúa el parámetro de la llamada a la función math.sqrt: *10 * 10*, cuyo resultado es *100*.
- Se evalúa la llamada a la función *math.sqrt(100)*, cuyo resultado es *10*.
- Se evalúa la operación *5 + 10*, cuyo resultado es *15*.
- Se evalúa la operación *20 - 2*, cuyo resultado es *18*.
- Por último, se evalúa la operación *15 < 18*, cuyo resultado es *True*.

Como recomendación final, tengamos en cuenta que si en algún momento tenemos dudas de la prioridad de los operadores que estamos usando,

siempre podemos usar los paréntesis para asegurar que estamos escribiendo lo que realmente queremos expresar.

Conversión de tipos

Python tiene un *sistema fuerte de tipos*, lo que en pocas palabras significa que cada literal, variable o expresión que utilicemos tiene asociado un tipo determinado, y que Python nunca va a convertir ese tipo a otro tipo de manera automática. Un entero o real no se convierte automáticamente en cadena de caracteres.

Hay una excepción a esta regla: cuando operamos enteros con reales todos se convierten a reales y se hace posteriormente la operación.

Podemos hacer conversiones forzadas de tipo (*casting*) con funciones que tienen los mismos nombres que los tipos.

Para entender esto, ejecuta el siguiente ejemplo:

```
resultado = 10 * 3.141519 - 19
print(resultado)
print(int(resultado))
print("El resultado del cálculo es " + str(resultado))
```

Como se puede observar, si no se usara str para convertir a cadena resultado se produciría un error (en concreto, un *TypeError*). Lo que nos diría el error en cuestión es que para poder realizar la operación de concatenación de cadenas, que aparece en la expresión *"El resultado del cálculo es " + resultado*, sería necesario que el segundo operando, *resultado*, fuera de tipo cadena (*str*). Esto no es así: *resultado* es de tipo *float*. Algunos lenguajes de programación realizan esta conversión de manera automática, convirtiendo el valor de resultado a una cadena de texto, antes de proceder a evaluar la expresión completa. No es el caso de Python: dado que tenemos un sistema fuerte de tipos, las conversiones de datos deben ser siempre explícitamente escritas por el programador.

Para llevar a cabo una conversión del tipo de una expresión, debemos usar funciones predefinidas cuyos nombres coinciden con los nombres de los tipos básicos que hemos visto hasta ahora: *bool, int, float, str*.

Expresiones bien formadas

Decimos que una expresión está *bien formada* (o también, que es una expresión correcta) cuando se cumple que:

- Los literales que aparecen en la expresión están correctamente escritos según las reglas que hemos visto.
- Las variables que aparecen en la expresión han sido definidas previamente, es decir se les ha asignado un valor, (o importadas mediante la instrucción *import*).
- Los operadores que aparecen en la expresión aparecen aplicados al número correcto de operandos, y los tipos de las expresiones que funcionan como operandos son los adecuados para dichos operadores.
- Las llamadas a funciones o métodos que aparecen en la expresión corresponden a funciones o métodos definidos previamente (o importados mediante la instrucción *import*). Además, el número y tipo de las expresiones utilizadas como parámetros de las llamadas son los esperados por dichas funciones y métodos.
- El operador punto (.) separando un objeto, o el nombre de una clase, de una propiedad o método de la clase a la que pertenece

Una expresión bien formada tiene un tipo definido. Y su valor puede ser calculado si no se producen otros errores de ejecución.

Si una expresión no está bien formada, Python devolverá un error al tratar de ejecutar el código. Por ejemplo, las expresiones siguientes están mal formadas.

```
nombre: str = "Juan"
print(13'2 * 5)
print((nombre[0] + nombre[1]) / 2)
print()"Ajo" * 3.1)
print(abs("-1.2"))
```

Tipos de agregados de datos

En Python existen algunos tipos que son agregados de datos. Estos permiten almacenar en una variable varios valores al mismo tiempo.

Cada uno de estos valores puede tener a su vez su propio tipo (es decir, puedo guardar en una única variable dos valores de tipo entero y un valor de tipo cadena, por ejemplo).

Entre otros, tenemos disponemos en Python de estos tipos de agregados de datos.

Tuplas

El tipo tupla (*tuple*) permite almacenar datos de cualquier tipo, en un orden determinado. Los literales se escriben concatenando los datos que se desea que estén incluidos en la tupla, separados por comas, y envolviéndolo todo con unos paréntesis. Por ejemplo:

```
("Mark", "Lenders", 15)
```

Si guardamos una tupla en una variable, podemos acceder a cada uno de los elementos de la tupla de la siguiente manera:

```
from typing import List, Set, Dict, Tuple

jugador: tuple[str,str,int] = ("Mark", "Lenders", 15)
print("Nombre:", jugador[0])
print("Apellidos:", jugador[1])
print("Edad:", jugador[2])
```

Las tuplas son *inmutables*, lo que significa que una vez que se ha asignado un valor a una variable de tipo tupla ya no podemos cambiar los valores encapsulados en dicha tupla, ni añadir o eliminar elementos. Prueba a ejecutar el siguiente código y observa el error que se produce:

```
jugador[2] = 16
```

Listas

El tipo lista (*list*) permite almacenar datos de cualquier tipo, en un orden determinado, al igual que las tuplas. La principal diferencia es que son *mutables,* es decir, una vez inicializada una variable de tipo lista, es posible cambiar el valor de una posición, añadir nuevos elementos o eliminarlos. Los literales se escriben concatenando los datos que se desea

que estén incluidos en la tupla, separados por comas, y envolviéndolo todo con unos corchetes. Por ejemplo:

```
[32, 36, 35, 36, 32, 33, 34]
```

Aunque al igual que en las tuplas los elementos pueden tener cada uno un tipo distinto, lo más habitual en las listas es que todos los elementos sean de un mismo tipo. Para acceder a los elementos se usan los corchetes, al igual que con las tuplas, con la diferencia de que ahora también podemos asignar nuevos valores a una posición determinada de la lista:

```
temperaturas:List[int] = [32, 36, 35, 36, 32, 33]
print("Temperatura lunes:", temperaturas[0])
temperaturas[1] = 35
print(temperaturas)
```

Conjuntos

El tipo conjunto (*set*) permite almacenar datos de cualquier tipo, sin ningún orden determinado, y sin posibilidad de elementos repetidos. Los literales se escriben concatenando los datos que se desea que estén incluidos en el conjunto (da igual el orden en que los escribamos), separados por comas, y envolviéndolo todo con unas llaves. Por ejemplo:

```
{32, 33, 34, 35, 36}
```

Observa lo que ocurre si inicializamos un conjunto con datos repetidos:

```
temperaturas_conjunto: Set[int] = {32,36,35,36,32,33,34}
print(temperaturas_conjunto)
```

Como el orden de los elementos en un conjunto no es relevante, no podemos acceder a dichos elementos usando los corchetes, como hacíamos en las tuplas y listas. Más adelante veremos qué operaciones podemos hacer con los conjuntos. Por ahora basta saber que son *mutables*, al igual que las listas.

Diccionarios

El tipo diccionario (*dict*) permite almacenar datos de cualquier tipo, sin ningún orden determinado. Cada valor almacenado se asocia a una clave, de manera que para acceder a los valores se utilizan dichas claves. Los literales se escriben concatenando las parejas clave-valor mediante comas y envolviéndolo todo mediante llaves; cada una de las parejas se escribe separando la clave y el valor asociado mediante dos puntos. Por ejemplo:

```
{"Almería": 19.9, "Cádiz": 19.1, "Córdoba": 19.1, \
      "Granada": 16.6, "Jaén": 18.2, "Huelva": 19.0, \
      "Málaga": 19.8, "Sevilla": 20.0}
```

Para acceder a un valor, debemos conocer la clave asociada. Los diccionarios son *mutables*. Veamos el siguiente ejemplo de código:

```
temperaturas_por_provincias:Dict[str,float] = \
      {"Almería":19.9,  "Cádiz": 19.1, "Córdoba": 19.1, \
      "Granada": 16.6, "Jaén": 18.2, "Huelva": 19.0, \
      "Málaga": 19.8, "Sevilla": 20.0}
print("Temperatura en Sevilla:",
temperaturas_por_provincias["Sevilla"])
temperaturas_por_provincias["Sevilla"] = 21.0
print(temperaturas_por_provincias)
```

Los diccionarios tienen métodos para consultar sus claves (keys()), sus valores (values()) y sus tuplas clave-valor.

```
print(temperaturas_por_provincias.keys())
print(temperaturas_por_provincias.values())
print(temperaturas_por_provincias.items())
```

Operaciones con agregados de datos

Los valores de un agregado de datos pueden ser a su vez de otro tipo agregado.

Dado que los agregados de datos son objetos, la mayoría de las operaciones con ellos se llevan a cabo mediante métodos. Más adelante haremos un repaso más detallado sobre los métodos disponibles para

cada tipo agregado, pero por ahora veremos cómo realizar las operaciones más básicas.

```
personas:Dict[str,List[int]] = {"Antonio": [1954,1987,1999], \
       "Juan": [1953,2020]}
print(equipos)
```

Añadir un elemento a una lista, un conjunto o un diccionario

```
temperaturas.append(29)
temperaturas_conjunto.add(29)
temperaturas_por_provincias["Badajoz"] = 15.8
```

Eliminar un elemento de una lista, un conjunto o un diccionario

```
del(temperaturas[0])
temperaturas_conjunto.remove(32)
del(temperaturas_por_provincias["Almería"])
```

Concatenar varias tuplas o listas, concatenar consigo misma varias veces, consultar el número de elementos de una tupla, lista, conjunto o diccionario.

```
print(jugador + (1.92, 81.2))
print(temperaturas + temperaturas)
print(temperaturas * 3)
print(len(jugador))
```

Consultar si un elemento forma parte de una tupla, lista, conjunto o diccionario

```
print(39 in temperaturas)
```

Añade un nuevo número a lista.

```
lista1: List[int] = [1, 2, 3, 4, 5]
lista2: List[int] = [-1, -2, -3, -4, -5]

lista1.append(7)
```

Elimina el último elemento de una lista

```
del(lista1[len(lista1)-1])
```

Obtener una nueva lista formada por 3 repeticiones de otra. Mostrar una lista en pantalla y su número de elementos

```
Ls:list[int] = lista1*3+lista2
print(ls)
print(len(ls))
```

Formateo de agregados de datos

Junto al método *format* es conveniente conocer el método *join* de las cadenas de caracteres. El método join une mediante un separador los elementos de un iterable de texto. Es de la forma

```
lista = ['Juan','Antonio','Miguel']
print(','.join(lista))
```

Los métodos format y join pueden ser combinados en una función para formatear listas, conjuntos y diccionarios. Estos métodos necesitan que los elementos de los agregados sean de tipo str. Si no lo son habrá que hacer la transformación previamente. Formas de hacer esto las aprendemos pronto.

```
from typing import List, Set, Dict, Tuple

ls: List[str] = ['1','4','7','9','23','-32','23']
st: Set[str] = {'1','4','7','9','23','-32','23'}
dt: Dict[str,str] = \
      {'a':'1','b':'4','c':'7','d':'9','e':'23','f':'-32'}

print('{0}{1}{2}'.format('{',','.join(ls),'}'))
print('{0}{1}{2}'.format('{',','.join(st),'}'))
print('{0}{1}{2}'.format('','\n'.join(dt),''))
print('{0}{1}{2}'.format('==',';'.join(dt.values()),'=='))
```

Operadores sobre agregados indexables

Entre otros, tenemos los siguientes agregados de datos indexables:

- Listas: agregados de datos *mutables* e *indexables*, habitualmente se usan para representar colecciones ordenadas de objetos homogéneos (aunque no hay problema en que una lista contenga objetos de distintos tipos).
- Cadenas de caracteres: agregados de caracteres *inmutables* e *indexables*, habitualmente se usan para representar cadenas de texto.
- Tuplas: agregados de datos *inmutables* e *indexables*, habitualmente se usan para representar *registros de datos* heterogéneos (aunque no hay problema en que todos los elementos de una tupla sean del mismo tipo).
- Rangos: una secuencia *inmutable* e *implícita* de números.

Todos estos son subtipos de Sequence.

Los *rangos* se construyen con la función predefinida *range*. Si intentamos imprimir un objeto *range* solo obtendremos la información sobre sus límites. Para acceder a todos los elementos debemos forzar la conversión a un objeto, por ejemplo, de tipo lista:

```
rango = range(10)
print(rango)
print(list(rango))
```

Podemos especificar rangos de varias formas:

- Indicando solo un límite superior: se genera un rango desde *0* hasta ese límite menos uno.
- Indicando el límite inferior y el límite superior del rango: se genera un rango desde el límite inferior (incluido) hasta el límite superior (excluido).
- Indicando ambos límites y un *paso*, que determina el incremento de un elemento del rango al siguiente.

- Indicando un paso negativo, en ese caso el límite izquierdo debe ser mayor que el derecho.

```
print(list(range(10)))
print(list(range(10, 20)))
print(list(range(10, 20, 2)))
print(list(range(20, 10, -1)))
r = list(range(20, 10, -1)[1:3])
print({18,19} == set(r))
```

Para acceder a los elementos de un agregado indexable se utilizan los corchetes, indicando la posición que nos interesa. La posición es un número entre *0* (la primera posición) y *tamaño-1* (la última posición). Si intentamos acceder a una posición no existente provocaremos un error, como en la última instrucción de la siguiente celda:

```
rango = range(10,20)
print(lista[0], lista[2])
print(tupla[1], tupla[2])
print(rango[0], rango[1])
print(cadena[0], cadena[1])

print(lista[-1])
print((2, 4, 6, 8)[-1])
print(rango[-2])
print(cadena[-1])
```

Los índices negativos permiten acceder a una secuencia desde el final. El índice *-1* accede a la última posición de una secuencia, el *-2* a la penúltima, y así sucesivamente. Los índices negativos se pueden usar en cualquier tipo de agregado indexado:

Se denomina *unpacking* al proceso de extraer valores desde un agregado indexable y guardarlos en variables independientes. Esto se consigue mediante una asignación en la que en la parte izquierda hay varias variables receptoras.

Podemos aplicar *unpacking* desde listas, tuplas y rangos. El número de elementos de la colección debe coincidir con el número de variables receptoras.

```
a, b, c = [1, 2, 3]
print(a, b, c)
a, b, c = (4, 5, 6)
print(a, b, c)
a, b, c = range(7, 10)
print(a, b, c)

a, b, c = [1, 2, 3]
a, b = b, a
print(a, b)
```

El *unpacking* se utiliza habitualmente en dos situaciones:

- Para recoger los valores devueltos por una función, cuando la función devuelve una tupla. De esta forma podemos utilizar una variable independiente para cada uno de los valores devueltos por la función:

```
# La función busca_alumno_mejor_calificacion devuelve el \
      alumno con mejor nota y su calificacion
alumno, calificacion =  \
      busca_alumno_mejor_calificacion(alumnos)
```

- Para recorrer los elementos de un agregado, cuando dichos elementos son a su vez agregados (generalmente, tuplas):

```
# La lista alumnos_con_notas es de la forma \
      [(alumno1, nota1), (alumno2,nota2), ...]
for alumno, calificacion in alumnos_con_notas:
      ...
```

Hay una serie de operadores comunes a *listas, tuplas, cadenas o rangos.*

En las situaciones posteriores que ilustraremos estos operadores usaremos como base las siguientes listas:

```
estados = ["nublado", "soleado", "ventoso", "nublado",  \
       "lluvioso"]
temperaturas = [23, 23, 28, 29, 25, 24, 20]
cadena2 = 'El tiempo es muy cambiante'
```

El operador *in* evalúa si un determinado elemento *pertenece* a un agregado indexable o no. Sería el equivalente al operador matemático ∈. He aquí algunos ejemplos:

```
# Sobre listas
print("nublado" in estados)

# Sobre tuplas
print(1 in (2, 1, 5))

# Sobre rangos
print(1 in range(10))

# Sobre cadenas
print('o' in cadena2)

# Sobre conjuntos
print(23 in set(temperaturas))
```

El operador *not in* determina si un determinado elemento *no pertenece* a un agregado. Sería el equivalente al operador matemático ∉. He aquí algunos ejemplos:

```
print(("nublado", "ventoso") not in estados)
print(1 not in range(10, 20))
print(1 not in (2, 3, 1))
```

Los operadores aritméticos *+* y * están definidos para listas, tuplas y cadenas (no para rangos). El significado de cada uno de ellos es el siguiente:

- El operador *+*, entre dos agregados indexables del mismo tipo. No se pueden concatenar de tipos distintos (por ejemplo, listas con tuplas o con cadenas).
- El operador *, entre un agregado indexable y un número, lo replica tantas veces como indique el número.

```
# Sobre listas
print(temperaturas + [22])
print(temperaturas * 2)

# Sobre tuplas
print((1, 2) + (3, 4))
print((1, 2) * 2)
```

La prioridad de los operadores es similar a la de los operadores aritméticos. De mayor a menor prioridad, este es el orden en el que se aplican los operadores de manejo de secuencias:

- Operador *
- Operador +
- Operadores *in*, *not in*

```
print([1] + [2] * 3)
print(1 in [2] + [1])
```

La función *len* que calcula el tamaño de un agregado:

```
temperaturas = [23, 23, 28, 29, 25, 24, 20]

print(len(temperaturas))
print(len((2, 4, "seis")))
print(len(range(10, 20)))
```

Los agregados indexables tienen, además, los métodos *count(x)* e *index*(x). El primero cuenta el número de ocurrencias de *x* en el agregado. El segundo el índice de la primera ocurrencia de parámetro.

Slicing

El *slicing* nos permite seleccionar un fragmento de un agregado indexable. Se usa para ello el operador *:* dentro de los corchetes, que nos permite especificar un rango de acceso a los elementos de un agregado indexable. Como en los rangos, se incluye el límite inferior, pero se excluye el límite superior. Se puede aplicar *slicing* sobre listas, tuplas, cadenas y rangos:

```
temperaturas = [23, 23, 28, 29, 25, 24, 20]
cd = 'El horizonte es incierto'
# Sobre listas
print(temperaturas[1:3])
# Sobre tuplas
print((2, 4, 6, 8, 9)[1:3])
# Sobre rangos
print(range(10,20)[3:6])
print(list(range(10,20)[3:6]))
# Sobre cadenas
print(cd[1:-1])
```

No es obligatorio definir ambos límites al especificar un *slicing*. Si no se especifica el límite inferior, se seleccionan los elementos de la secuencia desde el principio. Si no se especifica el límite superior, se seleccionan los elementos de la secuencia hasta el final:

```
print(temperaturas[:2])
print(temperaturas[4:])
print(temperaturas[:])
```

Se puede hacer *slicing* tanto con índices positivos como con índices negativos:

```
print(temperaturas[1:-1])
```

Al igual que ocurría en la definición de rangos, a la hora de especificar un *slicing* se pueden indicar los límites junto con un *paso*. De esta forma, gracias al *paso*, se puede hacer una selección no continua de los elementos de la colección:

```
print(temperaturas[0:7:2])
```

Listas

En muchas ocasiones tenemos varios datos que están relacionados entre sí. Las listas nos permiten almacenar todos esos datos en una misma variable, gracias a lo cual podemos realizar operaciones con todos los datos a la vez y no de uno en uno.

Piensa por ejemplo en las temperaturas previstas para los próximos siete días, los nombres de las capitales europeas o las edades de tus amigos. Si almacenas esos datos una lista, podrás acceder a cada uno de ellos por separado (la temperatura del próximo miércoles, por ejemplo), o realizar operaciones con todos a la vez (por ejemplo, saber cuál es la capital europea que tiene un nombre más largo).

Creación de una lista

Para crear una lista escribimos su nombre y le asignamos un valor, que es una secuencia de elementos separados por comas y encerrados entre corchetes. Si solo ponemos los corchetes, estaremos creando una lista vacía, que más tarde podremos rellenar. También podemos crear una lista vacía invocando a la función list.

```
# Una lista vacía:
lista_vacia = []
otra_lista_vacia = list()

colores = ["cyan", "magenta", "amarillo"]
temperaturas = [25.2, 24.9, 25.2, 26.7, 28.6, 29.5, 29.7]
```

Las listas pueden contener valores repetidos. Por ejemplo, el primer y el tercer elemento de la lista temperaturas tienen el mismo valor, aunque son dos elementos diferentes.

Los elementos de una lista pueden ser de cualquier tipo. Incluso pueden ser tuplas:

```
frecuencia_caracteres = [(23, "e"), (7, "s"), (12, "a")]
```

Podemos incluso crear listas con elementos de tipos diferentes, si bien no es algo muy frecuente:

```
lista_mix = ["Juan", 23, 75.4]
```

Al ser las listas mutables podemos utilizar el acceso a una posición para cambiar un elemento de la lista, o incluso un trozo de la lista, si usamos *slicing*:

```
# Modificación de una posición
temperaturas[3] = 12
print(temperaturas)

# Modificación de un grupo de posiciones
temperaturas[0:3] = [-1, -1, -1]
print(temperaturas)
```

Algunos métodos de las listas

Para añadir un elemento a una lista utilizamos el método *append*. El elemento se añade al final de la lista. Fíjate en su funcionamiento ejecutando el código siguiente y observando cómo cambia la lista:

```
print(colores)
colores.append("negro")
print(colores)
```

Si queremos añadir un elemento en una posición distinta al final, podemos hacerlo mediante el método *insert*, indicando la posición donde queremos añadir elemento como primer parámetro del método:

```
colores.insert(0, "rojo")
print(colores)

colores.insert(1, "verde")
print(colores)
```

La operación contraria a la anterior es la de eliminar un elemento de una lista. Para hacerlo utilizamos la función *del*, a la cual hemos de pasar como parámetro la posición que ocupa en la lista el elemento que queremos eliminar. Observa su funcionamiento en el siguiente ejemplo:

```
print(temperaturas)
del(temperaturas[2])
print(temperaturas)
```

Observamos cómo se elimina el tercer elemento de la lista, el que ocupa la posición 2. El primer elemento permanece, aunque su valor sea el mismo que el que hemos eliminado. Solo se elimina el que ocupa la posición indicada. Lógicamente, los elementos que le siguen ven

modificada su posición: el elemento 3 pasa a ser ahora el 2, el 4 pasa a ser el 3, y así sucesivamente.

También podemos eliminar todos los elementos de la lista mediante el método *clear*:

```
temperaturas.clear()
print(temperaturas)
```

Tuplas

Una tupla es un tipo de secuencia similar a las listas pero que es inmutable. Esto quiere decir que, una vez creada, no podemos añadir, eliminar ni modificar elementos.

Así como las listas se pueden definir por sus elementos colocados entre corchetes y separados por comas, las tuplas se pueden definir por sus elementos colocados entre paréntesis y separados también por comas:

```
t = (1, 2, 3)

t = 4, 5, 6   # Los paréntesis son opcionales...
lista_tuplas = [(1,2), (3,4), (5,6)] # ... salvo que las
tuplas sean elementos de otras secuencias

t = (1)  # Esto NO es una tupla de un elemento
print(type(t))

t = (1,) # Hay que añadir una coma al final para definir
tuplas de un solo elemento
print(type(t))
```

Podemos construir tuplas a partir de otras secuencias usando la función predefinida *tuple*:

```
tupla_pares_menores_20 = tuple(edades)
print(tupla_pares_menores_20)
```

Las tuplas, a diferencia de las listas, sí suelen usarse con elementos no homogéneos; pueden usarse, por ejemplo, para modelar diferentes características de un objeto. Por ejemplo, podemos definir las tuplas

```
persona1 = "John", "Doe", "varón", 23, 1.83, 87.3
persona2 = ("Jane", "Doe", "mujer", 25, 1.62, 64.0)
print(persona1, persona2)
```

Tuplas con nombre

Una de las ventajas de las tuplas es que permiten implementar recorridos muy legibles mediante el uso del *unpacking*, es decir, utilizando una variable para cada campo de la tupla en el bucle *for*. Sin embargo, cuando hay muchos campos, puede ser pesado realizar *unpacking*. La alternativa es utilizar una variable para nombrar a la tupla completa, y acceder a los campos que necesitemos mediante los corchetes. Pero esto obliga a conocer la posición de cada campo dentro de la tupla, y hace nuestro código poco legible.

Una solución más elegante en estos casos es la utilización del tipo *namedtuple*. Para ello, lo primero que hacemos es crearnos un tipo tupla personalizado, poniendo un nombre tanto al tipo de tupla que estamos creando como a cada uno de los campos:

```
from collections import namedtuple

Persona = namedtuple("Persona", "nombre, apellidos, sexo, \
       edad, altura, peso")
# Otra opción es pasar los nombres de los campos como una \
       lista:
# Persona = namedtuple("Persona", \
    ["nombre", "apellidos", "sexo", "edad", "altura", "peso"])
```

Una vez definido nuestro "tipo personalizado de tupla", podemos crear tuplas de esta forma:

```
persona1: Persona = Persona("John", "Doe", "varón", 23, \
        1.83, 87.3)
```

Observa que la tupla se crea igual que antes, indicando cada elemento de esta, separados por comas, todos ellos entre paréntesis. Pero ahora anteponemos el nombre que le hemos dado a nuestra tupla personalizada.

Una vez hecho esto, podemos acceder a los campos indicando sus nombres, de esta forma:

```
print("Nombre y apellidos: {0} \
       {1}".format(persona1.nombre,persona1.apellidos))
print("Edad: {}".format(persona1.edad))
print("Sexo: {}".format(persona1.sexo))

# Se pueden seguir usando los índices para acceder a los \
       elementos
print("Altura:", persona1[4])
```


Paso de parámetros y lambda expresiones

Para que las funciones nos permitan implementar funcionalidades realmente útiles y reutilizables es fundamental contar con un mecanismo de *paso de parámetros*. Los *parámetros* o *argumentos* (*arguments*) de una función son variables que permiten que la función reciba determinados valores con los que llevar a cabo su funcionalidad. En el momento de la invocación a la función, el programador especifica el valor que desea *pasarle* a dichos parámetros, de manera que se lleva a cabo una asignación entre las variables que representan a los parámetros en la función y los valores especificados en la llamada.

Los parámetros tienen tipos que deben especificarse. También el resultado de la función. Pueden tener valores por defecto. Pero si lo tienen deben ir detrás de los parámetros que no lo tengan.

Veamos un ejemplo consistente en un función *imprime_linea* que imprime una línea formada por repeticiones de los caracteres -=. El parámetro *nr* permite indicar el número de repeticiones de los caracteres que forman la línea a imprimir.

```
def imprime_linea(nr:int)->str:
    linea = "-=" * nr
    print(linea)

for i in range(5):
    imprime_linea(40)
```

Hemos añadido un parámetro de nombre *nr*, de manera que la línea que imprime la función estará formada por tantas repeticiones de la cadena *"-="* como indique dicho parámetro. Al haber añadido un parámetro conseguimos que la tarea que lleva a cabo la función sea más genérica, más configurable, por lo que habrá más ocasiones en las que podamos emplear la función. Por tanto, es importante decidir bien los parámetros para conseguir nuestro objetivo de la reutilización del código.

También podríamos hacer que los caracteres que forman la línea puedan ser escogidos en el momento de la invocación, añadiendo un nuevo parámetro:

```
def imprime_linea(nr:int, cadena:str)->str:
    linea = cadena * nr
    print(linea)
```

Veamos otro ejemplo en el que tras definir una función podemos invocarla en el contexto de una expresión.

```
def pvp(precio_sin_iva:float, iva_reducido:bool)->float:
    if iva_reducido:
        precio_con_iva = precio_sin_iva * 1.1
    else:
        precio_con_iva = precio_sin_iva * 1.21
    return precio_con_iva
```

Devuelve el precio de venta al público, pvp, a partir de un precio sin IVA. El parámetro "precio_sin_iva" sirve para indicar el precio sobre el que calcular el pvp. Si el parámetro "iva_reducido" es True, se aplicará el 10%. Si es False, el 21%.

Parámetros formales con valores por defecto

En la última definición de la función *imprime_linea* hemos incorporado parámetros para elegir la cadena concreta a utilizar y el número de repeticiones de esta. Si bien *parametrizar* todos los detalles de una función la hacen más configurable y por tanto también más reutilizable, tiene la desventaja de que dificultamos su utilización, puesto que

obligamos al programador que la quiera utilizar a elegir valores para todos los parámetros. Esto puede solucionarse utilizando *parámetros con valores por defecto* (también llamados *parámetros opcionales*).

Mediante este mecanismo, Python nos permite elegir valores por defecto para algunos de los parámetros formales de una función. De esta forma, cuando la función es invocada, el programador puede optar por ignorar dichos parámetros, en cuyo caso se tomarán los valores por defecto.

Volvamos a definir la función *imprime_linea* usando esta característica:

```
def imprime_linea(rp:int=40, cadena:str="-=")->None:
    linea = cadena * rp
    print(linea)
```

Podemos observar que los valores por defecto se añaden en la cabecera de la función, añadiendo un carácter = detrás del parámetro formal y a continuación su valor por defecto (en este caso no se ponen espacios alrededor del carácter =). Ahora podemos realizar llamadas a la función en las que aparezcan o no valores para los argumentos:

```
imprime_linea()
imprime_linea(20)
imprime_linea(20, ":)")
```

Por supuesto podemos combinar parámetros con valores por defecto y parámetros sin valores por defecto (podemos llamarles *parámetros obligatorios* a estos últimos). En ese caso, siempre deben venir primero los parámetros obligatorios y a continuación los parámetros con valores por defecto. Es decir, nunca puede aparecer un parámetro obligatorio después de un parámetro con valor por defecto.

Parámetros reales por nombre

Al invocar a una función tras su nombre aparecen los parámetros reales. Estos son valores que se asignarán a cada parámetro de la función. Python nos permite especificar los *parámetros reales* de dos maneras distintas:

- Mediante *parámetros posicionales*, esto es, pasando una lista de valores separados por comas, de manera que esos valores se asignan a los parámetros formales de la función en el orden en que estos aparecen en la cabecera de la función. Este es el método que hemos utilizado hasta ahora.
- Mediante *parámetros por nombre*, esto es, indicando para cada parámetro su nombre y el valor que queremos asignarle. De esta manera, no es necesario que respetemos el orden en que dichos parámetros están especificados en la cabecera de la función.

Veamos algunos ejemplos de llamadas a función usando parámetros por nombre:

```
imprime_linea(10, cadena=":;")
imprime_linea(rp=10)
imprime_linea(cadena=":;")
```

En la primera llamada estamos especificando valores para todos los parámetros de la función. En este caso, usar parámetros por nombre realmente no ayuda mucho, y de hecho hace que la llamada sea más larga de escribir. Por ello en la mayoría de las ocasiones optaremos por escribir los parámetros posicionales, sin indicar el nombre.

Sin embargo, cuando los parámetros por nombre se usan para indicar sólo algunos valores de parámetros con valores por defecto, es cuando realmente son prácticos. Por ejemplo, la llamada *imprime_linea(cadena=":;")* indica un valor para el parámetro *cadena*, obviando el valor para *rp*. Esto no podría hacerse mediante una llamada con parámetros posicionales, puesto que el parámetro *rp* es el primero que aparece.

En general, se suele proceder de la siguiente manera:

- Cuando una función tiene una serie de parámetros sin valores por defecto, se utilizan llamadas con parámetros posicionales (es decir, sin indicar el nombre de cada parámetro en la llamada).
- Si además la función tiene parámetros con valores por defecto, el programador utiliza en la llamada parámetros con nombre para

indicar únicamente los valores que quiere especificar, entendiendo que el resto de los parámetros se quedan con su valor por defecto.

Definición de funciones sin nombre

Una forma alternativa de definir funciones es usar *funciones sin nombre* (también llamadas *funciones anónimas, funciones lambda o expresiones lambda*). Se trata de una manera de describir funciones sencillas mediante una expresión, sin necesidad de definir la función.

Hay veces que usamos funciones cuyo cuerpo es muy sencillo.

```
def multiplica_por_dos(n:float)->float:
    return 2 * n

print(multiplica_por_dos(45))
```

Mediante unas expresiones *lambda* podemos inicializar una variable cuyo tipo, en el caso concreto anterior, será *Callable[[int],int]*.

```
from typing import Callable

multiplica_por_dos: Callable[[int],int] = lambda n: 2 * n
print(multiplica_por_dos(45))
```

La expresión lambda del ejemplo anterior es:

```
lambda n: 2 * n
```

La cual está definiendo una función que toma un parámetro *n* y devuelve el resultado de la expresión *2 * n*.

Se pueden definir funciones sin nombre de más de un parámetro; por ejemplo la siguiente función lambda:

```
lambda n, m: n + m
```

que representa una función que recibe dos parámetros y devuelve su suma.

Funciones como parámetros

Algunas veces queremos diseñar funciones genéricas y necesitamos pasar como parámetro una función que aún no tenemos definida. Por ejemplo:

- Queremos diseñar la función *transforma* que realice una transformación sencilla sobre los objetos de un iterable.
- Queremos diseñar la función *filtra* que realice filtre los objetos de un iterable.

En ambos casos necesitamos pasar como parámetro una función o un predicado. A diferencia de otros lenguajes de programación, en Python las funciones son un tipo más, como lo son el tipo entero, el tipo real o el tipo lista, entre otros. Que sea "un tipo más" (más formalmente se dice que las funciones son objetos de primera clase) significa que puedo declarar variables de tipo función, hacer asignaciones a esas variables, e incluso usar parámetros de tipo función.

Por ejemplo, en el siguiente código definimos un parámetro llamada *transformación* al que le asignamos distintas funciones mediante lambda expresiones o nombres de funciones ya definidas.

```
from typing import Callable

def transforma(ls:list[float],t: Callable[[float],float])
              ->list[float]:
    lt = []
    for elemento in ls:
        lt.append(t(elemento))
    return lt
```

La función *transforma* toma una lista como parámetro y devuelve otra lista formada por los resultados de aplicar una función de transformación a cada unuo de los elementos de una lista de entrada.

El parámetro *t* debe ser una función que reciba un único parámetro y devuelva algún valor.

La función *filtra* devuelve una lista formada por los resultados de aplicar una filtro a los elementos de una lista de entrada. El parámetro *f* debe ser predicado.

```
def filtra(ls:list[float],f:Callable[[int],bool])->list[int]:
    lf = []
    for elemento in ls:
        if filtro(elemento):
            lf.append(elemento)
    return lf
```

Podemos comprobar el funcionamineto de las funciones anteriores pasando distintas funciones en su segundo parámetro.

```
import math
ls = [1, 2, 3, 4, 5]
print(transforma(ls, math.sqrt))
print(transforma(ls, math.sin))
print(transforma(ls, lambda y: y*y))
print(filtra(ls,lambda x:x%2==0))
```

Entrada y salida estándar

Funciones input y print

Por regla general, cuando ejecutamos un programa en Python llamamos *entrada estándar* al teclado de nuestro ordenador, y *salida estándar* a la pantalla. Podemos leer datos desde el teclado mediante la función *input*, y escribir en la pantalla mediante la función *print*. Hemos de tener en cuenta que la entrada mediante input será siempre de tipo cadena de caracteres por lo que habrá que cambiarla posteriormente al tipo adecuado.

```
print("==== Cálculo de una potencia =====")
base = int(input("Introduzca un número entero (base):"))
exponente = int(input("Introduzca un número entero \
       (exponente):"))
```

La función *input* recibe opcionalmente un mensaje, que es mostrado al usuario para a continuación esperar que introduzca un texto. La ejecución del programa "se espera" en este punto, hasta que el usuario introduce el texto y pulsa la tecla *enter*. La función *input* devuelve el texto introducido por el usuario (excluyendo la pulsación de la tecla *enter*, que no aparece en la cadena devuelta). Si en nuestro programa estábamos esperando un dato numérico, en lugar de una cadena, será necesario convertir la cadena al tipo deseado mediante alguna de las funciones de construcción de tipos que ya conocemos (por ejemplo, *int* para obtener un número entero o *float* para obtener un número real).

Por su parte, la función *print* recibe una o varias expresiones por parámetros, y *muestra el resultado* de dichas expresiones en pantalla. Si el resultado de alguna de las expresiones es una cadena de texto, la muestra tal cual. Si el resultado de alguna de las expresiones es de cualquier otro tipo, la función *print* se encarga de convertir el resultado a cadena mediante el uso de la función *str*. Si recibe varias expresiones, por defecto *print* las muestra una tras otra, separadas por un espacio en blanco. Al finalizar de mostrar las expresiones, la ejecución de *print* finaliza imprimiendo un salto de línea; por consiguiente, la siguiente llamada a *print* escribirá en la siguiente línea de la pantalla.

La cabecera de la función print es:

```
print(*objects, sep=' ', end='\n') -> None:
```

El asterisco delante de objects indica que ahí puede haber un número varaible de parámetros. El parámetro *sep* es el separador usado para separar un valor a imprimir del siguiente. Tiene por el defecto como valor un espacio en banco. Al finalizar se imprime el valor de end que por defecto es un salto de línea. Ambos, el carácter usado para separar las distintas expresiones y el carácter usado como finalizador, pueden cambiarse utilizando los parámetros opcionales *sep*, *end*:

```
import random
numeros = [random.randint(1, 100) for _ in range(10)]

i = 0
for n in numeros:
       print(i, n, sep=': ')
       i = i+1
texto = "Muestrame con puntos"
for c in texto:
       print(c,end='.')
```

El resultado obtenido con el código anterior es:

```
1: 60
2: 2
3: 19
4: 16
5: 70
6: 33
7: 46
8: 97
9: 39
M.u.e.s.t.r.a.m.e. .c.o.n. .g.u.i.o.n.e.s.
```

Aunque el uso de los parámetros opcionales *sep* y *end* nos da algunas opciones para obtener la salida que deseamos en pantalla, a veces se nos puede ser insufiente. Por ejemplo, si queremos mostrar un mensaje formado por distintos trozos de texto y datos a extraer de variables o expresiones, puede que no siempre queramos usar el mismo separador entre cada dos expresiones. Un ejemplo sencillo lo tenemos en la siguiente sentencia que ya hemos escrito antes:

```
print("El resultado de", base, "elevado a", exponente, "es", \
              base**exponente, '.')
```

En este caso, nos interesa usar el espacio para separar los distintos trozos del mensaje a mostrar, salvo para el punto final, que debería aparecer a continuación del resultado de la expresión *base**exponente*. Además, la forma en que las cadenas de texto y las expresiones se van intercalando en los parámetros del *print* complica un poco la legibilidad de la sentencia. Es por todo esto por lo que es apropiado usar el *formateo de cadenas* visto antes.

```
a = int(input('Introduce un número:'))
b = int(input('Introduce un número:'))

print('El resultado de {} elevado a {} es {}.' \
       .format(a, b, a**b))
```

Lectura y escritura de ficheros

Muchas veces no es suficiente con la introducción de datos desde el teclado por parte del usuario. Como hemos visto en los ejercicios realizados a lo largo del curso, es muy habitual leer datos desde un fichero o archivo (que llamamos de entrada). Igualmente, es posible escribir datos en un fichero (que llamamos de salida).

Tanto la lectura como la escritura de datos en un fichero se pueden realizar de diversas formas:

- Mediante cadenas de texto libres, en lo que llamamos *ficheros de texto.*
- Mediante cadenas de texto de un formato predefinido, como es el caso de los ficheros *csv*.
- Mediante algún formato estándar de intercambio de datos (por ejemplo, *json*), lo que nos permite guardar y recuperar más tarde fácilmente el contenido de las variables de nuestros programas. A este tipo de escrituras y lecturas las llamamos *serialización* y *deserialización*, respectivamente.
- Mediante datos binarios, en lo que llamamos *ficheros binarios*. De esta forma, el programador tiene el control absoluto de los datos que se escriben o se leen de un fichero. Esto no lo veremos en esta asignatura.

Aquí solo veremos los dos primeros tipos.

Apertura y cierre de ficheros

Lo primero que hay que hacer para poder trabajar con un fichero es abrirlo. Al abrir un fichero, establecemos la manera en que vamos a trabajar con él: si lo haremos en modo texto o modo binario, o si vamos a leer o escribir de él, entre otras cosas.

La apertura de un fichero se realiza mediante la función *open*:

```
f = open('fichero.txt')
```

Si la apertura del fichero se lleva a cabo sin problemas, la función nos devuelve un *descriptor del fichero*. Usaremos esta variable más adelante para leer o escribir en el fichero.

Por defecto, el fichero se abre en modo texto para lectura. Podemos cambiar el modo en que se abre el fichero mediante el parámetro opcional *mode*, en el que pasaremos una cadena formada por alguno(s) de los caracteres siguientes:

- 'r': abre el fichero en modo lectura.
- 'w': abre el fichero en modo escritura. Si el archivo existía, lo sobrescribe (es decir, primero es borrado).
- 'a': abre el fichero en modo escritura. Si el archivo existía, las escrituras se añadirán al final del fichero.
- 't': abre el fichero en modo texto. Es el modo por defecto, así que normalmente no lo indicaremos y se entenderá que lo abrimos en modo texto. Es el modo que usaremos siempre en nuestra asignatura.
- 'b': abre el fichero en modo binario.

Veamos como ejemplo cómo abrir un fichero de texto para escribir en él, sobrescribiéndolo si ya existía:

```
f2 = open('fichero_escritura.txt', mode='w')
```

Cuando abrimos un fichero de texto es importante que tengamos en cuenta la *codificación de caracteres, enconding,* utilizada por el fichero. Existen diversos estándares, aunque el más utilizado hoy en día en el

contexto de Internet es el *utf-8*. Será éste el que usaremos preferiblemente. Por defecto, la función *open* decide la codificación de caracteres en función de la configuración de nuestro sistema operativo. Para especificar explícitamente que se utilice *utf-8* lo haremos mediante el parámetro opcional *encoding*:

```
f3 = open('fichero.txt', encoding='utf-8')
```

Cuando terminemos de trabajar con el fichero (por ejemplo, al acabar de leer su contenido), es importante *cerrarlo*. De esta forma liberamos el recurso para que puedan trabajar con él otros procesos de nuestra máquina, y también nos aseguramos de que las escrituras que hayamos realizado se llevan a cabo de manera efectiva en disco (ya que las escrituras suelen utilizar un buffer en memoria para mejorar la eficiencia). Para cerrar un fichero usamos el método *close* sobre el descriptor del fichero que queremos cerrar:

```
f.close()
f2.close()
f3.close()
```

Una forma de no olvidarnos de cerrar el fichero (algo muy habitual) es usar la sentencia *with*:

```
with open('fichero.txt', encoding='utf-8') as f:
    …
```

Una vez ejecutadas las instrucciones contenidas en el bloque *with*, el fichero es cerrado automáticamente. Esta variante tiene la ventaja además de que si se produce cualquier error mientras trabajamos con el fichero, que produzca la parada de la ejecución de nuestro programa, el fichero también es cerrado. Esto no ocurre si abrimos el fichero sin usar *with*.

El encoding de un fichero podemos obtenerlo con la función *print encoding* siguienete.

```
import chardet

def print_encoding(file:str)->None:
    with open(file,"rb") as f:
        data = f.read()
        enc = chardet.detect(data)
        return enc['encoding']
```

Lectura y escritura de texto

Una vez abierto un fichero en modo texto, podemos leer todo el contenido y guardarlo en una variable de tipo cadena mediante el método *read*:

```
with open('fichero.txt', encoding='utf-8') as f:
    contenido = f.read()

print(contenido)  # Mostramos el contenido del fichero
```

Como esa operación puede ser habitual diseñaremos una función para reutilizar

Aunque se puede hacer de esta forma, es más habitual procesar los ficheros de texto línea a línea. Para ello, podemos usar el descriptor del fichero dentro de un bucle *for*, como si se tratara de una secuencia de cadenas, de manera que en cada paso del bucle obtendremos la siguiente línea del fichero:

```
with open('fichero.txt', encoding='utf-8') as f:
    for linea in f:
        print(linea)
```

En el ejemplo anterior se visualizan cada línea separada con una línea vacía. Esto es así porque la línea leída del fichero incluye al final un salto de línea, y a su vez la función *print* incluye un salto de línea tras la cadena a mostrar. Si queremos mostrar el contenido del fichero sin incluir el fin de línea del print, podríamos hacer esto:

```
with open('fichero.txt', encoding='utf-8') as f:
    for linea in f:
        print(linea, end='')
```

Como este código es muy útil diseñamos la función *lineas_de_fichero* que podemos reutilizar.

```
def lineas_de_fichero(file:str,encoding='utf-8') -> list[str]:
    with open(file,encoding=encoding) as f:
        lineas_de_fichero =  [linea.rstrip('\n') \
             for linea in f]
        return lineas_de_fichero
```

Podemos observar que la función *lineas_de_fichero* devuelve una lista formada por las líneas del fichero de las que se ha eliminado los caracteres en blanco, lo que incluye el salto de línea, del final de línea.

Para escribir texto en un fichero, usaremos el método *write* sobre el descriptor del fichero y definiremos una función para reutilizar

```
with open('f_e.txt', mode='w', encoding='utf-8') as f:
    f.write('Este texto se almacenará en el fichero.')
```

Comprobemos si se ha realizado la escritura correctamente:

```
with open('f_e.txt', encoding='utf-8') as f:
    for linea in f:
        print(linea, end='')
```

Diseñamos igualmente la función reutilizable *write*.

```
def write(file:str,texto:str) -> None:
    with open(file, "w", encoding='utf-8') as f:
        f.write(texto)
```


Lectura y escritura de CSV

Un tipo de fichero de texto que usaremos mucho es el llamado formato *CSV* (por *Comma-Separated Values*). Estos ficheros se utilizan para almacenar datos de tipo tabular, al estilo de una hoja de cálculo.

```
import csv

with open("datos_2.txt",encoding=encoding) as f:
    lector = csv.reader(f, delimiter=delimiter)
    lineas =  [linea for linea in lector]
```

Diseñamos la función líneas csv para reutilizar este código.

```
def lineas_de_csv(file:str, delimiter:str=",",
             encoding='utf-8')-> list[list[str]]:
    with open(file,encoding= encoding) as f:
        lector = csv.reader(f, delimiter = delimiter)
        lineas_de_fichero =  [linea for linea in lector]
        return lineas_de_fichero
```

Ejercicios

Veamos algunos ejemplos de uso.

Sea el fichero *datos_3.txt* cuyo contenido es:

```
0,0,1,2
0
-1,2
14,3,7,8
0,4,0,9,2,0,1,2

1,0,7,6
```

Y el fichero *lin_qujote.txt*

```
EL INGENIOSO HIDALGO DON QUIJOTE DE LA MANCHA

Miguel de Cervantes Saavedra

     Capítulo primero

     Que trata de la condición y ejercicio del famoso hidalgo
     Don Quijote de la Mancha

     En un lugar de la Mancha, de cuyo nombre no quiero
acordarme, no ha mucho tiempo que vivía un hidalgo de los de
lanza en astillero, adarga antigua, rocín flaco y galgo
corredor. Una olla de algo más vaca que carnero, salpicón las
```

1. *Contar las lineas que tiene el fichero datos_3.txt*

```
lineas = lineas_de_fichero("../../../resources/datos_3.txt")
n = 0
for ln in lineas:
       n = n+1
print(n)
```

2. *Contar cuantas lineas vacías hay en el fichero datos_3.txt*

```
lineas = lineas_de_fichero("../../../resources/datos_3.txt")
n = 0
for ln in lineas:
       if len(ln) == 0:
            n = n+1
print(n)
```

3. Contar cuantos números hay en el fichero datos_3.txt

```
lineas = lineas_de_csv("../../../resources/datos_3.txt", \
             delimiter=',')
n = 0
for ln in lineas:
       for num in ln:
            n = n+1
print(n)
```

4. Sumar los numeros del ficheros datos_3.txt

Una primera posibilidad es leer el fichero como un csv con delimitador ‘,’.

```
lineas = lineas_de_csv("../../../resources/datos_3.txt",\
             delimiter=',')
s = 0
for ln in lineas:
       for num in ln:
             s = s + int(num)
print(s)
```

Una segunda posibilidad es leer las líneas del fichero, filtrar las que no son vacías y posteriormente dividir cada líea en fragmentos. Esto lo podemos conseguir el método split de las cadenas de caracteres.

```
lineas = lineas_de_fichero("../../../resources/datos_3.txt")
s = 0
for ln in lineas:
       if len(ln) > 0:
            for num in ln.split(","):
                    s = s + int(num)
print(s)
```

5. *Diseñar una función que lea los datos de un fichero como el datos_3.txt y los acumule*

Partimos de un fichero líneas formadas por números separados por un delimitador formado por un carácter. Acumular u¡es una operación muy general y podemos diseñar de una forma que podamos reutilizar.

```
from us.lsi.tools.File import lineas_de_csv
from typing import TypeVar, Callable

E = TypeVar('E')
R = TypeVar('R')

def acumula(fichero:str,delimiter:str=',',\
       encoding:str='utf-8',inicial:R='', \
       f:Callable[[R,E],R]=lambda r,e:r+e)->R:
    lineas = lineas_de_csv(fichero,delimiter=delimiter,\
              encoding=encoding)
    r:R = inicial
    for ln in lineas:
        if len(ln) > 0:
            for e in ln:
                r = f(r,e)
    return r
```

Para comprender el funcionamiento de esa función podemos hacer la llamada

```
r = acumula("datos_3.txt",encoding='ISO-8859-1')
print(r)
```

Podemos obervar que el resultado es una cadena de caracteres obtenida concatenando todos los números que había en el fichero. Esto es porque el parámetro initial tiene como valor por defecto la cadena vacía y la función que va acumulando en la variable r es el operador +.

Probemos ahora con

```
r = acumula("datos_3.txt",encoding='ISO-8859-1', inicial=0, \
       f=lambda r,e:r+int(e))
print(r)
```

El resultado es la suma de los números que se encontraban en el fichero. El producto de todo ellos se puede obtener como:

```
r = acumula("datos_3.txt",encoding='ISO-8859-1', inicial=1, \
       f=lambda r,e:r*int(e))
print(r)
```

El producto resultará cero si algún número lo es. Igualmente podemos obtener una lista de enteros formada por los números en el fichero:

```
r = acumula("datos_3.txt",encoding='ISO-8859-1', inicial=[], \
       f=lambda r,e:r+[int(e)])
print(r)
```

Finalmente obtenemos un conjunto de enteros formada por los números en el fichero:

```
r = acumula("datos_3.txt",encoding='ISO-8859-1', \
       inicial=set(), f=lambda r,e:r|{int(e)})
print(r)
```

Hemos visto como diseñar funciones muy reutilizables usando lambda expresiones como parámetros.

Podemos observar que el tipo de *inicial* es *R*. Es un tipo genérico que se concretará más adelante. El el tipo de la variable *r* que acumula el resultado.

El tipo de *f* es *Callable[[R,E],R].* Es decir será una función que combinará el valor en *r* con el valor *e* obtenido del fichero. Esta función puede ser muy general.

6. Averiguar el encoding del fichero lin_quijote.txt

```
print(encoding("lin_quijote.txt"))
```

7. Contar cuantas palabras contiene el fichero lin_quijote.txt.

Asumimos que las palabras están separadas por espacios en blanco, comas o puntos.

Una posibilidad es leer las líneas de un fichero y posteriormente divididir cada línea en palabras. Los delimitadores separan unas palabras de otras. Cuando hay varios delimitadores podemos usar la función split del módulo re.

Si el separador fuera un carácter único podríamos usar *lineas_de_csv*.

```
lineas =lineas_de_fichero("lin_quijote.txt", \
       encoding='ISO-8859-1')
n = 0
for ln in lineas:
       for p in re.split('[ ,.]',ln):
            if len(p) > 0:
                n = n+1
print(n)
```

8. Contar cuantos caracteres tiene el fichero que no sean delimitadores

```
lineas = lineas_de_fichero("lin_quijote.txt", \
       encoding='ISO-8859-1')
n = 0
for ln in lineas:
       for p in re.split('[ ,.]',ln):
              for c in p:
                    n = n+1
print(n)
```

El flujo de ejecución

Llamamos *flujo de ejecución* al orden en que se van ejecutando las instrucciones de un programa. Ya sabemos que en un programa en Python las distintas instrucciones se van ejecutando en orden secuencial, empezando por la primera. En concreto el programa empieza a ejecutarse en la primera línea que no esté indentada. Por ejemplo:

```
print("==== Cálculo de una potencia =====")
print("Introduzca un número entero (base):")
base = int(input())

print("Introduzca un número entero (exponente):")
exponente = int(input())

print("El resultado de", base, "elevado a", exponente, "es",\
      base**exponente)
```

El flujo de ejecución es el siguiente: primero se ejecuta la primera instrucción del programa, a continuación, la segunda, y así sucesivamente hasta llegar al final del programa. En realidad, el flujo de ejecución es un poco más enrevesado de lo que parece a primera vista, porque cuando en una instrucción aparece una llamada a una función, el flujo de ejecución *salta* hacia la primera de las instrucciones contenidas en la definición de dicha función. Así, por ejemplo, cuando se invoca a la función *print*, el flujo de ejecución salta hasta la primera de las instrucciones de la definición de

la función *print*. No sabemos qué instrucción es esa, porque se trata de una función predefinida y no conocemos su cuerpo.

Veamos un segundo ejemplo en el que aparezca una función definida por nosotros:

```
def lectura_datos():
    print("Introduzca un número entero (base):")
    base = int(input())
    print("Introduzca un número entero (exponente):")
    exponente = int(input())
    return base, exponente

print("==== Cálculo de una potencia =====")

base, exponente = lectura_datos()

print("El resultado de {} elevado a {} es {}" \
      .format(base, exponente, base**exponente))
```

Podemos observar que el flujo de ejecución es el siguiente:

- Primero se ejecuta la primera de las instrucciones no indentadas: print("==== Cálculo de una potencia =====")
- Después se ejecuta la segunda instrucción, que incluye una llamada a la función lectura_datos. Como consecuencia de dicha llamada, la ejecución de la instrucción queda en suspenso, y el flujo de ejecución pasa al cuerpo de la función.
- Se ejecutan secuencialmente todas las instrucciones del cuerpo de lectura_datos, hasta ejecutarse la instrucción return base, exponente. En ese momento, el flujo de ejecución pasa a la instrucción que contenía la llamada a la función y que había quedado en suspenso: base, exponente = lectura_datos().
- Se termina de ejecutar dicha instrucción (se realiza la asignación a las variables base y exponente).
- Se ejecuta la siguiente y última instrucción.

Este es el camino que sigue la ejecución de los programas en general. Para controlar este flujo de ejecución disponemos de las *instrucciones de control del flujo de ejecución*: instrucciones que permiten que una parte

del programa se ejecute o no en función de una determinada condición (instrucciones *selectivas* o *condicionales*), o que un bloque de instrucciones se ejecute de manera repetida (instrucciones *repetitivas* o *iterativas*).

La instrucción if, excepciones y la sentencia try

La instrucción *if* es una *instrucción de control de flujo selectiva* o *condicional.* Esto significa que permite que el flujo de ejecución de un programa siga un camino u otro dentro del código de un programa. Este control condicional del flujo de ejecución tiene algunas variantes. La primera es la instrucción if que tiene la forma:

```
if g1:
  s1
elif g2:
  s2
else:
  s3
```

Aquí se evalúa la guarda g1, una expresión lógica, y se es verdadera se ejecuta s1. Si no lo es se comprueba g2 y si lo es se ejecuta s2. Y así sucesivamente. Finalmente se puede incluir una cláusula *else* que ejecutará la sentencia s3.

La anterior estructura es la *sentencia if*. Es una sentencia porque no calcula ningún valor, sólo sirve para ordenar el flujo de ejecución del programa.

Junto a la sentencia en Python existe también el operator ternario *if/else*. Este operador calcula un valor y puede ser combinado con otros operadores para formar expresiones. Su forma es:

```
r = exp1(e) if p(e) else exp2(e)
```

Es decir el resultado del operador es exp1(e) si se cumple la condición p(e) y si no se cumple el reultado del operador es exp2(e). Veamos, por ejemplo una expresión que calcula el cuadrado de untero si es par y el cubo si es impar.

```
r = e*e if e%2==0 else e*e*e
```

Otra tarea frecuente es comprobar que se cumple una condición y si no se cumple abortar la ejecución.

En el flujo de ejecución de un programa podemos comprobar condiciones y disparar *excepciones* cuando estas no se cumplan. Una excepción es una ruptura del flujo de control que se produce usualmente cuando ocurre algún error y se dispara con *raise Exception(message)*. Como veremos abajo las excepciones se pueden gestionar con la sentencia *try*. Estas ideas las podemos encapsular en funciones para su uso posterior:

```
def checkArgument(condition:bool,message:str):
    if(not condition):
        raise Exception(message)

def checkNotNone(reference):
    if(not reference):
        raise Exception("Es None {0:s}".format(reference))

def checkElementIndex(index_bool:int,size:int):
    if(not (index_bool>=0 and index_bool<size)):
        raise Exception("Indice no en el rango: \
             Index = {0:d},size{1:d}".format(index_bool,size))
```

Veamos un ejemplo para aclarar lo anterior. Queremos resolver ecuaciones de primer y segundo grado dados sus coeficientes.

Solución de la ecuación de primer grado $ax + b = 0, a \neq 0$ cuya solución es $x = -\frac{b}{a}$.

Asumimos que si no se cumple la condición especificada se disparará una excepción.

```
import math
from typing import Tuple, Union
from math import pi,sqrt

def sol_ecuacion_primer_grado(a:float,b:float) -> float:
    checkArgument(a>0,'El coeficiente a debe ser distinto de \
              cero y es {0:.2f}'.format(a))
    return -b/a
```

Solución de la ecuación de segundo grado $ax^2 + bx + c = 0, , a \neq 0$, cuya solución es:

- $\Delta = b^2 - 4ac$
- $\frac{-b \pm \sqrt{\Delta}}{2a}$

```
def sol_ecuacion_segundo_grado(a:float,b:float,c:float) -> \
              tuple[float,float] | tuple[complex,complex]:
    checkArgument(a>0,'El coeficiente a debe ser distinto de \
              cero y es{0:.2f}'.format(a))
    disc = b*b-4*a*c
    if disc >= 0 :
        r1 = -b/(2*a)
        r2 = sqrt(disc)/(2*a)
        s1,s2 = r1+r2,r1-r2
        return (s1,s2)
    else :
        re = -b/(2*a)
        im = sqrt(-disc)/(2*a)
        s1,s2 = complex(re,im),complex(re,-im)
        return (s1,s2)
```

En el cuerpo de la función *sol_ecuacion_segundo_grado* se ha empleado una instrucción *if* que permite que se ejecuten unas instrucciones u otras concretas en función de distintas *condiciones* que llamamos *guardas*. Las condiciones son cada una de las *expresiones lógicas* que aparecen a la derecha de las palabras reservadas *if*. Una expresión lógica es una expresión cuyo tipo es lógico; es decir, sus valores son *True* (verdadero) o *False* (falso).

Hemos usado también la función *checkArgument* adecuada para comprobar una condición en le flujo de un programa y disparar una excepción si no se cumple.

Las excepciones se pueden gestionar. Gestionar excepciones es decidir que hacer si se ha disparado una excepción. Esto se consigue con la cláusula *try* que tiene la forma:

```
try:
    ...
except E1:
    ...
else:
    ...
```

Se intentan ejecutar las sentencias que van detrás de *try* y antes de *except*. Si se dispara una excepción de tipo E1 se ejecuta las sentencias después de *except* y antes de *else*. Si se dispara una excepción diferente se ejecutan las sentencias después del *else*. Si no se dispara ninguna excepción se ignora el *except* y el *else*.

Excepciones típicas son las aritméticas (división por cero)

```
a = 5
b = 0

try:
    c = a/b
except ArithmeticError:
    print('Se ha producido una división por cero de {} entre \
              {}'.format(a,b))
```

Y las de apertura de ficheros.

```
try:
    with open("prueba.txt") as f:
        pass
except IOError:
    print('No se encuentra el fichero prueba.txt')
```

Bucles

Las *instrucciones de control de flujo repetitivas* o *iterativas* permiten que un bloque de instrucciones se ejecute un número determinado de veces (incluyendo la posibilidad de que se ejecute una única vez o ninguna). No

siempre sabremos de antemano cuántas veces se va a ejecutar el bloque de instrucciones, pues esto puede depender del estado actual de ejecución del programa (es decir, del valor concreto de determinadas variables).

En Python tenemos dos instrucciones de este tipo: la instrucción *while* y la instrucción *for*.

Instrucción while

La instrucción *while* va seguida de una condición (es decir, una expresión lógica, como en la instrucción *if*) que se denomina *guarda*. A continuación de la condición aparecen dos puntos, y después un bloque de instrucciones, debidamente indentadas que llamaremos cuerpo del *while*. Las instrucciones del cuerpo se ejecutarán una y otra vez, mientras la condición sea cierta.

En el ejemplo se utiliza una sentencia *while* para repetir un conjunto de sentencias mientras que se cumpla una condición.

```
print("Del 1 al 5, ¿cómo te encuentras hoy?: ")
bienestar = int(input())

while bienestar < 1 or bienestar > 5:
    print("Por favor, introduce un valor del 1 al 5:")
    bienestar = float(input())

if bienestar < 3: # 1 ó 2
    print("¡Ánimo! Verás como el día mejora.")
elif bienestar < 5: # 3 ó 4
    print("No está mal, ¡hoy será un gran día!")
else: # 5
    print("¡Me alegro de que te sientas bien!")
```

Veamos ahora el cálculo del máximo común divisor y el mínimo común múltiplo de dos enteros positivos usando el *Algoritmo de Euclides*. Este algoritmo se basa en las propiedades *mcd(a,b) = mcd(b,a%b) y mcd(0,b) = b* . Por otra parte el mínimo común múltiplo cumple *a*b = mcd(a,b) * mcm(a,b)*

```
def mcd(a:int, b:int)->int:
    checkArgument(a>=0 and b>0,'El coeficiente a debe ser \
              mayor o igual que cero y b mayor que cero \
              y son: a = {0}, b = {1}'.format(a,b))
    while b > 0:
        a, b = b, a%b
    return a
```

Esta función puede llamarse

```
print(mcd(135,45))
```

La instrucción for

La instrucción *for* permite recorrer los elementos de un *agregado de datos*, ejecutando un bloque de instrucciones, que llamaremos su cuerpo, una vez por cada uno de sus elementos. Recordemos que rangos, cadenas de caracteres, listas, conjuntos y diccionarios son agregados de datos. Observemos el siguiente ejemplo:

```
from typing import Any

def muestra_todos(ls: list[Any]) -> None:
    for elemento in ls:
        print(elemento)
```

Como se puede ver en el ejemplo, la sintaxis de la instrucción *for* consiste en:

```
for elemento in agregado:
    bloque de instrucciones
```

Donde *agregado* es, por ahora, una cadena de caracteres, una lista, una tupla, un conjunto o un diccionario (entre otros), y *elemento* es una variable donde se almacenará cada elemento del agregado en cada vuelta del bucle (la variable *elemento* puede tener el nombre que se quiera).

Veamos los siguientes ejemplos cómo se recorren los elementos de distintos tipos de agregados, incluyendo una cadena de texto:

```
muestra_todos(["Tomate", "Cebolla", "Pimiento", "Pepino", \
       "Ajo","Aceite", "Vinagre"])

muestra_todos({"Tomate", "Cebolla", "Pimiento", "Pepino", \
       "Ajo","Aceite", "Vinagre"})

muestra_todos({"Tomate": 1000, "Cebolla": 50, \
       "Pimiento": 50, "Pepino":100, "Ajo": 15, \
       "Aceite": 100, "Vinagre": 20})

muestra_todos("Ingredientes: Tomate, ...")
```

Resumiendo:

- Las listas, tuplas y cadenas de caracteres se recorren en el orden en que están posicionados los elementos.
- Los conjuntos se recorren en un orden arbitrario (los elementos no tienen una posición determinada en los conjuntos).
- Al recorrer los diccionarios se obtienen únicamente las claves usualmente en un orden arbitrario
- Combinando la sentencia *for* con un agregado podemos diseñar funciones muy variadas.

Veamos algunas funciones que combinan *for* con un agregado.

1. *Obtener la suma de los elementos en una lista*

```
def suma(ls:list[int]) -> int:
    a = 0
    for e in ls:
        a = a+e
    return a
```

2. Obtener la media de los elementos de una lista

```
def media(ls:list[int]) -> float:
    a = (0,0)
    for e in ls:
        a = (a[0]+e,a[1]+1)
    return a[0]/a[1]
```

3. Obtener la desviación típica de los elementos de una lista

```
def desviacion_tipica(ls:list[int]) -> float:
    a = (0.,0.,0)  #(sum x^2, sum x, num elem)
    for e in ls:
        a = (a[0]+e*e,a[1]+e,a[2]+1)
    checkArgument(a[2]>0,'La lista esta vacía')
    return sqrt(a[0]/a[2]-(a[1]/a[2])**2)
```

Estas funciones ya vienen predefinidas en Pyhton con los nombres *sum, mean y pstdev.* Más adelante las estudiaremos con detalle.

Rangos

La instrucción *for* se puede utilizar también para repetir un bloque de instrucciones un número determinado de veces. En dichas ocasiones, nos será de utilidad la función predefinida *range*. Esta función devuelve una secuencia de enteros entre un limite inferior, uno superior y con un incremento dado. Podemos considerar que range es un agregado de datos virtual en el sentido que sus elementos no están juntos en memoria pero se nos proporcionan uno tras otro. Por ejemplo:

```
for i in range(5):
    print(i)
```

Podemos observar que "parece" que *range(5)* devuelve una lista con cinco números: *[0, 1, 2, 3, 4]*. En realidad, los números de la secuencia no están en ninguna lista ni ningún otro agregado en la memoria, sino que se van "generando" a medida que va haciendo falta. Es decir range es un *agregado virtual*. Es por eso por lo que si ejecutamos lo siguiente:

```
print(range(5))
print(range(0, 5))
```

No obtenemos ninguna lista con los elementos correspondientes, sino más bien una representación que nos indica que la secuencia va de 0 a 4 (observa que el último número que aparece no se incluye en la secuencia).

Si queremos observar los números que contiene la secuencia generada por *range*, podemos recorrer sus elementos con un *for*, como en el ejemplo anterior. O también podemos construir una lista a partir de la secuencia, de esta forma:

```
list(range(5))
```

Podemos pasar a *range* dos o tres parámetros.

```
print(list(range(5, 10)))
print(list(range(5, 10, 2)))
print(range(10, 5, -1))
```

4. *Sumar los valores de una secuencia aritmética*

```
def suma_aritmetica(a:int,b:int,c:int)->int:
    s = 0
    for e in range(a,b,c):
        s = s + e
    return s
```

La forma de presentar los resultados podría ser:

```
print("La suma de la pregresión aritmética de {0} a {1} con\
 razón {2} es{3}" \
       .format(2,500,7, suma_aritmetica(2,500,7)))
```

La instrucción break

Hay veces en que necesitamos salir del bucle impuesto por las sentencias *while* o *for*. Veamos la función suma_*primeros(ls:list[int], n:int) -> int*.

```
def suma_primeros(ls:List[int],n:int)->int:
    i = 0;
    a = 0
    for e in ls:
        if i < n:
            a = a + e
        i = i +1
    return a
```

La ejecución de la función suma_primeros puede ser ineficiente en algunas circunstancias. Veamos la siguiente prueba:

```
print("Suma de los primeros 10 números de una secuencia \
 de mil millones de elementos:", \
       suma_primeros(list(range(10000000)), 10))
```

¿Por qué está tardando tanto?

El problema es que el bucle que hemos escrito en *suma_primeros* ejecuta el bloque de instrucciones una vez por cada elemento de la secuencia recibida. Esta secuencia tiene un número muy alto de elementos, lo que ocasiona que, a pesar de la velocidad de ejecución de los microprocesadores actuales, la ejecución se eterniza. ¿Realmente es necesario recorrer todos los elementos de la secuencia, cuando sólo quiero sumar los primeros 10 elementos?

Veamos la siguiente modificación de la definición de la función:

```
def suma_primeros(ls:List[int], n:int)->int:
    suma = 0
    i = 0
    for e in ls:
        if i < n:
            suma += e
        else:
            break
        i = i +1
    return suma
```

La instrucción *break,* escrita dentro del bloque de instrucciones de un bucle, ocasiona que el bucle termine su ejecución.

```
print("Suma de los primeros 10 números de una secuencia de \
 mil millones de elementos:", \
      suma_primeros(list(range(100000000)), 10))
```

Iterables y sus operaciones

En casi todos los lenguajes actuales existe el concepto de *iterable*. Un *iterable* es, dicho de una forma simplificada, un agregado de datos, real o virtual, que puede ser recorrido mediante una sentencia *for*. Ya hemos visto ejemplos de iterables: rangos, cadenas de caracteres, listas, conjuntos y rangos.

Un iterable, en definitiva, es una vista de un agregado de objetos que permite obtenerlos sus elementos una tras otro y recorrerlos en un bucle *for*.

La diferencia de un iterable con un agregado de datos es que el iterable es, simplemente, un mecanismo para obtener unos objetos tras otros ya sea porque previamente estaban en memoria o porque se vayan generando consecutivamente.

Un agregado puede ser recorrido de diferentes maneras. El mecanismo para recorrer un agregado de una determinada manera se implementa mediante un iterador (*iterator)*. Un objeto es iterable en la medida que puede ofrecernos uno o varios iteradores.

Todo objeto iterable ofrece el método *__iter__()* que nos devuelve un iterador. Todo objeto de tipo iterator ofrece el método *__next__()* que nos va a ir dando consecutivamente los elementos del objeto iterable. Cunado no haya más elementos se disparará la excepción *StopIteration.* Para saber

si un objeto es iterable debemos saber si ofrece el método *__iter__()* y para saber si objeto es un iterador debemos comprobar si ofrece el método *__next__()*. Python ofrece las funciones predefinidas iter y next que llaman a los métodos anteriores.

La importancia de los iterables y los iteradores es que podemos diseñar algoritmos para iterables y no específicamente para listas, conjuntos, rangos o diccionarios. Esto es posible porque la sentencia *for* acepta un iterable en general.

Una forma de comprobar si un objeto es iterable es simplemente intentar recorrerlo mediante un for.

La sentencia fro puede ser reducida a una sentencia while usando las funciones iter y next.

```
ls = [1,5,7,9]
it = iter(ls)
while True:
       try:
              e = next(it)
              print(e)
       except StopIteration:
              break
```

Funciones de transformación de iterables y su secuenciación

Ya hemos comentado que las listas, los conjuntos, los diccionarios y los rangos son iterables. A partir de ellos se pueden definir otros iterables mediante funciones proporcionadas por Python.

Existen varias funciones predefinidas, *built-in*, que transforman un iterable en otro: *map*, *filter*, *accumulate*, *reversed*, *zip*, *enumerate*, etc. Estudiaremos ahors su uso. Más adelante veremos una posible implementación.

- *map(f:Callable[[E],R],iterable:Iterable[E]) -> Iterable[R].*
  El primer parámetro es una función que transforma cada elemento del iterable

- *filter(p: Callable[[E],bool], iterable:Iterable[E]) -> Iterable[E]*.
  El primer parámetro es una función que devuelve un tipo bool. Es decir un predicado.
- *accumulate(iterable:Iterable[E], func:Callable[[E,R],R] = operator.add, *, initial:R=None)-> Iterable[R].*
  Obtiene un nuevo iterable con los valores acumulados del iterable proporcionado. El parámetro initial proporciona un valor inicial si es distinto de None.
- *reversed(iterable:Iterable[E]) -> :Iterable[E]*.
  Obtiene un iterable con los valores recorridos en sentido inverso. Se puede aplicar solamente a agregados indexables o a objetos que tengan el método *__reversed__().*
- *zip(*iterables:Iterable[Iterable[Any]])->Iterable[tuple(Any,..)]*.
  Obtiene un iterable de tuplas formadas por un elemento de cada iterable. En general zip puede aceptar varios iterables. El asterisco delante del parámetro indica que pueden aparecer un número indeterminado de parámetros reales. El iterable producido se extenderá hasta el iterable más corto de entre los parámetros.
- *Operador * (unzip)*. Aplicado a un iterable los descompone en sus elementos.
- *enum(iterable:Iterable[E], start:int=0) -> :Iterable[tuple[int,E]]*
  Obtiene un iterable de tuplas formadas por un elemento y su posición comenzando en start

Junto a ellas podemos diseñar otras funciones para transformar iterables: *distinct*, *limit*, *flatmap*

- *distinct(iterable:Iterable[E]) -> Iterable[E].*
  Devuelve un iterable donde no se repiten los elementos
- *limit(iterable:Iterable[E], limit:int) -> Iterable[E].*
  Devuelve un iterable con un número máximo de limit elementos
- *flat_map(iterable:Iterable[E],fm:Callable[[E], Iterable[R]]=identity) -> Iterable[R]*.
  Devuelve un iterable con las partes de cada elemento definidas por la función fm


- *iterate(initial:E, predicate:Callable[[E],bool],operator:Callable[[E],E]) -> Iterable[E].*
  Devuelve un iterable con *initial* como valor inicial, cada elemento siendo obtenido del anterior aplicando *operator* mientras se cumpla *predicate*.

Las funciones anteriores se pueden combinar por secuenciación. Es decir definiendo un iterador a partir de otro de forma secuencial. Veamos algunos ejemplos de los anterior.

---

1. *Definir un iterable que recorra los cuadrados de los elementos de la progresión aritmética ue va desde a hasta b con con razón c y son múltiplos de d.*

```
def ej1(a:int,b:int,c:int,d:int)->Iterable[int]:
    it1 = map(lambda x:x**2,range(a,b,c))
    it2 = filter(lambda x:x%d==0, it1)
    return it2
```

Sea el fichero datos_2.txt de la forma:

```
0,0,1,2
0,1,4,5
0,2,9,1
0,3,7,8
0,4,0,9,2,0,1,2

1,0,7,6
```

2. *Definir un iterable que devuelva los enteros diferentes que contiene el fichero datos_2.txt*

```
def ej2(fichero:str)->Iterable[int]:
    it1 = lineas_de_csv(fichero)
    it2 = flat_map(it1)
    it3 = map(lambda e:int(e),it2)
    it4 = distinct(it3)
    return it4
```

3. *Definir un iterable con los caracteres de una cadena cuyo índice es par*

```
def ej3(cadena:str):
    it1 = enumerate(cadena)
    it2 = filter(lambda e:e[0]%2==0,it1)
    it3 = map(lambda e: e[1],it2)
    return it3
```

4. *Imprimir una cadena incluyendo un espacio en blanco entre cada dos caracteres*

```
print(*ej3("En un lugar de la Mancha de cuyo nombre no quiero
acordarme"),sep=" ")
```

Funciones de acumulación de iterables

Python ofrece un conjunto de funciones predefinidas para agregar los elementos de un iterable. Algunas de estas son:

- *len(iterable)*
  Obtiene el número de elementos del iterable
- *sum(iterable,start=0)*
  Obtiene la suma de los valores de un iterable con valor inicial opcional.

- *max(iterable,key:Callable[[E],R]=None, default=None):*
  Obtiene el máximo de los valores de un iterable.
- *min(iterable,key: Callable[[E],R]=None, default=None)*
  Obtiene el mínimo de los valores de un iterable.
- *all(iterable:Iterable[bool])*
  True si los valores del iterable son verdaderos.
- *any(iterable:Iterable[bool])*
  True si alguno de los valores del iterable es verdadero.
- *sorted(iterable,key:Callable[[E],R]=None,reverse: bool= False)*
  Obtiene una lista con los valores de un iterable ordenados.
- *reduce(f:Callable[[R,E],R], iterable:Iterable[E], initial:R=None)->R|None*
  Obtiene el valor acumulado del iterable usando la función de acumulación f y opcionalmente el valor inicial.

Las funciones *max* y *min* calculan, respectivamente, el máximo y el mínimo de un iterable. Estas funciones se pueden aplicar sobre cualquier iterable siempre que sea homogéneo (todos los elementos sean del mismo tipo) y ese tipo tenga un *orden natural*. Por ejemplo, los tipos (*int* y *float*) y las cadenas (*str*) sí tienen orden natural. Si los elementos del iterable son tuplas entonces los elementos se comparan utilizando su primer elemento.

La función *built-in sorted* crea una lista ordenada a partir de un iterable. El parámetro *reverse* nos permite indicar si queremos aplicar un orden ascendente o descendente. Otro parámetro interesante de la función *sorted* es *key*. A través de ese parámetro podemos indicar el uso de una función de comparación específica. Por ejemplo, podemos indicarle que utilice la función predefinida *len*, de manera que primero se pasará cada elemento por dicha función. Un caso típico de aplicación del parámetro *key* es la ordenación de una lista de tuplas. El criterio de ordenación *por defecto* de una lista de tuplas se basa en comparar los primeros componentes de las tuplas.

Adicionalmente diseñamos las funciones

- *index_predicate(iterable:Iterable[E],predicate:Callable[[E],bool], default:int=-1)->int:*
  Obtiene el valor del primer índice que cumple el predicado o -1 si ninguno lo cumple
- *find_first(iterable:Iterable[E], p:Callable[[E],bool]) -> E|None.*
  Obtiene el primer elemento que cumple el predicado o None si no hay ninguno.
- *count(iterable:Iterable[E], predicate:Callable[[E],bool]=lambda _:True)->int:*
  Obtiene el número de elementos que cumple el predicado

La combinación de iterables y acumuladores da lugar a código que puede resolver problemas complejos. Veamos algunos ejemplos.

1. *Ordena unas lista cadenas de caracteres según su tamaño*

```
dias = ["lunes", "martes", "miércoles", "jueves",\
       "viernes", "sábado", "domingo"]
print(sorted(dias, key=len))
```

2. *Multiplicar los elementos de un iterable*

```
from functools import reduce
print(reduce(lambda x,y: x*y,range(2,30,5)))
```

3. *Contar el número de apariciones de un carácter en una cadena*

```
texto = "En un lugar de la Mancha de cuyo nombre no quiero \
       acordarme"
print("Apariciones de la letra a:", count(texto, \
   lambda e:e=="a"))
```

4. *Encontrar el índice de la primera aparición de una lextra de un texto*

```
texto = "En un lugar de la Mancha de cuyo nombre no quiero \
acordarme"
print("Primera aparicion de la letra a:",\
     index_predicate(texto,lambda e:e=="a"))
```

5. *Obtener un conjunto con los elementos de un iterable*

```
from functools import reduce

texto = "En un lugar de la Mancha de cuyo nombre no quiero\
acordarme"
print(reduce(lambda s,e: s | {e}, texto, set()))
```

6. *Obtener cuantos letras distintas hay en una cadena*

```
texto = "En un lugar de la Mancha de cuyo nombre no quiero\
acordarme"

print(count(distinct(texto)))
```

7. Diseñar un acumulador que construya una lista a partir de un iterable

```
def tolist(iterable:Iterable[E])->list[E]:
    ls= []
    for e in iterable:
        ls.append(e)
    return ls
```

8. Diseñar un acumulador que construya un conjunto con los elementos de un iterable

```
def toset(iterable:Iterable[E])->set[E]:
    st= set()
    for e in iterable:
        st.add(e)
    return st
```

Diseño e implementación iterables: generadores. La sentencia yield

Podemos obtener nuevos iterables combinando for con *yield*. *Yield* es similar a *return* pero a diferencia de éste la función que lo usa produce un iterable. Es decir va dando los valores uno tras otro. Las funciones que incluyen la sentencia *yield* se denominan *generadores*.

Los *generadores* son una forma sencilla y potente de crear un iterador.

Una característica importante de los generadores es que tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas que se hagan al generador, es decir, a diferencia de una función común, una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración yield (que es donde terminó la función en la última llamada).

Los generadores son mecanismos adecuados para diseñar nuevos iteradores para agregados de datos. Veamos algunos ejemplos:

1. *Diseñar un iterador que recorra una lista hacia atrás*

```
from typing import Iterable, Any, List

def hacia_atras(ls:List[Any])-> Iterable[Any]:
    i = len(ls)-1
    while i>=0:
        yield ls[i]
        i = i-1

for a in hacia_atras(ls):
    print(str(a)+" ",end="")
```

Las funciones de transformación de iterables vistas arriba pueden ser implementadas con generaores.

Para no repetir usaremos este código compartido en los ejemplos siguientes

```
from typing import Iterable, Any, Callable,TypeVar

E = TypeVar('E')
R = TypeVar('R')

identity = lambda x:x
```

2. *Implementar la función map*

```
def map2(f:Callable[[E],R],iterable:Iterable[E]) -> \
      Iterable[R]:
    for e in iterable:
        yield f(e)
```

La hemos llamado map2 para no confundirla con la predefinida en Python.

3. *Implementar la función filter*

```
def filter2(p:Callable[[E],bool],iterable:Iterable[E]) -> \
      Iterable[E]:
    for e in iterable:
        if p(e):
            yield e
```

4. *Implementar la función accumulate*

```
def accumulate2(iterable:Iterable[E], \
      f:Callable[[R,E],R] = add, \
      initial:R=None)-> Iterable[R]:
    if initial is None:
        iterable = iter(iterable)
        a = next(iterable)
    else:
        a = initial
    yield a
    for e in iterable:
        a = f(a,e)
        yield a
```

5. *Dar una implementación para la función reversed*

```
def reversed2(iterable:Iterable[E])-> Iterable[E]:
    ls = list(iterable)
    i = len(ls) -1
    while i >= 0:
        yield ls[i]
        i = i -1
```

6. Implementar la función zip

```
def zip2(*iterables:Iterable[Iterable[Any]])->Iterable[Any]:
    iterables = [iter(it) for it in iterables]
    n = len(iterables)
    try:
        while True:
            ls = []
            for i in range(n):
                e = next(iterables[i])
                ls.append(e)
            yield tuple(ls)
    except StopIteration:
        return
```

7. Implementar la función enumerate

```
def enumerate2(iterable:Iterable[E], start:int=0) -> \
               Iterable[Tuple[int,E]]:
    i = start
    for e in iterable:
        yield (i,e)
        i = i +1
```

8. Implementar la función distinct

```
def distinct(iterable:Iterable[E]) -> Iterable[E]:
    s = set()
    for e in iterable:
        if e not in s:
            yield e
            s.add(e)
```

9. Implementar la función limit

```
def limit(iterable:Iterable[E], limit:int) -> Iterable[E]:
    i = 0
    for e in iterable:
        if i < limit:
            yield e
            i = i +1
        else:
            break
```

10. Implementar la función flat_map

```
def flat_map(iterable:Iterable[E],fm:Callable[[E], \
      Iterable[R]]=identity)-> Iterable[R]:
    for e in iterable:
        for pe in fm(e):
            yield pe
```

11. Implementación de la función iterate

```
def iterate(initial:E, predicate:Callable[[E],bool],\
      operator:Callable[[E],E]) -> Iterable[E]:
    e = initial
    while predicate(e):
        yield e
        e = operator(e)
```

Diseño e implementación de acumuladores

Los acumuladores proporcionados por Python tienen un equivalente construido con for. Veamos ejemplos

1. *Implementar la función sum*

```
def sum2(iterable, start = 0):
    a = start
    for e in iterable:
        a = a+e
    return a
```

2. *Implementar la función min*

```
def min2(iterable:Iterable[E], key:Callable[[E],R]=None, \
      default:E=None):
    if key is None:
        key = identity
    a = None
    for e in iterable:
        if a is None or key(e) < key(a):
            a = e
    if a is None:
        return default
```

3. Implementar la función max

```
def max2(iterable:Iterable[E], key:Callable[[E],R]=None, \
       default:E=None):
    if key is None:
        key = identity
    a = None
    for e in iterable:
        if a is None or key(e) > key(a):
            a = e
    if a is None:
        return default
```

4. Implementar la función all

```
def all2(iterable:Iterable[E]):
    a = True
    for e in iterable:
        if not e:
            a = False
            break
    return a
```

5. Implementar la función any

```
def any2(iterable:Iterable[E]):
    a = False
    for e in iterable:
        if not e:
            a = True
            break
    return a
```

6. Implementar la función sorted

```
def sorted2(iterable:Iterable[E], key:Callable[[E],R]=None, \
             reverse:bool= False)->List[E]:
    ls = list(iterable)
    n = len(ls)
    if key is None:
        key = identity
    for i in range(n):
        for j in range(n):
            if not reverse and key(ls[i]) < key(key(ls[j])):
                ls[i], ls[j] = ls[j], ls[i]
            if reverse and key(ls[i]) > key(key(ls[j])):
                ls[i], ls[j] = ls[j], ls[i]
    return ls
```

7. Implementar la función reduce

```
def reduce2(f:Callable[[R,E],R], iterable:Iterable[E], \
              initial:R=None)->R:
    if initial is None:
        iterable = iter(iterable)
        a = next(iterable)
    else:
        a = initial
    for e in iterable:
        a = f(a,e)
    return a
```

8. Implementar la función find_first

```
def find_first(iterable:Iterable[E], p:Callable[[E],bool]) ->
E | None:
    for e in iterable:
        if p(e):
            return e
    return None
```

9. Implementar la función first_and_last

Devuelve una tupla con el primero y el último

```
def first_and_last(iterable:Iterable[E],defaultvalue=None)\
       ->tuple[E,E]:
    first = last = next(iterable, defaultvalue)
    for last in iterable:
        pass
    return (first,last)
```

10. Implementar la función index_predicate

```
def index_predicate(iterable:Iterable[E],\
      predicate:Callable[[E],bool],default:int=-1)->int:
    for i,e in enumerate(iterable):
        if predicate(e):
            return i
    return default
```

Comprensión de listas, conjuntos, diccionarios y generadores

Las listas, los conjuntos y los diccionarios pueden crearse también por comprensión (en inglés, *comprehension*). Esto tiene mucha relación con la definición de conjuntos por comprensión de las matemáticas. Sea por ejemplo, en matemáticas esta expresión: $\{x^2 : x \in [3,7) | \ x\%2 == 1\}$. Esta expresión define un conjunto {9,25}, puesto que el intervalo es abierto por la derecha, con lo que solo comprende los números 3, 4, 5 y 6, pero la condición dice que solo tengamos en cuenta los impares, es decir, 3 y 5, y la expresión del principio dice que tomemos los cuadrados.

Las *expresiones por comprensión* son formas compactas de describir listas, conjuntos, diccionarios o generadores. Veremos más adelante la relación de estas expresiones con los iterables, sus transformaciones y la sentencia for.

Las expresiones por compresión para listas, conjuntos, diccionarios y generadores tienen formas similares.

Para listas

```
ls = [t(e) for e in it if f(e)]
```

Para conjuntos

```
st = {t(e) for e in it if f(e)}
```

Para diccionarios

```
dt = {t1(e):t2(e) for e in it if f(e)}
```

Para generadores

```
gn = (t(e) for e in it if f(e))
```

Como vemos las expresiones por comprensión tienen la misma forma para listas, conjuntos, diccionarios y generadores. Donde *t, t1,t2* son funciones de tipo *Callable[[E],R]*, *f* de tipo *Callable[[E],bool]*, *it* un iterable. La diferencia entre unas y otras es que una lista usa [], un conjunto {}, un diccionario también {} pero aparecen : para separar la clace y el valor y los generadores usan ().

Estas expresiones por comprensión son equivalentes a la secuenciación de una operación de filtro, filter, otra de trasnformación, map, y un acumulador. Asi tenemos las siguientes equivalencias:

```
ls = [t(e) for e in it if f(e)]
```

La expresión por comprensión para listas anterior es equivalente a:

```
it1 = filter(f, it)
it2 = map(t,it1)
ls = []
for e in it2:
       ls.append(e)
```

Igualemente para conjuntos

```
st = {t(e) for e in it if f(e)}
```

La expresión por comprensión para conjuntos anterior es equivalente a:

```
it1 = filter(f, it)
it2 = map(t,it1)
st = set()
for e in it2:
       st.add(e)
```

Para diccionarios

```
dt = {t1(e):t2(e) for e in it if f(e)}
```

La expresión por comprensión para diccionarios anterior es equivalente a:

```
it1 = filter(f, it)
it2 = map(t,it1)
dt = {}
for e in it2:
       dt[t1(e)] = t2(e)
```

Para generadores

```
gn = (t(e) for e in it if f(e))
```

La expresión por comprensión para generadores anterior es equivalente a:

```
it1 = filter(f, it)
gn = map(t,it1)
```

Veamos algunos ejemplos.

1. *Escribir una lista por comprensión que contenga los cuadrados de los enteros de 3 a 70 que son múltiplos de 3*

```
s = [x**2 for x in range(3, 70) if x % 3 == 0]
print(s)
```

2. *Escribir un diccionario por comprensión que a partir de una lista de nombres construya un diccionario con el nombre y su índice en la lista*

```
nombres = ["Miguel", "Ana", "José María", "Guillermo", \
       "María", "Luisa"]
ranking = {nombre: nombres.index(nombre) for nombre in \
       nombres}
```

3. *Construir un diccionario por comprensión que almacene la frecuencia de aparición de cada carácter de un texto*

```
texto = "este es un pequeño texto para probar la siguiente \
       definición por comprensión"
frecuencias_caracteres = {caracter: texto.count(caracter) \
       for caracter in texto if caracter!=" "}
print(frecuencias_caracteres)
```

4. *Un último ejemplo, en el que a partir de un texto construimos un diccionario con iniciales como claves y la lista de palabras como valor para cada clave*

```
texto = "este es un pequenyo texto para probar la siguiente \
definicion por comprension"
iniciales = {p[0] for p in texto.split()}
palabras = {p for p in texto.split()}
palabras_por_iniciales = {c: [p for p in palabras if p[0]==c]\
       for c in iniciales}
print(palabras_por_iniciales)
```

Veremos otras formas más eficientes de llevar a cabo el cálculo anterior mediante un acumulador específico para estos casos.

Ejemplos de equivalencias entre expresiones por comprensión y bucles for

Hemos visto expresiones por comprensión para listas, conjuntos, y diccionarios. Veamos ahora la equivalencia de estas expresiones con los bucles for.

1. *Lista por compresión y su equivalente*

```
ls = [x**2 for x in range(10,20) if x%2==0]
ls2 = []
for x in range(10,20):
    if x%2==0:
        ls2.append(x**2)
```

2. Conjunto por compresión y su equivalente

```
s = {x**2 for x in range(10,20) if x%2==0}
s2 = set()
for x in range(10,20):
    if x%2==0:
        s2.add(x**2)
```

3. Diccionario por compresión y su equivalente

```
d = {x:x**2 for x in range(10,20) if x%2==0}
d2 = {}
for x in range(10,20):
    if x%2==0:
        d2[x]=x**2
```

4. Aplanamiento de un iterable anidado y su equivalente

```
with open('datos_2.txt') as f:
    lector = csv.reader(f, delimiter = ',')
    lineas: List[List[str]] =  [linea for linea in lector]

suma2 = sum(int(e) for linea in lineas for e in linea)
suma2 = 0
for linea in lineas:
    for e in linea:
        suma2 = suma2 + int(e)
```

5. Equivalencia de enumerate

```
q2 = [x for i,x in enumerate(progresion_aritmetica(10,300,7))\
       if i%2==0]

q2 = []
i = 0
for x in progresion_aritmetica(10,300,7):
    if i%2==0:
        q2.append(x)
    i= i+1
```

6. Equivalencia de zip

```
ls = [1,2,3,4,5,6]
s = [e+x for e,x in zip(ls,range(10,3000,7))]

s = []
ls2 = list(range(10,3000,7))
for i in range(min(len(ls),len(ls2))):
       s.append(ls[i]+ls2[i])
```

Funciones de formateo de iterables

Los iterables deben ser formateados adecuadamento para ser presentandos en pantalla o incluso para convertirse en hileras de caracteres simplemente.

Vemos aquí una función para formatear iterables

```
def str_iterable(iterable:Iterable[E], \
       sep:str=',',prefix:str='{',suffix:str='}', \
       ts:Callable[[E],str]=str) ->str:
    return "{0}{1}{2}" \
       .format(prefix,sep.join(ts(x) for x in iterable),suffix)
```

Donde el primer parámetro es un iterable, sep es el separador entre dos elementos con una coma por defecto, prefix es el prefijo con { por defecto

y suffix el sufijo con } por defecto. La funcion ts transforma cada elemento a cadena de caracteres y por defecto tiene la fuinción str.

Funciones de lectura y escritura de ficheros

La lectura de ficheros es repetitiva por lo que es conveniente disponer de un conjunto de funciones para leer y escribir ficheros. Veamos algunas.

Una función para transformar un fichero en una lista de líneas. Se lee todo el fichero antes de ser procesado.

```
def lineas_de_fichero(file:str,encoding='utf-8') -> list[str]:
    with open(file,encoding=encoding) as f:
        lineas_de_fichero =  [linea.rstrip('\n') \
             for linea in f]
        return lineas_de_fichero
```

Una función para transformar un fichero con estructura csv, es decir líneas que pueden ser divididas en partes por un separador, en una lista de listas de partes de líneas. Se lee todo el fichero antes de ser procesado.

```
def lineas_de_csv(file:str, delimiter:str=",", \
      encoding='utf-8')-> list[list[str]]:
    with open(file,encoding= encoding) as f:
        lector = csv.reader(f, delimiter = delimiter)
        lineas_de_fichero =  [linea for linea in lector]
        return lineas_de_fichero
```

Una función para leer un fichero completo en una variable

```
def texto_de_fichero(file:str,encoding:str='utf-8') -> str:
    with open(file, "r", encoding=encoding) as f:
        texto = f.read()
        return texto
```

Una función para escribir una variable en un fichero

```
def write(file:str,texto:str) -> None:
    with open(file, "w", encoding='utf-8') as f:
        f.write(texto)
```

Una función para escribir un iterable en un fichero

```
def write_iterable(file:str,iterable:Iterable[str]) -> None:
    with open(file, "w", encoding='utf-8') as f:
        for ln in iterable:
            f.write(ln)
```

Una función para conocer el encoding de un fichero

```
def print_encoding(file:str)->str:
    with open(file,"rb") as f:
        data = f.read()
        enc = chardet.detect(data)
        return enc['encoding']
```

La mayoría de las funciones anteriores lee completamente en memoria los datos de un fichero antes de procesarlos. Esto puede ser un problema cuando el fichero sea muy grande. Podemos diseñar otras funciones que obtengan iterables de líneas o fragmentos de las mismas que podamos procesar sin cargar el fichero entero en memoria.

Una función que obtenga un iterable de las líneas de un fichero

```
def lineas_iterable(file:str,encoding:str='utf-8') -> \
      Iterable[str]:
    with open(file, "r", encoding=encoding) as f:
        for line in f:
            yield line.strip()
```

Una función que obtenga un iterable las líneas de un fichero convertidas en listas de partes separadas por el delimitador.

```
def iterable_de_csv(file:str, delimiter:str=",", \
             encoding='utf-8')-> Iterable[list[str]]:
    with open(file,encoding= encoding) as f:
        lector = csv.reader(f, delimiter = delimiter)
        for line in lector:
            yield line
```

Una función que obtenga un iterable de las partes de las líneas de un fichero separadas por el delimitador.

```
def iterable_de_csv_partes(file:str, delimiter:str=",",\
      encoding='utf-8')-> Iterable[str]:
    with open(file,encoding= encoding) as f:
        lector = csv.reader(f, delimiter = delimiter)
        for line in lector:
                for p in line:
                    yield p
```

Funciones para representaciones gráficas

Hay representaciones gráficas que se repiten frecuentemente:

- *Representar una función*. Se trata de representar una función en un dominio dado. Usamos *draw_function*
- *Representar un diagrama de tarta*. Usamos la función *draw_piechar*
- *Representar un diagrama de barras*. Usamos la función *draw_barchart*
- *Representar una multilínea*. Usamos la función *draw_multiline*

Veamos el código para cada una de ellas. Asumimos el siguiente código común:

```
from typing import Callable
import matplotlib.pyplot as plt
```

Una función para representar una función en en un intervalo a, b con incrementos c.

```
def draw_function(f:Callable[[float],float], \
              a:float,b:float,c:float)->None:
    plt.axes()
    n = int((b-a)/c)
    x = [a+i*c for i in range(0,n)]
    y = [f(v) for v in x]
    plt.plot(x, y)
    plt.show()
```

Una función para representar un diagrama de tartas

```
def draw_piechar(labels:list[str],sizes:list[int])->None:
    plt.pie(sizes, labels=labels)
    plt.axis('equal')
    plt.show()
```

Una función para representar un diagrama de barras

```
def draw_barchart(labels:list[str],sizes:list[int], \
      title:str='Diagramama de Barras',y_label:str='Eje Y'):
    y_pos = list(range(len(sizes)))
    plt.bar(y_pos,sizes, align='center', alpha=0.5)
    plt.xticks(y_pos,labels)
    plt.ylabel(y_label)
    plt.title(title)
    plt.show()
```

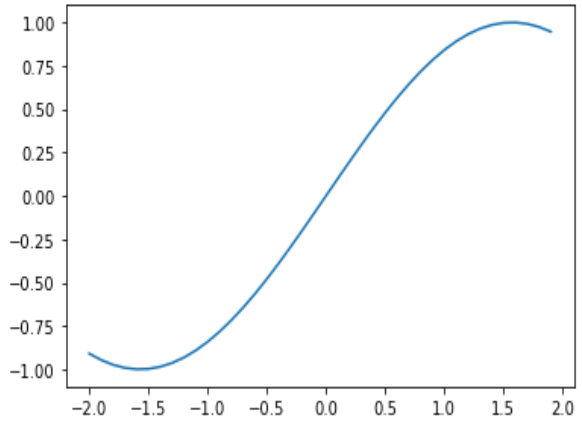
Una función para representar un diagrama de barras

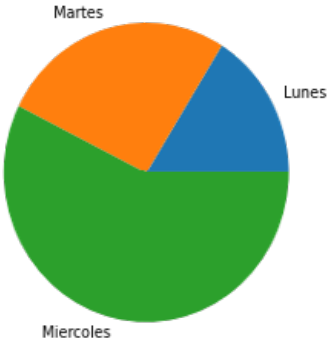
```
def draw_multiline(points:List[Tuple[float,float]], \
      y_label:str='eje y',x_label:str='eje_x', \
      title:str='Grafico'):
    plt.ylabel(y_label)
    plt.xlabel(x_label)
    plt.title(title)
    plt.plot([x[0] for x in points], [x[1] for x in points])
    plt.show()
```

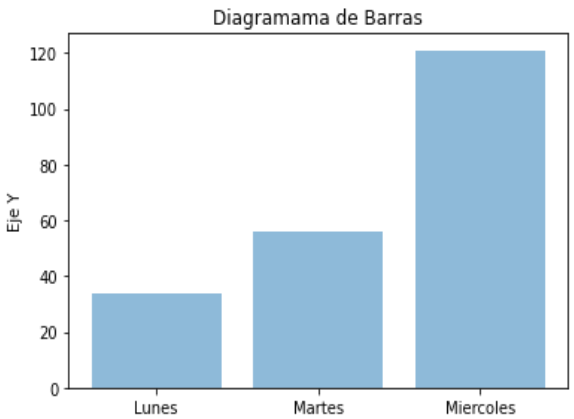
Que podemos usar

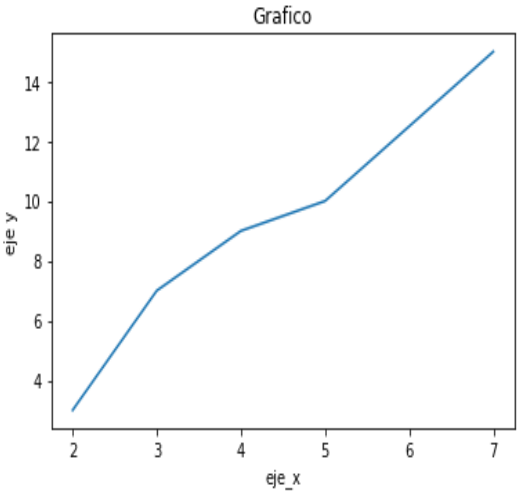
```
draw_function(lambda x:sin(x),a=-2,b=2,inc=0.1)
draw_piechar(['Lunes','Martes','Miercoles'],[34,56,121])
draw_barchart(['Lunes','Martes','Miercoles'],[34,56,121])
draw_multiline([(2,3),(3,7,),(4,9),(5,10),(7,15)])
```

Para obtener los siguients gráficos:









En el repositorio hay disponibles funciones similares para dibujar gráficos usando herramientas de Google.

Clases y tipos

Los lenguajes de programación vienen con un conjunto de tipos definidos. Hemos visto enteros, hileras de caracteres, listas, tuplas, diccionarios, etc.

Un tipo representa una entidad del mundo que no rodea. Por ejemplo un *Vector2D* que describe las propiedades de todos los vectores del plano de dos dimensiones. Definido un tipo podemos crear objetos de ese tipo. Cada objeto creado tendrá los métodos adecuados para preguntar por sus propiedades o modificarlas.

En los lenguajes de programación es necesario definir tipos nuevos para estructurar el código.

Un tipo tiene un nombre y un conjunto de propiedades que son sus características visibles.

Las propiedades pueden ser *básicas* o *derivadas*. El valor de las primeras no puede ser obtenidas a partir de otras. El valor de las propiedades derivadas se obtiene a partir de otras propiedades. Las propiedades básicas, en el caso del *Vector2D*, son *x, y*. Una propiedad derivada puede ser *distancia_al_origen*.

Una forma de definir tipos nuevos en Python es mediante tuplas con nombre. Pero hay otra forma más elaborada y aconsejable mediante una clase (*class*) etiquetada con *@dataclass* e indicando los atributos de la clase. Las propiedades básicas y derivadas se implementan mediante

métodos etiquetados con *@propertie*. Los métodos son de dos tipos: de instancia y de factoría. Los primeros llevan como primer parámetro *self*, los segundos no. Por *self* nos referimos al objeto actual.

En general cuando definimos un tipo nuevo definimos un conjunto de *atributos* del tipo. El conjunto de atributos define el *estado* de los objetos del tipo. Los atributos de un objeto son generalmente privados del objeto y a partir de ellos se definen las propiedades básicas y derivadas que son visibles y usables desde el exterior del objeto.

Una posibilidad sencilla, que usaremos en primer lugar, es dotar a la clase de tantos atributos como propiedades básicas y diseñar métodos con la etiqueta *@propertie* para las propiedades derivadas pero hay otras posibilidades como veremos.

Cuando definimos un tipo nuevo (mediante una clase) definimos también una igualdad entre dos objetos de esa clase. Por simplicidad asumimos que dos objetos son iguales si tienen sus propiedades básicas iguales. Esto implica que está definido e operador == para objetos de ese tipo.

También podemos definir un orden sobre estos objetos. Por simplicidad el orden vendrá definido por la primera propiedad básica, luego por la segunda, etc. La definición de orden la tendremos si ponemos *order = True* en los parámetros de *dataclass*. De esta forma definimos el *orden natural* para el tipo. Al hacerlo así están definidos los operadores *<, <=, >, >=, ==*.

El tipo que definamos puede ser *mutable* o *inmutable*. Esto se consigue haciendo *frozen=True* en los parámetros de dataclass.

La representación como hilera de caracteres se hace con la función *__str__*.

Los objetos nuevos se pueden crear mediante un *constructor* que será el nombre de la clase seguido de un parámetro por cada propiedad básica. Un tipo tiene también métodos de factoría que son mecanismos específicos para crear objetos. Los métodos de factoría están etiquetados con *@staticmethod*.

Antes de implementar un tipo los diseñamos. Veamos el diseño del tipo *Vector2D*:

Vector2D
Propiedades

- x: float, básica
- y: float, básica
- copy: Punto2D, derivada, *(x,y)*
- modulo: float, derivada, *sqrt{x^2+y^2}*
- distancia_a(p:Punto2D): float, derivada, *sqrt((x-p.x)^2+(y-p.y)^2)*

Representación

- (2.3,4.5)

Factoría

- of(x:float,y:float) -> Vector2D
- parse(text:str)-> Vector2D

Para implementar el tipo comenzamos por la clase

```
from __future__ import annotations
from math import sin, cos, radians, atan2, degrees, sqrt, acos
from dataclasses import dataclass
from us.lsi.tools import Preconditions

@dataclass(frozen=True,order=True)
class Vector2D:
    x: float
    y: float
```

Con el código anterior hemos declarado un tipo inmutable Vector2D con x,y como propiedades básicas y un orden natural establecido primero por x, luego por y.

La siguiente tarea es diseñar los métodos de factoría: of y parse. El primero construye un objeto usando el constructor.

```
@staticmethod
def of(x: float,y: float)->Vector2D:
        return Vector2D(x,y)
```

El constructor es la función Vector2D del mismo nombre de la clase. Un constructor va seguido de valores para cada una de las propiedades básicas.

El método de factoría parse convierte un cadena con el formato (2.3,4.5) en un vector.

```
@staticmethod
def parse(text:str)->Vector2D:
       text = text[1:-1]
       x,y = text.split(',')
       return Vector2D(float(x),float(y))
```

La representación de los valores del tipo

```
def __str__(self)->str:
        return '({0:.2f},{1:.2f})'.format(self.x,self.y)
```

Las propiedades copy y modulo se diseñan como métodos con el decorador @property. Estas propiedades son métodos de instancia y por eso llevan self como primer parámetro.

```
@property
def copy(self: Vector2D)->Vector2D:
       return Punto2D(self.x,self.y)

@property
def modulo(self)->float:
       return sqrt(self.x*self.x+self.y*self.y)
```

La implementación de una clase puede usar otros tipos implementados previamente. Una vez definida una clase disponemos de un tipo nuevo con el cual podemos crear objetos y aplicarles los métodos definidos.

```
p = Vector2D.of(2.3, 44.5)
print(p)
p2 = Vector2D.parse('(2.3,-4.5)')
p3 = Vector2D(2.3,6.7)
print(p == p2)
print(p.modulo)
print(p.x)
p4 = Vector2D.of(3.4,5.6)
p5 = Vector2D.of(3.,4.)
```

El mecanismo que hemos explicado antes, *class* con el decorador *@dataclass*, es adecuado para definir tipos de datos generalmente inmutables con una igualdad, un orden, un constructor y una representación por defecto definidos a partir de las propiedades básicas.

Si queremos más flexibilidad debemos diseñar una clase (class) sin el decorador *@dataclass*. Ahora tenemos que definir los atributos (el estado de los objetos) y a partir de ellos el constructor. Posteriormente definir los métodos para acceder o modificar las propiedades y entre ellos los métodos que definen la igualdad y el hashCode.

La flexibilidad que permite diseñar una clase sin el decorador @dataclass implica tener en cuenta muchos más detalles. Supongamos que queremos diseñar un tipo mutable *Fraccion* con sus propiedades numerador, denominador, la igualdad y los operadores que tienen las fracciones.

Ahora tenemos que diseñar los siguientes elementos:

- Los *atributos*: Definen el estado el objeto. Podemos escogerlos para que guarden las propiedades básicas. Su identificador será el nombre de la propiedad precedido de _. En el resto del cuerpo de la clase se accederá a los atributos mediante self.atributo
- El *constructor*: Es el mecanismo para construir objetos. El constructor se define mediante el método __init__. Dentro del cuerpo del constructor se definen los atributos.
- Los *métodos de factoría* serán métodos con el decorador *@staticmethod* que no llevan self como primer parámetro.
- Los *métodos individuales* serán métodos sin el decorador *@staticmethod* y con self como primer parámetro

- La *igualdad* se define con el método __eq__ y __hash__ y dota a los valores del operador ==
- La *representación* se define con el método __str__
- El *orden natural* con los métodos *__eq__, __ne__, __lt__ y el decorador* @total*ordering*. Definir el orden total dota a los valores de los operadores *<,<=,>,>=,==*

El diseño del tipo Fracción es

Fracción
Propiedades

- numerador, int
- denominador, int, >0
- __add__(f:Fraccion):Fraccion,
- __sub__(f:Fraccion):Fraccion,
- __neg__():Fraccion,

Operaciones

- set_numerador(n:int)
- set_denominador(d:int)

Representacion

- numerador/denominador

Igualdad

- f1 = f2
  f1.numerador*f2.denominador = f2.numerador*f1.denominador

Orden natural

- f1 < f2 ==
  f1.numerador*f2.denominador < f2.numerador*f1.denominador

Métodos de factoría

- of(n,d):Fraccion
- random(lm:int):Fraccion

```
def mcd(a:int, b:int)->int:
    checkArgument(a>=0 and b>0,'El coeficiente a debe ser \
       mayor o igual que cero y b mayor que cero y son: \
       a = {0}, b = {1}'.format(a,b))
    while b > 0:
        a, b = b, a%b
    return a
```

Los valores de una fración los vamos a guardar simplificados. Para ello necesitaremos la función anterior que calcule el máximo común divisor.

Comencemos declarando la clase

```
from __future__ import annotations
from functools import total_ordering
from random import randint
from us.lsi.tools.Preconditions import checkArgument

@total_ordering
class Fraccion:
    def __init__(self, n:int, d:int=1):
        checkArgument(d != 0, \
             'El denominador no puede ser cero y es {}'\
             .format(d))
        self._numerador = n
        self._denominador = d
        self.__normaliza()
```

Con el método *__init__* hemos declarado dos atributos privados: _numerador y _denominador. El método __init__ define el contructor de la forma *Fraccion(n,d)*. Los valores de la fracción los queremos guardar normalizados. Para ello usamos el método privado, empieza por __, __normaliza().

```
def __normaliza(self)->None:
        n = self._numerador
        d = self._denominador
        sg = 1 if d > 0 else -1
        m = mcd(abs(n),abs(d))*sg
        self._numerador = n//m
        self._denominador = d//m
```

Podemos observar que los valores *n, d* se dividen por el máximo común divisor y el signo de *d* antes de guardarlos en los correspondientes atributos. Guardamos la fracción normalizada.

La siguiente tarea son los métodos de factoría:

```
@staticmethod
def of(n: int,d:int=1)->Fraccion:
       return Fraccion(n,d)

@staticmethod
def random(lm:int)->Fraccion:
       n = randint(-lm,lm)
       d = randint(1,lm)
       return Fraccion(n,d)
```

La representación será de la forma

```
def __str__(self):
        if self._denominador == 1:
            return '{0:d}'.format(self._numerador)
        else:
            return '{0:d}/{1:d}' \
                    .format(self._numerador,self._denominador)
```

La igualdad se consigue implementando el método __eq__.

```
def __eq__(self, other):
        if isinstance(other, Fraccion):
            return self._numerador == other._numerador and \
                    self._denominador == other._denominador
        return False
```

La noción de orden la conseguimos mendiante los métodos *__ne__, __lt__* y el decorador *@totalordering*.

```
def __ne__(self, other):
        return not (self == other)

def __lt__(self, other):
        return self._numerador*other._denominador < \
             self._denominador*other._numerador
```

El hashcode mediante el método *__hash__*.

```
def __hash__(self):
        return  hash(self._numerador)*31 + \
             hash(self._denominador)
```

Con los métodos anteriores los valores del tipo fracción pueden ser comparados con los operadores relacionales usuales: *==, !=, <, <=, >, >=.* En Python si tu tipo implementa determinados operadores sus valores pueden ser operados con los operadores asociados. Asi el método *__eq__* tiene asociado el operador ==, *__ne__* tiene asociado !=, *__lt__* se corresponde con <, *add* se asocia con +, etc.

La suma de dos fracciones se implementa con el método *__add__*. Igualemente podemos diseñar los métodos *__sub__*, *__mult__*, *__truediv__*, y *__neg__*. El escoger estos nombres permite usar los operadores binarios +, -, *, / y el unario -.

```
def __add__(self,other)->Fraccion:
        n = self.numerador*other._denominador \
             +self.denominador*other.numerador
        d = self.denominador*other.denominador
        resultado = Fraccion(n,d)
        return resultado
```

En la clase _numerador y _denominador son atributos privados no accesibles desde el exterior de la clase. A partir de ellos diseñamos las propiedades.

```
@property
def numerador(self):
        return self._numerador
@property
def denominador(self):
        return self._denominador
```

Si queremos que el tipo sea mutable podemos diseñar operaciones: métodos para modificar las propiedades

```
def set_numerador(self,n):
        self._numerador = n
        self.__normaliza()

def set_denominador(self,d):
        self._denominador = d
        self.__normaliza()
```

Que podemos usar como

```
f1 = Fraccion.of(6)
print(f1.numerador)
print(f1.denominador())
f2 = Fraccion.of(3, 6)
print(f1 >= f2)
f3 = Fraccion.of(2, 12)
print(f3)
```

Podemos modificar las propiedades al ser un tipo mutable.

```
f3.set_denominador(10)
print(f3)
print(f3+f2)
print(f3-f2)
print(f3*f2)
print(f3/f2)
ls = (Fraccion.random(1000) for _ in range(50))
print(str_iterable(ls))
```

Las ventajas en inconvenientes de diseñar un nuevo tipo con el decorador *@dataclass* o no pueden verse en los ejemplos anteriores. Si no usamos *@dataclass* tenemos todas las posibilidades del uso de *class* en Python: podemos definir los atributos que decidamos, a partir de ahí las propiedades, modificables o no, podemos elegir el criterio de la igualdad y del hashCode y el criterio de orden que decidamos. Pero debemos implementar el constructor, los métodos que definen las propiedades, el método que define la igualdad, el hashCode, el orden natural y la representación.

El uso de @dataclass ya nos da implementado el constructor, el criterio de igualdad, el orden natural, y la representación. Pero estos están definidos por las propiedades básicas y su orden.

Es importante recordar que el uso del decorador *@propertie* hace que los métodos etiquetados se llamen si paréntesis.

Clases abstractas y herencia

En muchos casos los tipos que diseñamos comparten un conjunto de propiedades u operaciones comunes. Estas operaciones tienen una cabcera común pero un código específico para cada tipo. Para conseguir ese objetivo diseñamos un tipo abstracto. Es decir un tipo cuyos métodos no tienen código.

Sean los tipos geométricos *Punto2D, Segmento2D, Circulo2D, Poligono2D* que comparten propiedades y métodos como ahora veremos. Queremos, además, usar listas y agregados de estos tipos geométricos y a todos los objetos de los agregados aplicar los métodos comunes. ¿Cómo abordar este problema?

La solución es diseñar un objeto abstracto, *Objeto2D*, que tenga los métodos compartidos y si la hubiera la funcionalidad compartida. El resto de los tipos geométricos heredarán los métodos de Objeto2D e incluirán su propia funcionalidad en cada caso. Veamos.

```
from __future__ import annotations
from abc import abstractmethod

class Objeto2D:
    @abstractmethod
    def traslada(self, v:Vector2D) -> Objeto2D:
        pass
      …
```

Hemos declarado la clase Objeto2D con sus métodos abstractos, etiquetados con *@abstractmethod.* El tipo *Punto2D* hereda de Objeto2D el método traslada entre otros.

```
@dataclass(frozen=True,order=True)
class Punto2D(Objeto2D):
    x: float
    y: float

    def traslada(self,v:Vector2D)->Punto2D:
        return Punto2D(self.x+v.x,self.y+v.y)
    …
```

El tipo *Punto2D* hereda todos los métodos de Objeto2D tengan funcionalidad o los implementa poniendo los detalles necesarios. De la misma forma ocurre con segmento. Pero ahora la funicionalidad de traslada es diferente.

```
@dataclass(frozen=True,order=True)
class Segmento2D(Objeto2D):
    p1: Punto2D
    p2: Punto2D

    def traslada(self, v:Vector2D) -> Segmento2D:
        return Segmento2D(\
             self.p1.traslada(v), self.p2.traslada(v))
      …
```

De la misma forma podriamos diseñar los tipos *Poligono2D, Circulo2D*. Todos herededan de Objeto 2D. Todos tienen un método traslada pero cada uno pone su funcionalidad específica para el método.

La clase *Objeto2D* podría tener, además, métodos con una funcionalidad concreta que serían heredados por todas las clases hijas. Esta es una forma de reutilizar funcionalidad.

Más adelante veremos con detalle un ejemplo de geometría que nos permitirá entrar más en detalle en este tema.

Diccionarios

Los *diccionarios* son un agregado de datos, como lo son las listas o las tuplas. La principal característica que los diferencia de otros tipos de agregados es que los valores contenidos en un diccionario están *indexados* mediante claves. Esto significa que para acceder a un valor contenido en un diccionario, debemos conocer la clave correspondiente, de manera parecida a como para acceder a un elemento concreto de una lista o una tupla necesitamos conocer la posición que ocupa dicho elemento, su índice.

A diferencia de las listas y las tuplas, que se indexan mediante números enteros en el rango *0..n-1*, siendo *n* la longitud de la lista o la tupla, los diccionarios se indexan por una clave que no tiene que ser de tipo *int*.

Veamos algunos ejemplos de inicialización y acceso a elementos de un diccionario, comparándolo con una tupla:

```
datos_personales_tupla = ("Miguel", "González Buendía", 24, \
      1.75, 72.3)
# Acceso al elemento indexado en la posición 0
print(datos_personales_tupla[0])

datos_personales_diccionario = {"nombre": "Miguel", \
      "apellidos": "González Buendía", "edad": 24, \
      "altura": 1.75, "peso": 72.3}
# Acceso al elemento indexado con la clave "nombre"
print(datos_personales_diccionario["nombre"])
```

Tal como se ve en el ejemplo, los diccionarios son una alternativa al uso de tuplas para representar información heterogénea sobre algún tipo de entidad. La ventaja fundamental es que mientras la tupla nos obliga a recordar la posición que ocupa cada uno de los datos de la entidad, para poder utilizarla en el resto del código, el acceso a los datos a través del diccionario se hace mediante claves que son más fáciles de recordar, repercutiendo en un código más legible.

Los diccionarios son *mutables*, lo que significa que podemos añadir o eliminar parejas clave-valor, como veremos más adelante. Los valores pueden ser de cualquier tipo; sin embargo, *las claves deben ser obligatoriamente de algún tipo inmutable.* Lo más frecuente es que sean cadenas o números, o bien tuplas formadas por cadenas y/o números.

Inicialización de diccionarios

Existen múltiples opciones para inicializar un diccionario. A continuación se muestran distintos ejemplos:

Diccionario vacío, mediante función dict

```
diccionario = dict()
```

Diccionario vacío, mediante llaves

```
diccionario = {}
```

Mediante una secuencia de tuplas (cada tupla representa una pareja clave-valor)

```
diccionario = dict([("clave1", "valor1"), \
       ("clave2",    "valor2"), ("clave3", "valor3")])
```

También podemos pasar cada pareja clave-valor como un parámetro por nombre a la función dict. En este caso, las claves siempre serán cadenas

```
diccionario = dict(clave1 = "valor1", clave2 = "valor2",\
clave3 = "valor3")
```

Si tenemos las claves y las tuplas en dos secuencias, podemos usar zip

```
claves = ["clave1", "clave2", "clave3"]
valores = ["valor1", "valor2", "valor3"]
diccionario = dict(zip(claves, valores))
```

Mediante las llaves, podemos especificar una serie de parejas clave-valor. Esta es quizás la opción más frecuente cuando se quiere inicializar un diccionario con unos valores conocidos.

```
diccionario = {"clave1": "valor1", "clave2": "valor2", \
       "clave3": "valor3"}
```

Aunque al mostrar los diccionarios las claves y valores asociados aparecen en el mismo orden en que fueron escritos al inicializar el diccionario, dichas parejas clave-valor no tienen un orden determinado dentro del diccionario. Por tanto, *dos diccionarios serán iguales si tienen las mismas parejas,* independientemente del orden en que fueran insertadas en el diccionario:

```
diccionario2 = {"clave2": "valor2", "clave3": "valor3", \
       "clave1": "valor1"}
print(diccionario==diccionario2)
```

En los ejemplos anteriores tanto las claves como los valores son de tipo cadena. Por supuesto, podemos usar otros tipos, tanto para las claves como para los valores (recordando que los tipos de las claves deben ser inmutables, como señalamos antes). Es frecuente que los valores sean a su vez de algún tipo agregado.

Por ejemplo el glosario de un libro, indicando las páginas en las que aparecen distintos conceptos. Las claves son de tipo cadena y los valores de tipo lista

```
glosario = {'programación estructurada':[14,15,18,24,85,86],\
       'funciones': [2,3,4,8,9,10,11,14,15,18], ...}
```

Operaciones con diccionarios

Repasaremos en esta sección las operaciones más comunes con diccionarios.

Para *acceder a un valor a partir de una clave*, podemos utilizar los corchetes (de forma parecida a como accedemos a los elementos de una lista) o el método get. Siendo el diccionario:

```
diccionario = {"clave1": "valor1", "clave2": "valor2",\
       "clave3": "valor3"}
```

Acceso a un valor a partir de una clave mediante corchetes o mediante método get.

```
print(diccionario["clave1"])
print(diccionario.get("clave1"))
```

Si utilizo en los corchetes una clave que no existe en el diccionario, se produce un error

```
print(diccionario["clave4"])
```

Si utilizamos el método get con una clave no existente, obtenemos un valor por defecto (None):

```
print(diccionario.get("clave4"))
```

Podemos cambiar el valor por defecto que devuelve get cuando no encuentra una clave, mediante un segundo parámetro:

```
print(diccionario.get("clave4","noexiste"))
```

Para *añadir una nueva pareja clave-valor o modificar el valor para una clave ya existente* podemos usar una instrucción de asignación, junto con el operador de acceso anterior (los corchetes):

Inserción de una nueva pareja

```
diccionario["clave4"] = "valor4"
```

Modificación del valor para una clave existente

```
diccionario["clave1"] = "valor1_modificado"
```

Si queremos volcar toda la información contenida en un diccionario en otro diccionario, usaremos el método *update*. Debemos tener en cuenta que al hacer esto puede que estemos sobrescribiendo los valores asociados a algunas claves del diccionario que estamos actualizando; esto ocurrirá cuando en el diccionario que estamos volcando haya claves iguales a las claves del diccionario que estamos actualizando:

```
diccionario2 = {"clave4": "valor4_modificado", \
       "clave5": "valor5", "clave6": "valor6"}

diccionario.update(diccionario2)
print(diccionario)
```

Si usamos la función predefinida *len* sobre un diccionario, obtenemos el número de parejas clave-valor que contiene el diccionario:

```
print("Número de items que tiene el diccionario:",\
len(diccionario))
```

Para *eliminar una pareja clave-valor*, utilizamos la instrucción *del*. Borrado de una pareja clave-valor

```
del diccionario["clave4"]
```

Si intentamos borrar una clave inexistente, obtenemos un error

```
del diccionario["clave4"]
```

Podemos borrar todo el contenido de un diccionario, mediante el método clear:

```
diccionario.clear()
```

Para acabar con las operaciones básicas, podemos consultar la pertenencia de una clave a un diccionario mediante el operador *in*, que puede aparecer combinado con el operador *not*:

```
if "clave1" in diccionario:
    print("Existe clave1")
if "clave4" not in diccionario:
    print("No existe clave4")
```

Iterables proporcionados por un diccionario

En ocasiones necesitaremos realizar alguna tarea utilizando únicamente las claves o los valores de un diccionario. Para obtener todas las claves o los valores de un diccionario usaremos los métodos keys y values que son dos iterables:

```
diccionario = {"clave1": "valor1", "clave2": "valor2", \
       "clave3": "valor3"}
print(set(diccionario.keys()))
print(set(diccionario.values()))
```

También podemos *obtener las parejas clave-valor*, en forma de tuplas, mediante el método *items*:

```
print(set(diccionario.items()))
```

Si utilizamos un diccionario en una instrucción *for ... in ...*, obtendremos en cada iteración una clave del diccionario. Es decir por defecto el iterable asociado a un diccionario proporciona un iterador que recorre sus claves.

```
for clave in diccionario:
    print(clave)
```

El método *items* proporciona otro iterador que recorre las tuplas formadas por los pares clave-valor del diccionario:

```
for clave, valor in diccionario.items():
    print(clave, valor)
```

El método values da un iterador que proporciona los valores del diccionario

```
for valor in diccionario.values():
    print(valor)
```

El tipo Counter

El tipo *Counter* es una especialización del tipo diccionario *dict*, es decir hereda de él, en la que los valores siempre son de tipo entero. Están pensados para representar frecuencias.

Supongamos que queremos saber cuántas veces aparecen en una lista cada uno de los elementos que la componen. Este problema aparece frecuentemente de una forma u otra como paso intermedio para resolver muchos algoritmos. Es por ello por lo que ya viene resuelto por el tipo *Counter*.

Un *Counter* se puede construir a partir de un iterable

```
from collections import Counter
import random

iterable = (random.randint(0,100) for _ in range(100))
frecuencias = Counter(iterable)
print(frecuencias)
```

Otro ejemplo

```
r = [random.randint(0,100) for _ in range(50)]
print(sorted(Counter(r).items(),reverse = True))
```

Aquí debemos tener en cuenta que los items del counter forman un iterable de tuplas donde la primera componente son los elementos del iterable y la segunda su frecuencia. La ordenación se hace con respecto al orden natural de las tuplas que es el orden natural de su primera componente. Las tuplas están ordenadas seún los números del iterable de mayor a menor.

Podemos ordenar la salida con respecto a las frecencias

```
print(sorted(Counter(r).items(), key = lambda x:x[1],\
reverse = True))
```

Podemos acceder a la frecuencia de un valor concreto preguntándole al contador por esa clave.

```
print(contador[6])
```

A diferencia de los diccionarios, si pregunto por una clave que no existe no obtengo KeyError, sino que me devuelve el valor 0.

```
print(contador[7])
```

Un Counter se puede actualizar a partir de otro diccionario. No es necesario que este otro diccionario sea un *Counter*, sino que puede ser un diccionario normal, siempre y cuando cumpla que sus valores sean de tipo entero:

```
votos_almeria = {'PP':27544, 'PSOE':20435}
votos_sevilla = {'PP':23544, 'PSOE':29435}

contador = Counter(votos_almeria)
contador.update(votos_sevilla)
print(contador)
```

El código anterior en realidad está "sumando" los conteos expresados en los diccionarios *votos_almeria* y *votos_sevilla*. El tipo *Counter* también nos permite usar el operador + para realizar la misma tarea:

```
contador1 = Counter(votos_almeria)
contador2 = Counter(votos_sevilla)
print(contador1+contador2)
```

La diferencia es que ahora se obtiene un nuevo objeto *Counter*, cada vez que se realiza la operación +. Por tanto, si necesitamos acumular muchos contadores en un solo, siempre será más eficiente utilizar el método *update*.

El método *most_common* de *Counter* devuelve los elementos más frecuentes junto con sus frecuencias. Como ejmplo podemos ver lo número que más se repiten en un fichero cuyas líneas son números separados por comas.

```
it = lineas_de_csv("datos_2.txt",encoding='ISO-8859-1')
print(Counter(flat_map(it)).most_common(5))
```

El tipo OrderedDict

Es un subtipo de diccionario que recuerda el orden de inserción de las claves. Una forma de crear un diccionario ordenado es a partir de un iterable de tuplas ordenado adecuadamente.

```
def palabras_frecuentes(file:str)->str:
    counter = Counter(palabras(file))
    return str_dictionary(OrderedDict(sorted(counter.items(),\
       key=lambda t: t[0])[:50]))
```

Donde palabras es una función que nos devuelve lun iterable con las palabras de un fichero separadas por delimitadores. El el ejercicio de Cálculos sobre un Libro veremos más detalles.

Operaciones de agrupación en iterables

Hay otros cálculos sobre iterables que se repiten frecuentemente y producen diccionarios:

- *Agrupamientos en listas*. Se trata de agrupar los elementos de un iterable es listas según un criterio. Usamos para ello la función *grouping_list*
- *Agrupamientos en conjuntos*. Se trata de agrupar los elementos de un iterable es listas según un criterio. Usamos para ello la función *grouping_set*
- *Reducción por grupos*. Se trata de acumular los elementos de cada grupo de un iterable usando un operador binario y partiendo del primer elemento. Usamos para ello la función *grouping_reduce*
- *Tamaño de grupos*. Se trata de contar, o sumar las veces que aparecen, los elementos de cada grupo de un iterable. Usamos para ello la función *groups_size*.

Veamos los detalles. Asumimos el código

```
from typing import List, Set, Dict, Iterable, TypeVar,
Callable
import random

K = TypeVar('K')
V = TypeVar('V')
E = TypeVar('E')
R = TypeVar('R')

identity = lambda x:x
```

La función *grouping_list* agrupa los elementos de un iterable es grupos formados por listas. En cada grupo se incluyen los valores calculados por la función *fvalue* a partir de los elementos del grupo. Los elementos en cada grupo tienen la misma clave calculada con la función *fkey.*

```
def grouping_list(iterable:Iterable[E],
              fkey:Callable[[E],K],
              fvalue:Callable[[E],V]=identity) -> \
       Dict[K,List[V]]:
    return grouping_reduce(iterable,fkey,lambda x,y:x+y,
              lambda x: [fvalue(x)])
```

De manera similar tenemos la función *grouping_set*

```
def grouping_set(iterable:Iterable[E],
              fkey:Callable[[E],K],
              fvalue:Callable[[E],V]=identity) -> \
       Dict[K,Set[V]]:
    return grouping_reduce(iterable,fkey,lambda x,y:x|y,
              lambda x: {fvalue(x)})
```

La función *grouping_reduce* obtiene un diccionario formado por pares cuya clave está calculada por la función fkey y los valores son los resultados de acumular todos los valores en el grupo mediante el operador binario *op* y tras ser transformados por la función *fvalue*.

```
def grouping_reduce(iterable:Iterable[E], \
              fkey:Callable[[E],K],op:Callable[[V,V],V], \
              fvalue:Callable[[E],V]= identity) -> Dict[K, E]:
    a = {}
    for e in iterable:
        k = fkey(e)
        if k in a:
            a[k] = op(a[k],fvalue(e))
        else:
            a[k] = fvalue(e)
    return a
```

La función *frecuencies* obtiene un diccionario con pares formados por las claves de los grupos de un iterable y su número de elementos. Es similar a tipo Counter.

```
def groups_size(iterable:Iterable[E],
              fkey:Callable[[E],K],
              fsum:Callable[[E],int]=lambda e:1)->Dict[K,int]:
    return grouping_reduce(iterable,fkey,op=lambda x,y:x+y, \
              fvalue= fsum)
```

Los resultados de la función anterior devuelven un diccionario que se puede convertir en un *Counter* visto anteriormente o en un *OrderedDict*.

La función str_dictionary formatea un diccionario.

```
def str_dictionary(dictionary:Dict[K,V],
       sep:str='\n',prefix:str='',suffix:str='') ->str: \
    ts = lambda x:'({}:{})'.format(str(x[0]),str(x[1]))
    return "{0}{1}{2}".format(prefix,sep.join(ts(x) \
       for x in sorted(dictionary.items(),
              key=lambda x:x[0])),suffix)
```

Los parámetros sep, prefix y suffix son, respectivamente, el separador, el prefijo y el sufijo.

Veamos algunos ejemplos. Dada un iterable como

```
import random
it: List[int] = (random.randint(0,100) for _ in range(0,100))
```

1. *Obtener los grupos de un iterable que se forman con los conjuntos elementos según su resto por 7*

```
print(str_dictionary(grouping_set(ls,fkey=lambda x:x%7)))
```

2. *Obtener los grupos de un iterable que se forman con las listas de elementos según su resto por 7*

```
print(str_dictionary(grouping_list(ls,fkey=lambda x:x%7)))
```

3. *Obtener el número de elementos en los grupos de un iterable que se forman al dividir por 7*

```
print(str_dictionary(frecuencies(ls,fkey=lambda x:x%7)))
```

4. *Obtener un diccionario de pares cuyas claves son los restos por 7 y cuyos valores la suma de los elementos del grupo*

```
print(str_dictionary(grouping_reduce(ls,\
       fkey=lambda x:x%7,op=lambda x,y:x+y)))
```

5. *Obtener un diccionario de pares cuyas claves son los restos por 7 y cuyos valores son booleanos que indican si todos los elementos del grupo son pares*

```
print(str_dictionary(grouping_reduce(ls,\
       fkey=lambda x:x%7,op=lambda x,y:x and y%2==0)))
```

6. *Obtener un diccionario de pares cuyas claves son los restos por 7 y cuyos valores son los mínimos del grupo correpondiente*

```
g = grouping_reduce(ls,fkey = lambda x: x%7, \
       op = lambda x,y: min(x,y))
```


Conjuntos

Los conjuntos son el último tipo de agregado que estudiaremos en este libro. Sus principales características son:

•*Son mutables*.

- *No admiten duplicados*. Si insertamos un nuevo elemento que ya existía en el conjunto, simplemente no se añade.
- *Los elementos no tienen una posición asociada*, como si tenían en las listas o en las tuplas. Es decir no son indexables. Por tanto, podemos recorrer los elementos de un conjunto, o preguntar si contiene a un elemento determinado, pero no acceder a una posición concreta.

Los conjuntos, como ya hemos visto, se pueden definir por compresión de forma similar a listas y diccionarios de la forma:

```
s = {exp(x) for s in iterable if p(x)}
```

Operaciones sobre conjuntos

Para inicializar un conjunto, podemos hacerlo usando las llaves, o la función *set*. Un conjunto vacío debe inicializarse meiante set(), puesto que {} entonces Python se entiende que estamos inicializando un diccionario vacío.

```
conjunto = set()
```

Inicializar un conjunto explícitamente

```
conjunto = {1, 2, 3}
```

Inicializar un conjunto a partir de los elementos de un iterable

```
lista = [1, 5, 5, 2, 2, 4, 3, -4, -1]
conjunto = set(lista)
```

Podemos observar que al inicializar un conjunto a partir de un iterable se eliminan los duplicados. Precisamente este es uno de los usos más habituales de los conjuntos: obtener los valores distintos contenidos en un iterable.

Los conjuntos son iterables. A diferencia de las listas, no podemos saber en qué orden se recorrerán sus elementos:

```
for elemento in conjunto:
    print(elemento)
```

También podemos utilizar el *operador de pertenencia in*, para preguntar por la pertenencia de un elemento al conjunto. Aunque esto es algo que podíamos hacer con las listas, en el caso de los conjuntos la operación es mucho más eficiente. Por tanto, si en un algoritmo se realizan una gran cantidad de operaciones de pertenencia, puede ser apropiado volcar los elementos en un conjunto en lugar de en una lista.

Todas las operaciones matemáticas entre conjuntos están implementadas en Python mediante operadores. En concreto, podemos hacer:

- *Unión de conjuntos*, mediante el operador |.
- *Intersección de conjuntos*, mediante el operador &.
- *Diferencia de conjuntos*, mediante el operador -.
- *Diferencia simétrica de conjuntos*, mediante el operador ^.

Ejemplos

```
a = {1, 2, 3, 4, 5, 6}
b = {4, 5, 6, 7, 8}

print("Unión:", a | b)
print("Intersección:", a & b)
print("Diferencia:", a - b)
print("Diferencia simétrica:", a ^ b)
print({7,8,1,2,3} == a ^ b)
print(2 in b-a)
print(a & b >= a | b)
```

También es posible preguntar si un conjunto es un subconjunto de otro, con el operador <= (o con el operador <, para preguntar si es un subconjunto propio).

Igualmente podemos usar los operadores > y >= para averiguar si un conjunto es un superconjunto (propio) de otro.

```
a = {1, 2, 3, 4, 5, 6}
b = {1, 2, 3}

print("¿Es b un subconjunto de a?", b <= a)
print("¿Es a un subconjunto de sí mismo?", a <= a)
print("¿Es a un subconjunto propio de sí mismo?", a < a)
```

Además de las operaciones propias de conjuntos, también pueden usarse las operaciones para iterables.

```
print("Tamaño del conjunto a:", len(a))
print("Suma de elementos de a:", sum(a))
print("Mínimo de a:", min(a))
print("Máximo de a:", max(a))
print("Elementos de a, ordenados:", sorted(a))
```

Los operadores <, <=, > y >= también se pueden usar entre listas.

```
lista1 = [1,2,3,4]
lista2 = [1,2]
lista3 = [3,4]
lista4 = []
print(lista2 < lista1)
```

El tipo SortedSet

El un subtipo de conjunto que mantiene ordenados los objetos que lo co
mponen.

Diseño de rutas

En cada problema de una determinada complejidad la primera tarea es abordar el diseño de tipos que necesitaremos. Esta tarea es básica para una programación de calidad. También lo es el diseño de funciones reutilizables.

Descripción del problema

Las rutas GPS (también llamadas tracks) contienen información sobre los puntos de un determinado trayecto. Casi cualquier dispositivo que tenga GPS (móviles, relojes, pulseras fitbit, ...) permite registrar esta información. Trabajaremos con un formato simple: CSV. Los datos que acompañan el ejercicio se corresponden con una ruta real.

Queremos hacer una serie de cálculos sobre los datos de entrada. Todos esos cálculos los acumularemos en tipo Ruta descrito más abajo.

El formato de entrada con el que trabajaremos contempla una línea por cada punto del trayecto que incluye cuatro informaciones:

- *tiempo* en el que fue tomada la medición en horas, minutos y segundos
- *latitud* del punto en el que fue tomada la medición en grados
- *longitud* del punto en el que fue tomada la medición en grados
- *altitud* del punto en el que fue tomada la medición en metros

He aquí un fragmento de dicho fichero con las cinco primeras líneas:

```
00:00:00,36.74991256557405,-5.147951105609536,712.2000122070312
00:00:30,36.75008556805551,5.148005923256278,712.7999877929688
00:01:30,36.75017642788589,-5.148165263235569,714.00
0:02:04,36.750248931348324,5.148243047297001,714.5999755859375
00:02:19,36.750430315732956,-5.148255117237568,715.0
```

Distancia a lo largo de la superficie de la tierra según la fórmula de Harvesine:

$$a = sin^2(\Delta\varphi/2) + cos\ \varphi_1 * cos\ \varphi_2 * sin^2(\Delta\lambda/2) \qquad (1)$$

$$d = 2\ R\ atan2(a, \sqrt{1-a})$$

Donde φ_1, φ_2 son las las latitudes de los puntos, $\Delta\varphi$ la diferencia de latitudes y $\Delta\lambda$ la diferencia de longitudes

Tipos

Los tipos adecuados para resolver el problema son: *Coordenadas2D, Coordenadas3D, Marca, Intervalo y Ruta*. Veamos para cada tipo su diseño y posteriormente su implementación.

Los tipos *Coordenadas2D, Coordenadas3D* los diseñamos para reutilizarlos en otros problemas.

Los objetos de tipo *Marca* nos servirán para representar la ubicación de un corredor en el espacio (sus coordenadas) y en un momneto del tiempo.

Un objeto de tipo *Intervalo* representa dos marcas consecutivas.

Una ruta es una secuencia de marcas.

Coordenadas2D

Representa las coordenadas geográficas de un punto sobre la superficie de la tierra.

Propiedades: Inmutable

- latitud: float, básica, en grados,
- longitud: float, básica, en grados
- copy: Coordenada2D, derivada
- distance(other: Coordenada2D): float, derivada, *formula de harvesine*
- es_cercana(other:Coordenada2D, d:float): bool, derivada,

Invariante

- $-90 \leq latitud \leq 90$
- $-180 \leq longitud \leq 180$
- es_cercana(other,d) = *distance(other) < d*

Representación

- (longitud,latitud)

Factoría

- parse(text:str)-> Coordenadas2D,en grados
- of(latitud:float,longitude:float)-> Coordenadas2D
- center(coordenadas: List[Coordenada2D]) -> Coordenada2D

Cada tipo que diseñamos lo implementamos mediante una clase:

```
from __future__ import annotations
from dataclasses import dataclass, asdict, astuple
from math import sin, cos, sqrt, atan2, radians
from us.lsi.tools.Preconditions import checkArgument

@dataclass(frozen=True,order=True)
class Coordenadas2D:
    latitud: float
    longitud: float
```

La clase tiene como campos las propiedades básicas del tipo que queremos implementar. En este caso latitud y longitud. La clase tiene el decorador *@dataclass* que tiene como objetivo generar automáticamente un constructor y varios métodos útiles que implementan la igualdad y la representación de los objetos del tipo.

El decorador *@dataclass* puede tener parámetros. En este caso son: *frozen=True, order=True.* El primero indica que las propiedades de un objeto no se pueden cambiar lo que implica que no se puede asignar un nuevo valor a las mismas. Esto lo hemos escogido así porque queremos un tipo inmutable. El segundo especifica que el tipo tiene un orden natural que está establecido comparando los campos por orden.

El decorador *@dataclass* permite que podamos usar las funciones *astuple, asdict* para convertir los objetos en tuplas o diccionarios. Define automáticamente un constructor *Coordenadas2D(latitud,longitud).*

Los métodos de factoría son importantes para crear objetos. Diseñaremos dos: *of* y *parse*. Estos métodos llevan el decorador *@staticmethod*.

```
@staticmethod
def of(latitud:float,longitud:float) -> Coordenadas2D:
       checkArgument(-90<=latitud and latitud<=90,
              f'latitud {latitud} no es correcta')
       checkArgument(-180<=longitud and longitud<=180,
              f'logitud {longitud} no es correcta')
       return Coordenadas2D(latitud,longitud)

@staticmethod
def parse(text:str) -> Coordenadas2D:
       lat,long = text[1:-1].split(',')
       return Coordenadas2D.of(float(lat),float(long))
```

El método de factoría *of* tiene como responsabilidad adicional comprobar que los valores de las propiedades básicas están dentro del rango especificado.

Los siguiente es implementar la representación de los objetos del tipo que se consigue con el método *__str__*.

```
def __str__(self:Coordenadas2D) -> str:
        return '({0},{1})'.format(self.latitud,self.longitud)
```

El diseño del tipo incluye los métodos *distancia, copy, es_cercana*. La distancia implementa la fórmula de harvesine:

```
def distancia(self:Coordenadas2D,other:Coordenadas2D)-> float:
        radio_tierra = 6373.0
        latitud_a, longitud_a =
             radians(self.latitud), radians(self.longitud)
        latitud_b, longitud_b =
             radians(other.latitud), radians(other.longitud)
        inc_lat  = latitud_b - latitud_a
        inc_long = longitud_b - longitud_a

        a = sin(inc_lat / 2)**2 + cos(latitud_a) * \
             cos(latitud_b) * sin(inc_long / 2)**2
        c = 2 * atan2(sqrt(a), sqrt(1 - a))
        return radio_tierra * c
```

El método *es_cercana* se implementa a partir de la distancia y copy usando el constructor. Este método lleva el decorador @propertie que permite usar el método como propiedad. Es decir sin paréntesis.

```
def es_cercana(self:Coordenadas2D, c:Coordenadas2D,
              d:float) -> bool:
        return self.distancia(c) <= d

@property
def copy(self)->Coordenadas2D:
        return Coordenadas2D(self.latitud,self.longitud)
```

El tipo incluye un método estático adicional: *center*. Permite calcular el centro de una lista de coordenadas.

```
@staticmethod
def center(coordenadas:list[Coordenadas2D]) -> \
              Coordenadas2D:
       latMean = mean(x.latitud for x in coordenadas)
       lngMean = mean(x.longitud for x in coordenadas)
       return Coordenadas2D.of(latMean,lngMean)
```

Coordenadas3D

Representa un punto sobre la superficie de la tierra.

Propiedades: Inmutable

- latitud: float, básica, en grados
- longitud: float, básica, en grados
- altitud: float, básica, en kms,
- coordenada2d: Coordenada2D, derivada
- copy: Coordenada3D, derivada
- distancia(other: Coordenada3D): float, derivada,
- es_cercana(other:Coordenada3D, d:float): bool, derivada

Invariante

- *distancia(other)* = sqrt{coordenada2d.distance(other.coordenada2d)^2-(altitud-other.altitud)^2)
- es_cercana(other,d) = *distance(other) < d*

Representación

- (longitud,latitud,altitud)

Factoría

- parse(text:str)-> Coordenadas3D, en grados, kms
- of(latitud:float,longitude:float,altitud:float) -> Coordenadas3D, grados y km

Como en le caso de Coordenadas2D el tipo se implementa mediante ua clase con el decorador *@dataclass* que lleva los parámetros *frozen=True,order=True.*

```
from __future__ import annotations
from math import sqrt
from us.lsi.coordenadas.Coordenadas2D import Coordenadas2D
from dataclasses import dataclass, asdict, astuple
from us.lsi.tools.Preconditions import checkArgument

@dataclass(frozen=True,order=True)
class Coordenadas3D:
    latitud: float
    longitud: float
    altitud: float
```

Los métodos de factoría:

```
@staticmethod
def of(latitud:float,longitud:float,altitud:float) -> \
             Coordenadas3D:
       checkArgument(-90<=latitud and latitud<=90,
                     f'latitud {latitud} no es correcta')
       checkArgument(-180<=longitud and longitud<=180,
                     f'logitud {longitud} no es correcta')
        return Coordenadas3D(latitud,longitud,altitud)

@staticmethod
def parse(text:str) -> Coordenadas2D:
       lat,long,alt = text[1:-1].split(',')
       return Coordenadas2D.of(
                    float(lat),float(long),float(alt))
```

La representación:

```
def __str__(self:Coordenadas3D) -> str:
        return '({0},{1},{2})' \
             .format(self.latitud,self.longitud,self.altitud)
```

Los métodos distancia y es_cercana:

```
def distancia(self:Coordenadas3D,other:Coordenadas3D) -> \
              float:
       c1 = self.to2D
       c2 = other.to2D
       d_2d = c1.distancia(c2)
       inc_alt = self.altitud-other.altitud
       return sqrt(inc_alt**2 + d_2d**2)

def es_cercana(self:Coordenadas3D, c:Coordenadas3D, d:float) \
              -> bool:
       return self.distancia(c) <= d
```

Y la propiedad *to2D*:

```
@property
def to2D(self:Coordenadas3D) -> Coordenadas2D:
       return Coordenadas2D(self.latitud,self.longitud)
```

Marca

Representa la coodenadas 3D de un punto en la superfice de la tierra en un tiempo concreto

Propiedades: Inmutable

- tiempo: LocalTime, básica
- coordenadas:Coordenadas3D, básica
- latitud: Double, derivada, en grados
- longitud: Double, derivada, en grados
- altitud: Double, derivada, en Km

Representación:

- (00:00:30,2.3,0.5,7.9)

Factoría

- parse(linea:list[str])->Marca
- of(tiempo: LocalTime,latitud:float,longitud: float,altitud:float))
  ->Marca

El tipo Marca se implementa mediante una clase con las propiedades básicas tiempo y coordenadas.

```
from __future__ import annotations
from dataclasses import dataclass
from datetime import time
from us.lsi.tools.Dates import parse_time, str_time
from us.lsi.coordenadas.Coordenadas3D import Coordenadas3D

@dataclass(frozen=True)
class Marca:
    tiempo: time
    coordenadas: Coordenadas3D
```

Las propiedades derivadas:

```
@property
def latitud(self):
      return self.coordenadas.latitud
@property
def longitud(self):
      return self.coordenadas.longitud
@property
def altitud(self):
      return self.coordenadas.altitud
```

El cálculo de la distancia entre marcas

```
def distancia(self,other):
       return self.coordenadas.distancia(other.coordenadas)
```

La representación:

```
def __str__(self):
       return '({0},{1:>20},{2:>20},{3:>20})'
            .format(str_time(self.tiempo,"%H:%M:%S"),
                  self.latitud, self.longitud, self.altitud)
```

Y los métodos de factoría

```
@staticmethod
def of(tiempo:time,latitud:float,longitud:float,altitud:float)
              ->Marca:
       coordenadas = Coordenadas3D.of(latitud,longitud,altitud)
       return Marca(tiempo,coordenadas)
@staticmethod
def parse(linea: list[str]) -> Marca:
       tiempo,latitud,longitud,altitud = linea
       tiempo = parse_time(tiempo,'%H:%M:%S')
       coordenadas = Coordenadas3D.of(float(latitud),
              float(longitud), float(altitud)/1000)
       return Marca(tiempo, coordenadas)
```

Intervalo

Un intervalo es un par de marcas consecutivas

Propiedades: Inmutable

- principio: Marca, básica
- fin: Marca, básica
- desnivel: float, derivada, km
- velocidad: float, derivada, km/hora, *tiempo >0*
- longitud: float, derivada, km
- tiempo: float, derivada, km

Invariante

- principio <= fin

Representación:

- ((00:00:30,2.3,0.5,7.9), (00:00:35,2.4,0.6,8.1))

Factoría

- of(principio: Marca, fin: Marca)->Marca,

La clase que implementa el tipo:

```
from __future__ import annotations
from dataclasses import dataclass
from us.lsi.ruta.Marca import Marca
from us.lsi.tools import Preconditions
from us.lsi.tools.Dates import to_datetime

@dataclass(frozen=True)
class Intervalo:
    principio: Marca
    fin: Marca
```

Algunas propiedades derivadas:

```
@property
def desnivel(self) -> float:
       return self.fin.coordenadas.altitude- \
              self.principio.coordenadas.altitude
@property
def longitud(self) -> float:
       return self.principio.coordenadas.distancia(
              self.fin.coordenadas)
```

El resto de las propiedades derivadas.

```
@property
def tiempo(self) -> float:
       return ((to_datetime(self.fin.tiempo) - \
              to_datetime(self.principio.tiempo)).seconds)/3600
@property
def velocidad(self) -> float:
       Preconditions.checkArgument(self.tiempo > 0, \
              'El tiempo debe ser mayor que cero y es {0:.2f}'\
              .format(self.tiempo))
       return self.longitud/self.tiempo
```

La representación y el método de factoría

```
@staticmethod
def of(principio: Marca, fin:Marca) -> Intervalo:
       checkArgument(principio <= fin,\
                'Principio={0},fin={1}'.format(principio,fin))
       return Intervalo(principio,fin)

def __str__(self):
       return '({0},{1})'.format(self.principio,self.fin)
```

Ruta

Es una secuencia de marcas. La secuencia es abierta. Es decir no se vuelve al punto de partida.

Propiedades: Mutable

- marcas: list[Marca], básica
- n,int, derivada, *n = len(marcas)*
- intervalo(i:int):Intervalo, derivada, *0<=i<n-1*
- tiempo: float, derivada
- longitud: float, derivada
- velocidad: float, derivada
- velocidad_en_intervalo(i:int): float, derivada
- desnivel_en_intervalo(i:int):float, derivada
- desnivel_en_intervalo(i:int):float, derivada
- desnivel_acumulado: tuple[float,float], derivada

Representación:

- (00:00:30,2.3,0.5,7.9) ..., (00:00:35,2.4,0.6,8.1)}

Factoría

- leer_de_fichero(fichero:str)->Ruta

Este es un tipo mutable. Lo implementamos medainte una clase sin el decorador *@dataclass*. Los campos y el constructor se especifican de otra forma:

```
from __future__ import annotations
from us.lsi.ruta.Marca import Marca
from us.lsi.ruta.Intervalo import Intervalo
from us.lsi.tools.File import lineas_de_csv
from us.lsi.tools.Preconditions import checkElementIndex
from us.lsi.tools import Graphics
from us.lsi.tools import Draw
from itertools import accumulate
from us.lsi.tools.Iterable import str_iterable,limit
from typing import Iterable

class Ruta:
      def __init__(self, marcas:list[Marca]):
            self.marcas=marcas
            self.n = len(marcas)
```

Es importante destacar que hemos definido un constructor con el único parámetro marcas. El constructor calcula una propiedad derivada de mucho uso *n*.

Un objeto ruta se lee de un fichero *csv* mediante un metodo de factoría usando la función *lineas_de_csv* y el método parse del tipo Marca.

```
@staticmethod
def ruta_of_file(fichero: str) -> Ruta:
      marcas = [Marca.parse(x) for x in \
            lineas_de_csv(fichero, delimiter =",")]
      return Ruta(marcas)
```

La representación construye una cadena de caracteres a partir de todas las marcas de la ruta

```
def __str__(self):
       return '\n'.join(str(m) for m in self.marcas)
```

Veamos la implementación del resto de propiedades comenzando por la propiedad intervalo:

```
def intervalo(self, i:int) -> Intervalo:
        Preconditions.checkElementIndex(i, self.n-1)
        return Intervalo.of(self.marcas[i],self.marcas[i+1])
```

Hemos de tener en cuenta que un intervalo se define entre una marca y la siguiente. Solamente hay disponibles intervalos que comienzan en las marcas *0..n-2.* Tengamos en cuenta que la última marca tiene índice *i = n-1.*

Las propiedades tiempo, longitud y velocidad media serán:

```
@property
def tiempo(self) -> float:
        return sum(self.intervalo(i).tiempo \
             for i in range(0,self.n-1))
@property
def longitud(self) -> float:
        return sum(self.intervalo(i).longitud \
             for i in range(0,self.n-1))
```

Y la velocidad media

```
@property
def velocidad_media(self) -> float:
        return self.longitud/self.tiempo
```

Los desniveles acumulados creciente y decreciente:

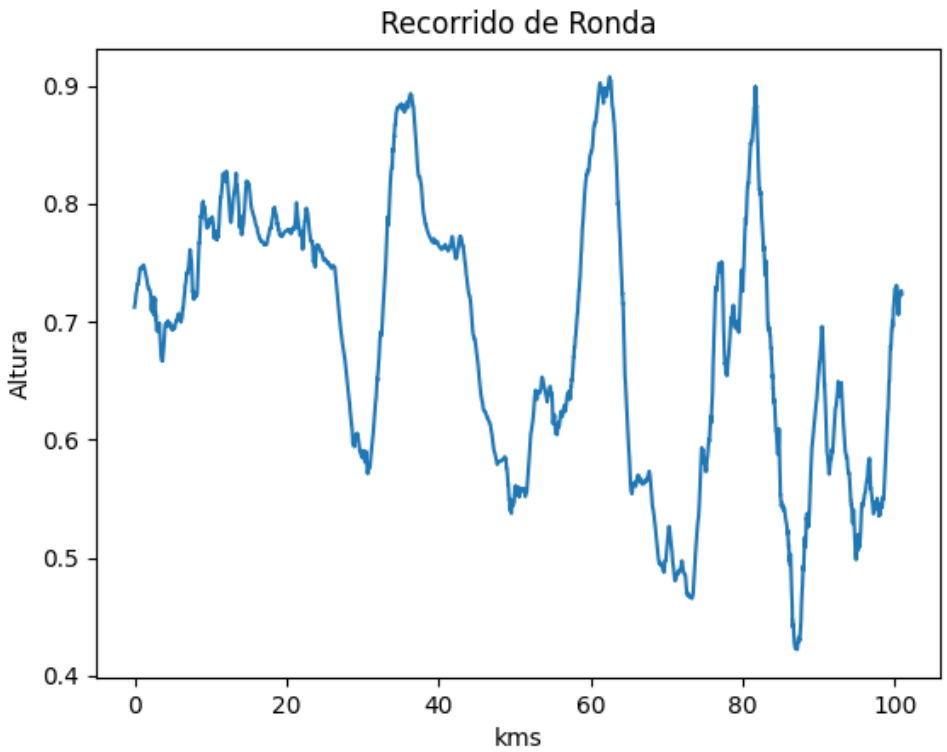
```
@property
def desnivel_creciente_acumulado(self) -> float:
        return sum(self.intervalo(i).longitud \
             for i in range(0,self.n-1) \
             if self.intervalo(i).desnivel > 0)

@property
def desnivel_decreciente_acumulado(self) -> float:
        return sum(self.intervalo(i).longitud
             for i in range(0,self.n-1) \
             if self.intervalo(i).desnivel < 0)
```

La representación gráfica la podemos conseguir usando la función *Draw.draw_multiline* que usa la librería matplotlib de Pyhton

```
def mostrar_altitud(self):
        distancias = list(accumulate( \
              (self.intervalo(i).longitud \
              for i in range(0, self.n-1)),initial=0))
        alturas = [self.marcas[i].coordenadas.altitud \
              for i in range(0,self.n)]
        datos = list(zip(distancias,alturas))
        Draw.draw_multiline(datos,y_label='Altura', \
              x_label='kms', \
              title='Recorrido de Ronda')
```

El resultado es



Una segunda posibilidad es usar la función *Graphics.lineChart* que usa los gráficos de Google

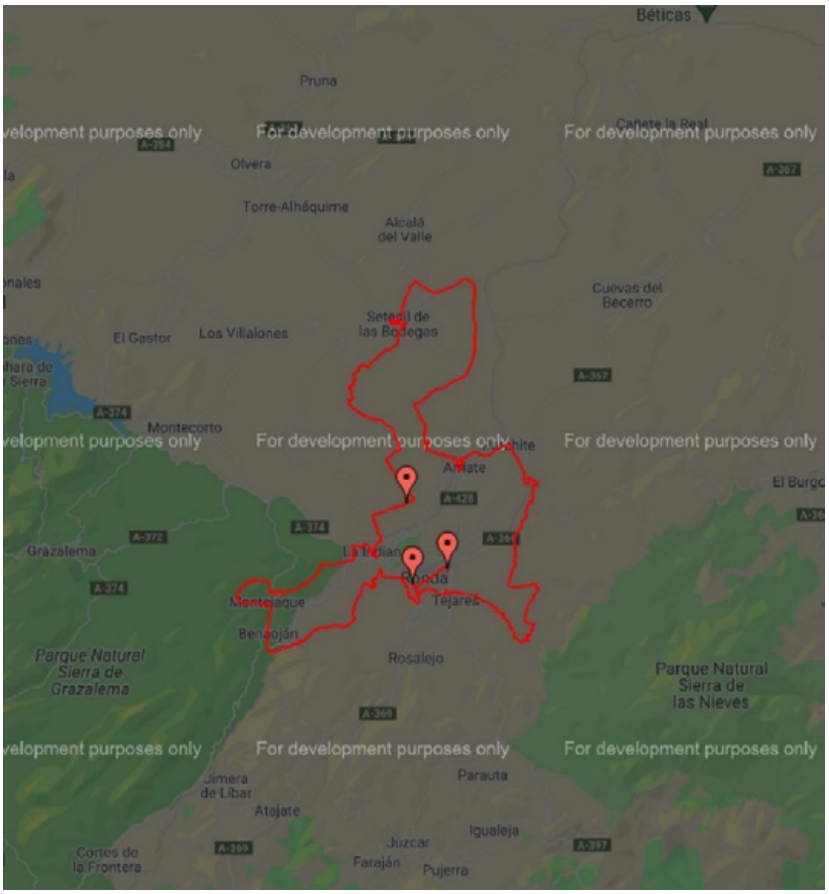
```
def mostrar_altitud_google(self,fileOut):
        distancias = list(accumulate(
              (self.intervalo(i).longitud
                    for i in range(0, self.n-1)),initial=0))
        alturas = [str(self.marcas[i].coordenadas.altitud)
                    for i in range(len(self.marcas))]
        campos = ["Posicion","Altura"]
        Graphics.lineChart(fileOut,"Ruta Ronda",campos,
                    (distancias,alturas))
```

Con un resultado similar.

Otra tarea interesante es representar la ruta en un mapa usando la función *GraphicsMaps.polyline.*

```
def mostrar_mapa_google(self,fileOut:str)->None:
        coordenadas = [c.coordenadas.to2D
              for c in self.marcas]
        polyline(fileOut,coordenadas)
```

Cuyo resultado es



Servicio de bicicletas de sevilla (Sevici)

Se dispone de los datos de las estaciones de la red de Sevici. Los datos se encuentran en un fichero CSV. Cada línea del fichero contiene seis datos:

- Nombre de la estación
- Número total de bornetas de la estación
- Número de bornetas vacías
- Número de bicicletas disponibles
- Latitud
- Longitud

Los datos dependen del instante en que se obtiene el fichero, ya que el número de bicicletas y bornetas libres varía de forma continua. Estas serían, por ejemplo, las primeras líneas del fichero en un momento dado:

```
name,slots,empty_slots,free_bikes,latitude,longitude
149_CALLE ARROYO,20,11,9,37.397829929383,-5.97567172039552
257_TORRES ALBARRACIN,20,16,4,37.38376948792722,-5.908921914235877
243_GLORIETA DEL PRIMERO DE MAYO,15,6,9,37.380439481169994,-5.953481197462
109_AVENIDA SAN FRANCISCO JAVIER,15,1,14,37.37988413609134,-5.974382770011
073_PLAZA SAN AGUSTIN,15,10,4,37.38951386231434,-5.984362789545622
```

Lo primero es llevar a cabo el diseño de tipos.

Estación: Inmutable, con orden natural

Propiedades:

- Numero: int, >=0
- Name: str
- Slots: int; >=0
- Empty_Slots: int, >=0
- Free_Bykes; Integer, >=0
- Ubicacion: Coordenadas2D;
- Nombre_compuesto: str, derivada

Representación: A definir

Métodos de factoría

- Parse: A definir
- Of: A definir

Red: Inmutable, sin orden natural

Propiedades:

- Estaciones: List[Estacion],básica, modificable
- NumeroDeEstaciones: int,derivada
- PorNombreCompuesto:Dict[str,Estacion], derivada
- PorNumero: Dict[int,Estacion], derivada
- EstacionDeNumero(n:int):Estacion
- EstacionDeNombreCompuesto(nombre:str):Estacion
- EstacionesCercanasA(Coordenadas2D c, float distance): list[Estacion],derivada
- EstacionesConBicisDisponibles: Set[Estacion],derivada
- EstacionesConBicisDisponibles(Integer n): Set[Estacion],derivada
- Ubicaciones: Set[Coordenadas2D], derivada
- UbicacionEstacionesDisponibles(int k): derivada
- EstacionMasBicisDisponibles: Estacion,derivada
- EstacionesPorBicisDisponbles: Dict[int,List[Estacion]],derivada
- NumeroDeEstacionesPorBicisDisponibles: Dict[int,int],derivada

Operaciones

- *Add(e:Estacion):None*
- *Remove(e:Estacion):None*

Invarriante:

- Todas las estaciones tienen números diferentes

Representacion: A definir

Métodos de factoría

- Lee_de_fichero: Red,lectura de un fichero

Solo mostramos el código de algunos métodos el resto está en el repositorio. Diseñamos Estación mediante @dataclass y Red mediante una clase. Veamos primero el tipo Estacion.

```
from dataclasses import dataclass
from us.lsi.tools import Preconditions
from us.lsi.coordenadas.Coordenadas2D import Coordenadas2D

@dataclass(frozen=True,order=True)
class Estacion:
    numero: int
    name: str
    slots: int
    empty_slots: int
    free_bikes: int
    ubicacion: Coordenadas2D
```

Con ello hemos diseñado el tipo inmutable Estacion con las propiedades básicas indicadas. Este tipo vien dotado de orden natural. Veamos ahora las propiedades derivadas

```
@property
def nombre_compuesto(self) -> str:
      return '{0}_{1}'.format(self.numero,self.name)
```

Y la representación:

```
def __str__(self):
       return '{0:3d} {1:>35s} {2:2d} {3:2d} {4:2d} \
      {5: >40s}'.format(self.numero,self.name,self.slots,\
      self.empty_slots,self.free_bikes,str(self.ubicacion))
```

La representación está escogida para que podamos ver la lista de estaciones con campos alineados por la derecha o por la izquierda.

Veamos los métodos de factoría. En primer lugar of. Este método lo diseñamos para que compruebe si los datos de la estación son correctos: *slots>=0, empty_slots>=0, free_bikes>=0.*

```
@staticmethod
def of(numero:int,name:str,slots:int,empty_slots:int,\
       free_bikes:int,ubicacion:Coordenadas2D) -> Estacion:
       checkArgument(slots>=0,"Slots deben ser mayor o igual\
que cero y es {0:d}".format(slots))
       checkArgument(empty_slots>=0,"Empty_Slots deben ser\
mayor o igual que cero y es {0:d}".format(empty_slots));
       checkArgument(free_bikes>=0,"Free_Bikes deben ser\ mayor
o igual que cero y es {0:d}".format(free_bikes));
        return Estacion(numero,name, slots,empty_slots,\
       free_bikes,ubicacion)
```

Tras este diseñamos el método de parsing. Asumimos que vamos a leer el fichero con el método *File.lineas_de_csv.* Esto implica que cada línea con los datos de una estación vendrá dada por una lista de cadenas de caracteres. Antes de llamar al método *Estacion.of* debemos convertir cada trozo de texto en entero o float según el tipo del campo.

```
@staticmethod
def parse(linea: list[str]) -> Estacion:
       name,slots,empty_slots,free_bikes,longitude,latitude\
              = linea
       checkArgument('_' in name,'{0} no contiene _'\
              .format(name))
       numero, name = name.split("_")
       numero = int(numero)
       slots = int(slots)
       empty_slots = int(empty_slots)
       free_bikes = int(free_bikes)
       ubicacion = \
              Coordenadas2D(float(longitude),float(latitude))
        return Estacion.of(numero,name,slots,
              empty_slots,free_bikes,ubicacion)
```

Implementemos ahora el tipo Red. Este lo vamos a diseñar como un tipo mutable. Solo propiedad básica será mutable. Las propiedades derivadas no se podrán modificar aunque cambiarán su valor cuando cambie la propiedad básica. Algunas propiedades derivadas es conveniente calcularlas cuando construimos el objeto o cuando modificamos la propiedad básica.

```
from __future__ import annotations
from us.lsi.sevici.Estacion import Estacion
from us.lsi.tools.File import encoding, lineas_de_csv
from us.lsi.coordenadas.Coordenadas2D import Coordenadas2D
from sortedcontainers import SortedSet
from us.lsi.tools.Iterable import grouping_list, \
      str_iterable,frequencies

class Red:
    def __init__(self, e:list[Estacion],\
              por_nombre_compuesto:dict[str,Estacion]=None,\
              por_numero:dict[int,Estacion]=None):
       self._estaciones = e
       self._por_nombre_compuesto:dict[str,Estacion] =
              por_nombre_compuesto
       self._por_numero:dict[int,Estacion] = por_numero
```

Ahora diseñamos las propiedades consultables y las operaciones.

```
@property
def estaciones(self)->list[Estacion]:
       return self._estaciones

@property
def por_nombre_compuesto(self)->dict[str,Estacion]:
       return self._por_nombre_compuesto

@property
def por_numero(self)->dict[int,Estacion]:
       return self._por_numero
```

Las tienen como objetivo modificar las propiedades del objeto. Hemos previsto dos: add y remove. En estas operaciones, además de modificar las propiedades, deben mantener el invariante: no puede haber dos estaciones con el mismo número y actualizar las propiedades derivadas por_numero y por nombre compuesto.

```
def add(self,estacion:Estacion)->None:
        checkArgument(estacion.numero not in self.por_numero,\
       'El numero {} de la estacion esta repetido'\
       .format(estacion.numero))
       checkArgument(estacion.nombre_compuesto \
       not in self.por_nombre_compuesto,
       'El nombre compuesto {} de la estacion esta repetido'\
       .format(estacion.nombre_compuesto))
        self._estaciones.append(estacion)
        pnc = {e.nombre_compuesto:e for e in self._estaciones}
        pn = {e.numero:e for e in self._estaciones}
        self._estaciones.sort()
        self._por_nombre_compuesto = pnc
        self._por_numero = pn
```

Las mismas ideas se aplican en remove

```
def remove(self,estacion:Estacion)->None:
        self._estaciones.remove(estacion)
        pnc = {e.nombre_compuesto:e for e in self._estaciones}
        pn = {e.numero:e for e in self._estaciones}
        self._estaciones.sort()
        self._por_nombre_compuesto = pnc
        self._por_numero = pn
```

La representación la diseñamos para mostrar una estación por línea

```
def __str__(self) -> str:
        return str_iterable(self._estaciones,sep='\n',\
       prefix='Estaciones\n',suffix='\n----------------------')
```

Tenemos dos métodos de factoría: of y parse ambos estáticos

```
@staticmethod
def of(e:list[Estacion],\
       por_nombre_compuesto:dict[str,Estacion]=None,\
       por_numero:dict[int,Estacion]=None) -> Red:
       return Red(e,por_nombre_compuesto,por_numero)
```

Los datos los leemos de un fichero csv que contiene la información de una estación por línea.

```
@staticmethod
def data_of_file(fichero: str) -> Red:
        estaciones = [Estacion.parse(x)\
       for x in lineas_de_csv(fichero,\
       encoding='cp1252')[1:]]
       checkArgument(len(estaciones) == \
       len({e.numero for e in estaciones}),\
       'Hay numeros de estacion repetidos')
       pnc = {e.nombre_compuesto:e for e in estaciones}
       pn = {e.numero:e for e in estaciones}
       estaciones.sort()
       return Red.of(estaciones,pnc,pn)
```

Se ha previsto saltar la primera linea leída del fichero que contiene el encabezado que no vamos a usar. Hemos comprobado que todos los números de estación son diferentes. Se deja como ejercicio encontrar el primero repetido.

Calculemos las estaciones cercanas a una posición geográfica dada

```
def estaciones_cercanas_a(self, c: Coordenadas2D, \
       distancia:float) -> list[Estacion]:
       return sorted(e for e in self.estaciones\
       if e.ubicacion.distancia(c) <= distancia)
```

Estciones con más de k bicicletas disponibles

```
def estaciones_con_bicis_disponibles(self, k:int=1) ->
       set[Estacion]:
      return {e for e in self.estaciones if e.free_bikes >= k}
```

Estación con más bicicletas disponibles

```
@property
def estacion_con_mas_bicis_disponibles(self) -> Estacion:
        return max(self.estaciones,\
              key = lambda e:e.free_bikes)
```

Estaciones agrupadas por número de bicicletas libres.

```
@property
def estaciones_por_bicis_disponibles(self) -> \
       dict[int,list[Estacion]]:
       return grouping_list(self.estaciones, \
              lambda e: e.free_bikes)
```

Número de estaciones por bicicletas disponibles.

```
@property
    def numero_de_estaciones_por_bicis_disponibles(self) -> \
       dict[int,int]:
       return frequencies(self.estaciones,\
       lambda e: e.free_bikes)
```


Cálculos sobre un libro

El objetivo es leer un libro de un fichero y hacer cálculos sobre las palabras que contiene, en que líneas aparecen, ordenarlas por frecuencias, etc. Se usará el fichero *quijote.txt* que está disponible en el repositorio.

Se supone que las palabras están separadas una de otras por separadores que podemos definir. Los separadores que usaremos son *"[,;.\n()?¿!¡:\"]"* aunque podemos añadir o eliminar separadores.

Queremos implementar las siguientes funciones:

- *numeroDeLineas(file:str)->int*
- *numeroDePalabrasDistintasNoHuecas(file:str)->int*
- *palabrasDistintasNoHuecas(file:str)->Set[str]*
- *longitudMediaDeLineas(file:str)-> float*
- *numeroDeLineasVacias(file:str)->int*
- *lineaMasLarga(file:str)->str*
- *primeraLineaConPalabra(file:str,palabra:str)->int*
- *lineaNumero(file:str,n_int)-> str*
- *frecuenciasDePalabras(*file:str*)->Dict[str,int], Frecuencia de cada palabra. Ordenadas por palabras*
- *palabrasPorFrecuencias(file:str)->Dict[int,Set[str]], palabras agrupadas por sus frecuencias de aparición. Ordenadas por frecuencias*

- *lineasDePalabra(file:str)->Dict[str,Set[int]], grupos de líneas donde aparece cada palabra.*

Ahora vamos a resolver el problema como un conjunto de funciones agrupadas en un módulo. Veamos la implementación de esas funciones. En primer lugar obtengamos de un fichero un conjunto de palabras a descartar, palabras huecas. Las importaciones necesarias pueden encontrarse en el repositorio.

```
sep = r'[ ,;.\n():?!\"]'

def palabras_huecas() -> set[str]:
    lns = lineas_de_fichero(\
       "../../../resources/palabras_huecas.txt")
    return {p for p in lns}
```

Un iterable con las palabras relevantes, no huecas, del libro.

```
def palabras_no_huecas(file: str) -> Iterable[str]:
    huecas = palabras_huecas()
    lns = lineas_de_fichero(file,encoding='utf-16')
    pls = flat_map(lns,lambda x: re.split(sep, x))
    palabras = (p for p in pls if p not in huecas \
       if len(p) > 0)
    return palabras
```

Hemos leído el fichero de palabras huecas una vez y mantenido los datos en memoria. Otro con las palabras relevantes dintintas

```
def palabras_no_huecas_distintas(file: str) -> Iterable[str]:
    return distinct(palabras_no_huecas(file))
```

El número de líneas del libro, número de palabras huecas, número de palabras, número de palabras relevantes distintas se pueden obtener fácilemente de los métodos anteriores.

Longitud media de las líneas

```
def longitud_media_de_lineas(file:str) -> float:
    return average(len(ln) for ln in lineas_de_fichero(file))
```

La línea más larga

```
def linea_mas_larga(file:str) -> str:
    return max(lineas_de_fichero(file), key= lambda x:len(x))
```

Número de líneas vacías

```
def numero_de_lineas_vacias(file:str) -> int:
    return count(ln for ln in
       lineas_de_fichero(file,encoding='utf-16')\
       if len(ln) == 0)
```

Línea número n

```
def linea_numero(file:str,n:int) -> str:
    return    find_first(enumerate(lineas_de_fichero(file)),\
       predicate=lambda p:p[0] == n).get()[1]
```

Frecuencias de palabras

```
def frecuencias_de_palabras(file:str) -> OrderedDict[str,int]:
    d = frequencies(palabras_no_huecas(file))
    return OrderedDict(sorted(d.items(), key=lambda t: t[0]))
```

Palabras por frecuencias. El diccionario invertido devuelto por el método anterior.

```
def palabras_por_frecuencias(file:str) -> \
       OrderedDict[int,set[str]]:
    d = invert_dict_set(frecuencias_de_palabras(file))
    return OrderedDict(sorted(d.items(), key=lambda t: t[0]))
```

Líneas en las que aparecen las palabras

```
def palabra_en_lineas(file:str) -> OrderedDict[str,set[int]]:
    lns = lineas_de_fichero(file,encoding='utf-16')
    palabras = ((i,p) for i,linea in enumerate(lns) \
       for p in re.split(sep,linea) if len(p) >0)
    d = grouping_set(palabras,fkey=lambda e: e[1], \
       fvalue=lambda e: e[0])
    return OrderedDict(sorted(d.items(), key=lambda t: t[0]))
```

Hemos de destacar la estructura para definir el iterable de pares número de línea palabra. Usamos la estructura

```
((i,p) for i,linea in enumerate(lns) \
       for p in re.split(sep,linea) if len(p) >0)
```

Convertimos las líneas en pares numero-linea y cada línea la dividimos en palabras. Finalmente filtramos las palabras no vacías. Luego agrupamos las lineas en las que aparece una palabra y finalmente construimos un diccionario ordenado por las palabras.

Las k palabras más frecuentes

```
def palabras_frecuentes(file:str, k:int)->str:
    return Counter(palabras_no_huecas(file)).most_common(k)
```

WhatsUp

En este ejercicio vamos a analizar los mensajes de un grupo de Whatsapp. En concreto, podremos obtener toda esta información:

- Número de mensajes a lo largo del tiempo
- Número de mensajes según el día de la semana, o la hora del día
- Número de mensajes escritos por cada participante en el grupo
- Palabras más utilizadas en el grupo
- Palabras más características de cada participante en el grupo

Para llevar a cabo el ejercicio, necesitamos un archivo de log de un grupo de Whatsapp. Un fichero de log ficticio que podemos utilizar para desarrollar los ejercicios (generado a partir de los diálogos de la primera temporada de The Big Bang Theory).

El el formato del fichero que genera Whatsapp varía según se trate de la versión Android o iOS. Veamos un ejemplo de cada caso:

- *Android:*

```
26/02/16, 09:16 - Leonard: De acuerdo, ¿cuál es tu punto?
26/02/16, 16:16 - Sheldon: No tiene sentido, solo creo que es
      una buena idea para una camiseta.
```

- *iOS:*

```
[26/2/16 09:16:25] Leonard: De acuerdo, ¿cuál es tu punto?
[26/2/16 16:16:54] Sheldon: No tiene sentido, solo creo   que
      es una buena idea para una camiseta.
```

Como se puede observar, cada mensaje del chat viene precedido por la fecha, la hora, y el nombre del usuario. El formato no es CSV, como en otros ejercicios, sino que es, digamos, más libre. Así, la fecha ocupa el principio de una línea, que puede venir seguida de una coma. Tras un espacio en blanco, viene la hora (que puede contener o no los segundos), seguida de un espacio, un guión y otro espacio, o bien de dos puntos y un espacio, según la versión de Whatsapp. Por último, tenemos el nombre del usuario, acabando en dos puntos, y tras otro espacio, aparece el texto del mensaje.

Tipos

PalabraUsuario:

Propiedades

- Palabra:str,básica
- Usuario: str,básica

Mensaje:

Propiedades:

- Fecha:LocalDate,básica
- Hora: LocalTime,básica
- Usuario: str,básica
- Texto: str,básica

Representación: en el formato Android

Factoría:

- Parse dado un texto con todas las partes del mensaje

Conversacion:

Propiedades:

- Mensajes: list[Mensaje],básica

- PalabrasHuecas:set[str] ,básica
- MensajesPorPropiedad(Callable[[Mensaje],P] f):
  dict[P,list[Mensaje]], derivada
- NumeroMensajesPorPropiedad(Callable[[Mensaje],P] f):
  dict[P,int], derivada
- NumeroPalabrasUsuario: dict[str,int], derivada, NPU
- FrecuenciaPalabrasUsuario: dict[PalabraUsuario,int], derivada, FPU
- NumeroPalabrasResto: dict[str,int], derivada, NPR
- FrecuenciaPalabrasResto: dict[PalabraUsuario,int], derivada, FPR
- ImportanciaPalabra(usuario:str, umbral:int): Double. IP
- PalabrasCaracteristicasUsuario(usuario:str, umbral:int):
  dict[str,float], PCU

Notas

$$IP(u,p) = \frac{FPU(u,p) * NPR(u)}{NPU(u) * FPR(u,p)}$$

Asumimos que $FPR(u,p)$ tiene un valor mínimo, caso de que no use p, de un valor dado por ejemplo 0.00001

PalabrasCaracteristicasUsuario(usuario,umbral): Un diccionario con la importancia de las palabras para el usuario que tengan frecuencia superior a umbral

Expresiones regulares

De cada línea del fichero habrá que extraer los distintos campos de información (fecha, hora, usuario y texto de cada mensaje). La mejor manera de extraer esta información es mediante el uso de *expresiones regulares*. Estas expresiones nos permiten definir patrones en los que pueden encajar distintos trozos de texto, y extraer información a partir de distintos trozos de la cadena que ha sido reconocida. Cada lenguaje de programación tiene pequeñas diferencias para tratar las expresiones

regulares. En el caso el patrón que describe la estructura de los mensajes es:

```
String RE =
"(?<fecha>\\d\\d?/\\d\\d?/\\d\\d?) ,? (?<hora>\\d\\d?:\\d\\d)
- (?<usuario>[^:]+): (?<texto>.+)";
```

- Cada uno de los tramos encerrados entre paréntesis, y con ? seguido de un nombre, se corresponde con un dato que vamos a extraer. Hay cuatro tramos.
- El primer tramo de la cadena, *(?<fecha>\\d\\d?/\\d\\d?/\\d\\d?),* es un patrón que encaja con las fechas que aparecen en los mensajes. Los \\d indican dígitos, y las interrogaciones indican partes que pueden aparecer o no.
- El segundo tramo, *(?<hora>\\d\\d?:\\d\\d),* encaja con las horas.
- El tercer tramo, *(?<usuario>[^:]+)* , reconoce los nombres de usuario. La expresión *[^:]+* significa "cualquier secuencia de caracteres hasta dos puntos excluidos" (es decir, que reconoce todo el trozo de texto que viene antes de los dos puntos).
- Y el último tramo, *(?<texto>.+),* captura el texto de los mensajes.

Diseñemos los tipos necesarios. En primer lugar el tipo Mensaje y el par UsuarioPalabra. El tipo UsuarioPalabra podemos implementarlo como una tupla, una tupla con nombre o una clase. Los hacemos de la última forma:

```
@dataclass(frozen=True,order=True,)
class UsuarioPalabra:
    usuario:str
    palabra:str

    @staticmethod
    def of(usuario: str, palabra:str) -> UsuarioPalabra:
        return UsuarioPalabra(usuario,palabra)

    @staticmethod
    def of_tuple(t: tuple[str,str]) -> UsuarioPalabra:
        return UsuarioPalabra(t[0],t[1])

    def __str__(self):
        return "(%s,%s)" % (self.usuario,self.palabra)
```

Ahora el tipo mensaje.

```
RE = r'(?P<fecha>\d\d?/\d\d?/\d\d?) (?P<hora>\d\d?:\d\d) -\
(?P<usuario>[^:]+): (?P<texto>.+)'

@dataclass(frozen=True,order=True)
class Mensaje:
    fecha:date
    hora: time
    usuario: str
    texto: str
```

Aquí declaramos la expresión regular que especifica las partes en un mensaje tal como se ha descrito arriba. Con ello podemos diseñar el parsing

```
@staticmethod
def parse(mensaje: str) -> Mensaje:
       matches = re.match(RE,mensaje)
       if(matches):
            fecha = parse_date(matches.group('fecha'),\
              '%d/%m/%y')
            hora = parse_time(matches.group('hora'), '%H:%M')
            usuario = matches.group('usuario')
            texto = matches.group('texto')
            return Mensaje(fecha, hora, usuario, texto)
        else:
            return None
```

La expresión regular nos ha permitido dividir el mensaje en trozos. Posteriormente convertimos cada trozo al tipo adecuado.

Si el mensaje no se corresponde con el patrón devolvemos *None*.

Finalmente la representación es de la forma:

```
def __str__(self):
        return "%s %s - %10s:\n  %s" %\
(str_date(self.fecha,'%d/%m/%y'),str_time(self.hora,'%H:%M'),\
self.usuario,self.texto)
```

Hemos usado otra forma de construir cadenas de caracteres con el operador % que tiene una filosofía similar al método format pero ahora los elementos a sustituir en la cadena de formato están especificados por % en vez de {}.

El último tipo es l Conversacion.

```
week_days = ("Monday","Tuesday","Wednesday","Thursday",\
"Friday","Saturday","Sunday")
sep = r'[ ,;.\n():?!\"]'

class Conversacion:
    def __init__(self, mensajes: List[Mensaje], \
            palabras_huecas: Set[str]):
        self._mensajes: List[Mensaje] = mensajes
        self._palabras_huecas: Set[str] = palabras_huecas
        self._usuarios: Set[str] = \
            {m.usuario for m in self._mensajes}
        self._mensajes_por_usuario: Dict[str,Mensaje] = None
        self._numero_de_mensajes_por_usuario = None
        self._frecuencia_de_palabras: Dict[str,int] = None
        self._numero_de_palabras: int = None
        self._frecuencia_de_palabras_por_usuario: \
            Dict[UsuarioPalabra,int] = None
        self._numero_de_palabras_por_usuario: Dict[str,int]= \
            None
        self._frecuencia_de_palabras_por_resto_de_usuarios: \
            Dict[UsuarioPalabra,int] = None
        self._numero_de_palabras_por_resto_de_usuarios:  \
            Dict[str,int] = None
```

Diseñamos un tipo donde incluimos como atributos privados toda las básicas y derivadas. Las derivadas las inicializamos a None y cuando hagamos el primer cálculo actualizaremos ese atributo. Esta técnica nos permite no repetir cálculos que podrían ser costosos. Veamos los métodos de factoría.

```
@staticmethod
def data_of_file(file: str) -> Conversacion:
      ms = (Mensaje.parse(m)\
            for m in lineas_de_fichero(file))
      mensajes = [m for m in ms if m is not None]
      palabrasHuecas = {p for p in \
      lineas_de_fichero("palabras_huecas.txt") if len(p) >0}
      return Conversacion(mensajes,palabrasHuecas)
```

Como vemos se recorren las líneas del fichero y a partir e cada una de ellas construimos un mensaje.

La representación:

```
def __str__(self) -> str:
        it1 = str_iterable(self.palabras_huecas)
        it2 = str_iterable(self.mensajes,separator='\n',\
              prefix='',suffix='')
        return 'Palabras huecas =\n{0:s}\nMensajes = \n{1:s}'\
       .format(it1, it2)
```

Algunos los atributos los usamos para implementar propiedades

```
@property
def mensajes(self) -> List[Mensaje]:
       return self._mensajes

@property
def palabras_huecas(self) -> Set[str]:
       return self._palabras_huecas

@property
def usuarios(self) -> Set[str]:
       return self._usuarios
```

Este diseño hace posible que las propiedades *mensajes* y *palabras_huecas* puedan ser consultadas pero no modificadas.

Para implementar el resto de las propiedades diseñamos un para de métodos privados que usaremos posteriormente. Ambos toman una funciín que a partir de un mensaje calcula una propiedad. El primero agrupa los mensajes según la propiedad calculada. El segundo el número de mensajes con esa propiedad.

```
def _mensajes_por_propiedad(self, p: Callable[[Mensaje],P])->\
              Dict[P,List[Mensaje]]:
        return grouping_list(self.mensajes,fkey=p)

def _numero_de_mensajes_por_propiedad(self,\
              p:Callable[[Mensaje],P])-> Dict[P,int]:
        return frequencies(self.mensajes,fkey=p)
```

Lo métodos anteriores podemos usarlos para implementar algunas de las propiedades pedidas. Veamos primero los mensajes por usuario. Pensamos que esta propiedad va a ser consultada muy a menudo por lo tanto usamos la técnica explicada previamente de guardar el cálculo una

vez realizado en un atributo privado y reutilizar el cálculo las veces siguientes. Por otra parte reutilizamos el método privado *_mensajes_por_propiedad.*

```
@property
def mensajes_por_usuario(self) -> Dict[str,Mensaje]:
      if self._mensajes_por_usuario is None:
           self._mensajes_por_usuario = \
            self._mensajes_por_propiedad(lambda m: m.usuario)
      return self._mensajes_por_usuario
```

Otras propiedades que pensamos no se llamarán muy frecuentemente podemos diseñarlas de forma similar.

```
@property
def mensajes_por_dia_de_semana(self) -> Dict[str,Mensaje]:
       return   self._mensajes_por_propiedad(\
            lambda m: m.fecha.weekday())
```

```
@property
def mensajes_por_fecha(self) -> Dict[date,Mensaje]:
       return self._mensajes_por_propiedad(lambda m: m.fecha)
```

```
@property
def mensajes_por_hora(self) -> Dict[int,Mensaje]:
       return self._mensajes_por_propiedad(\
            lambda m: m.hora.hour)
```

El número de mensajes por propiedad concreta podemos implementarlo reutilizando el método privado diseñado previamente.

```
@property
def numero_de_mensajes_por_usuario(self) -> Dict[str,int]:
       if self._numero_de_mensajes_por_usuario is None:
           self._numero_de_mensajes_por_usuario = \
            self._numero_de_mensajes_por_propiedad(\
            lambda m: m.usuario)
       return self._numero_de_mensajes_por_usuario
```

```
@property
def numero_de_mensajes_por_dia_de_semana(self) ->\
       Dict[str,int]:
         return self._numero_de_mensajes_por_propiedad(\
               lambda m: m.fecha.weekday())
```

```
@property
def numero_de_mensajes_por_fecha(self) -> Dict[date,int]:
         return self._numero_de_mensajes_por_propiedad(\
               lambda m: m.fecha)
```

```
@property
def numero_de_mensajes_por_hora(self) -> Dict[int,int]:
         return self._numero_de_mensajes_por_propiedad(
               lambda m: m.hora.hour)
```

La frecuencia de palabras podemos implementar usando las funciones *flat_map y frequencies*.

```
@property
def frecuencia_de_palabras(self) -> Dict[str,int]:
       if self._frecuencia_de_palabras is None:
            ms_tex = (m.texto for m in self.mensajes)
            ps = flat_map(ms_tex,lambda x: re.split(sep, x))
            palabras = (p for p in ps
              if len(p) > 0 and p not in self.palabras_huecas)
            self._frecuencia_de_palabras =\
                    frequencies(palabras,fkey=identity)
         return self._frecuencia_de_palabras
```

El número de palabras podemos implementarlo a partir de la propiedad anterior.

```
@property
def numero_de_palabras(self) -> int:
        if self._numero_de_palabras is None:
            self._numero_de_palabras = sum(n for _,n in \
                    self.frecuencia_de_palabras.items())
        return self._numero_de_palabras
```

El cálculo del *frecuencia_de_palabras_por_usuario* es más complejo. Requiere definir varios iterables por comprensión y usarlos de forma secuencial.

```
@property
def frecuencia_de_palabras_por_usuario(self) ->
              Dict[UsuarioPalabra,int]:
        if self._frecuencia_de_palabras_por_usuario is None:
            ms_us_tex = ((m.usuario,m.texto) \
                    for m in self.mensajes)
            plsu = (UsuarioPalabra.of(u,p) for u,t in \
                    ms_us_tex for p in re.split(sep,t))
            plsuf = (pu for pu in plsu if len(pu.palabra) > 0\
                   and pu.palabra not in self.palabras_huecas)
            self._frecuencia_de_palabras_por_usuario = \
                    frequencies(plsuf)
        return self._frecuencia_de_palabras_por_usuario
```

El número de palabras por ususario se driva de la propiedad anterior.

```
@property
def numero_de_palabras_por_usuario(self) -> Dict[str,int]:
        return frequencies(\
             self.frecuencia_de_palabras_por_usuario.items(),\
             fkey=lambda e:e[0].usuario)
```

El cálculo de la frecuencia_de_palabras_por_resto_de_usuarios usa varios for anidados par mayor claridad del código.

```
@property
def frecuencia_de_palabras_por_resto_de_usuarios(self) ->
             Dict[UsuarioPalabra,int]:
       if self._frecuencia_de_palabras_por_resto_de_usuarios\
                    is None:
             fpal = ((up.usuario,up.palabra,f) for up,f in \
             self.frecuencia_de_palabras_por_usuario.items())
             d = {}
             for u,p,f in fpal:
                for x in self.usuarios:
                    if x != u:
                        up = UsuarioPalabra.of(x,p)
                        d[up] = d.get(up,0) + f
          self._frecuencia_de_palabras_por_resto_de_usuarios\
            = d
        return \
           self._frecuencia_de_palabras_por_resto_de_usuarios
```

El número de palabras por resto de usuarios se deriva de la propiedad anterior.

```
@property
def numero_de_palabras_por_resto_de_usuarios(self) -> \
             Dict[str,int]:
       return frequencies(\
   self.frecuencia_de_palabras_por_resto_de_usuarios.items(),\
       fkey=lambda e:e[0].usuario)
```

Con las propiedades anteriores podemos abordar el cálculo de la importancia de las palabras

```
def importancia_de_palabra(self,usuario:str,palabra:str) ->\
             float:
        return (self.frecuencia_de_palabras_por_usuario\
             [UsuarioPalabra.of(usuario,palabra)] / \
             self.numero_de_palabras_por_usuario[usuario]) * \
     (self.numero_de_palabras_por_resto_de_usuarios[usuario] \
      /self.frecuencia_de_palabras_por_resto_de_usuarios\
             [UsuarioPalabra.of(usuario,palabra)])
```

Y las palabras características de cada usuario

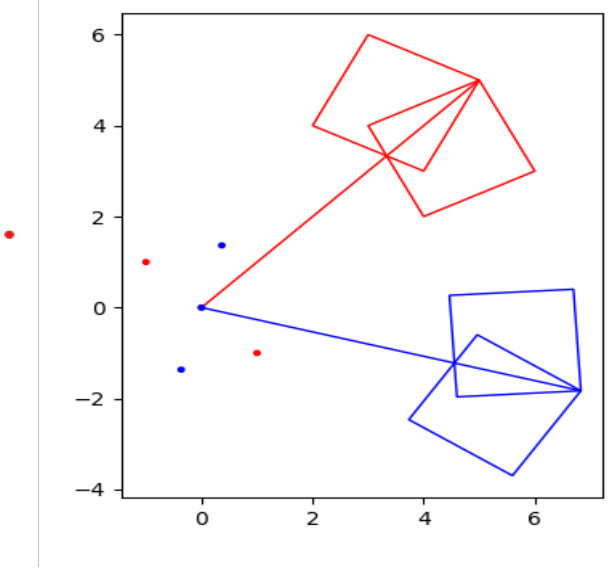
```
def palabras_caracteristicas_de_usuario(\
             self,usuario:str,umbral:int) -> Dict[str,float]:
        r1 = (e for e in \
             self.frecuencia_de_palabras_por_usuario.items()\
             if e[0].usuario == usuario)
        r2 = (e for e in r1 if \
   self.frecuencia_de_palabras.get(e[0].palabra,0) > umbral)
        r3 = (e for e in r2 if e[1] > umbral)
        r4 = (e for e in r3 if \
      self.frecuencia_de_palabras_por_resto_de_usuarios \
      .get(e[0],0) > umbral)
        r5 = ((e[0].palabra,self.importancia_de_palabra(e[0]\
             .usuario,e[0].palabra)) for e in r4)
        return {e[0]:e[1] for e in r5}
```

Con las propiedades anteriores podemos construir un diagrama de barras o de tarta para distintas propiedades. Por ejemplo:

```
def diagrama_de_tarta_mensajes_por_dia_de_semana(self,
             file_out:str) -> None:
        ls = [x for x in \
      self.numero_de_mensajes_por_dia_de_semana.items()]
        ls.sort(key=lambda e:e[0])
        nombres_columna = [week_days[x[0]] for x in ls]
        datos =  [x[1] for x in ls]
        nombres_datos = ['DiaDeSemana','NumeroDeMensajes']
        Graphics.pieChart(file_out,"MensajesPorDiaDeSemana",\
             nombres_datos,nombres_columna, datos)
```

Objetos geométricos

En este ejercicio nos proponemos implementar un conjunto de objetos geométricos: *Punto2D, Segmento2D, Circulo2D, Poligono2D*. Todos ellos comparten propiedades que estarán representadas en objetos del tipo *Objeto2D.* A su vez un *Agregado2D* será una lista de los objetos geométricos anteriores. Todos ellos podrán ser sometidos a algunas operaciones geométricas y representados gráficamente. Un posible gráfico obtenido de esta manera sería:



En el gráfico aparecen puntos, segmentos, cuadrados y sus objetos rotados. Junto a los anteriores vamos a diseñar otros tipos de objetos que aparecen en la geometria del plano como *Vector2D* y *Recta2D*.

Vector2D

Empecemos creando una clase para el tipo Vector2D.

```
from __future__ import annotations
from math import sin, cos, radians, atan2, degrees, sqrt, acos
from dataclasses import dataclass
from us.lsi.tools import Preconditions

@dataclass(frozen=True,order=True)
class Vector2D:
    x: float
    y: float
```

El diseño del tipo *Vector2D* lo suponemos conocido. En la implementación hemos decidido mantener como atributos las propiedades básicas *x, y*. Las propiedades modulo y angulo las consideramos derivadas y no las mantenemos como atributos. Esto implica que si queremos construir un vector dado su módulo y su ángulo debemos calcular primero su *x, y*. Las propiedades x, y son los mismos atributos que hemos declarado inmutables.

Y diversos métodos de factoría

```
@staticmethod
def of(x:float,y:float) -> Vector2D:
      return Vector2D(x,y)
```

```
@staticmethod
def parse(txt:str) -> Vector2D:
      txt = txt[1:-1]
      x,y = txt.split(',')
      return Vector2D(float(x),float(y))
```

```
@staticmethod
def of_grados(modulo:float,angulo:float) -> Vector2D:
       Preconditions.checkArgument(modulo > 0, \
            'El modulo debe ser mayor o igual a cero y es\
{0:.2f}'.format(modulo))
       return Vector2D.of_radianes(modulo,radians(angulo))
```

```
@staticmethod
def of_radianes(modulo:float, angulo:float)-> Vector2D:
        Preconditions.checkArgument(modulo >= 0,
             'El modulo debe ser mayor o igual a cero y es\
{0:.2f}'.format(modulo))
        return Vector2D.of(modulo*cos(angulo)\
             ,modulo*sin(angulo))
```

Ahora diseñamos las propiedades derivadas

```
@property
def modulo(self) -> float:
      return sqrt(self.x*self.x+self.y*self.y)
```

```
@property
def angulo(self) -> float:
      return atan2(self.y,self.x)
```

Algunas propiedades derivadas más: copy, ortogonal y unitario.

```
@property
def copy(self) -> Vector2D:
      return Vector2D(self.x,self.y)

@property
def ortogonal(self)-> Vector2D:
      return Vector2D.of_xy(-self.y,self.x)

@property
def unitario(self)-> Vector2D:
      return Vector2D.of_radianes(1.,self.angulo)
```

La representación podría ser limitando los decimales a dos:

```
def __str__(self)->str:
      return '({0:.2f},{1:.2f})'.format(self.x,self.y)
```

Ahora diseñamos la operaciones suma, resta, producto por escalar y opuesto con los métodos correspondientes para usar los operadores *+, -, /,-*

```
def __add__(self,v:Vector2D)-> Vector2D:
      return Vector2D.of(self.x+v.x,self.y+v.y)
```

```
def __sub__(self,v:Vector2D)-> Vector2D:
      return Vector2D.of(self.x-v.x,self.y-v.y)
```

```
def __mul__(self,factor:float)-> Vector2D:
      return Vector2D.of(self.x*factor,self.y*factor)
```

```
def __neg__(self)-> Vector2D:
      return Vector2D.of(-self.x, -self.y)
```

Y por último el producto escalar, el vectorial, el ángulo con otro vector y la proyección sobre otro vector:

```
def multiply_escalar(self,v:Vector2D) -> float:
        return self.x*v.x+self.y*v.y
def multiply_vectorial_2d(self,v:Vector2D) -> float:
        return self.x*v.y-self.y*v.x
```

```
def angulo_con(self,v:Vector2D) -> float:
        return acos(self.multiply_escalar(v) \
             / (self.modulo * v.modulo))
def proyecta_sobre(self,v:Vector2D)->Vector2D:
        vu = v.unitario
        f = self.multiply_escalar(vu)
        return vu*f
```

Que podemos usar como

```
v0 = Vector2D.parse('(-23.4,67')
print(v0)
v = Vector2D.of(1.,1.)
print(v)
print(v.modulo)
print(degrees(v.angulo))
print(v.ortogonal)
print(degrees(v.angulo_con(v.ortogonal)))
v2 = Vector2D.of(1.,0.)
v3 = Vector2D.of(0.,1.)
print(v2).multiply_vectorial_2d(v3))
```

Objeto2D

Los siguientes elementos que queremos diseñar comparten un conjunto de métodos y propidades comunes. Estos son:

- Rotar un angulo con respecto a un punto
- Trasladar según un vector
- Homotecia según un punto y una razón
- Proyectar sobre una recta
- Obtener el Patch de un objeto geométrico. Donde Patch es un tipo adecuado para la representación gráfica del objeto

Todos estos métodos comunes los incluimos en una clase Objeto2D. Esta clase tendrá métodos sin funcionalidad que serán concretados en las clases hijas. Estos métodos tendrán el decorador *@abstractmethod* y sus cuerpo *pass* al no tener funcionalidad. Una clase que tiene métodos abstractos es una clase abstracta aunque puede que tenga algunos métodos con una funcionalidad. En Python este tipo de clase heredan de una clase especial llamada ABC. Esta clase proporciona algunos servicios que no veremos aquí.

Definida esta clase podemos ya tener, como más adelante veremos, agregados de objetos geométricos cada uno de los cuales implementará de una forma concreta cada uno de los métodos anteriores. La clase será de la forma.

```
from __future__ import annotations
from typing import TypeVar
from abc import ABC,abstractmethod

Punto2D = TypeVar('Punto2D')
Vector2D = TypeVar('Vector2D')
Recta2D = TypeVar('Recta2D')

class Objeto2D(ABC):
    @abstractmethod
    def rota(self, p:Punto2D, angulo:float) -> Objeto2D:
        pass
    @abstractmethod
    def traslada(self, v:Vector2D) -> Objeto2D:
        pass
    @abstractmethod
    def homotecia(self, p:Punto2D, factor:float) -> Objeto2D:
        pass
    @abstractmethod
    def proyecta_sobre_recta(self, r:Recta2D) -> Objeto2D:
        pass
    @abstractmethod
    def simetrico_con_respecto_a_recta(self, r:Recta2D) -> \
            Objeto2D:
        pass
    @abstractmethod
    def shape(self) -> Patch:
        pass
```


Recta2D

Este no es un tipo que hereda de Objeto2D pero vamos a diseñarlo primero paa poder concretar el resto de los tipos.

Una recta viene especificada por un punto, el tipo Punto2D que diseñaremos más tarde, y un vector de tipo Vector2D.

```
from __future__ import annotations
from us.lsi.geometria.Vector2D import Vector2D
from us.lsi.geometria.Punto2D import Punto2D
from dataclasses import dataclass

@dataclass(frozen=True,order=True)
class Recta2D:
    punto: Punto2D
    vector: Vector2D
```

Podemos construirla mediante un punto y un vector o por dos puntos y para ello diseñamos los correspondientes métodos de factoría:

```
@staticmethod
def of(p:Punto2D,v:Vector2D) -> Recta2D:
      return Recta2D(p,v)

@staticmethod
def of_puntos(p1:Punto2D,p2:Punto2D) -> Recta2D:
      return Recta2D(p1,p2.vector_to(p1))
```

Algunos métodos que podemos aplicar a una recta: encontrar un punto de la recta dado un número real, encontrar una paralela por un punto y una perpendicular por un punto:

```
def punto_en_recta(self,factor:float = 0.) -> Punto2D:
        return self.punto + self.vector * factor

def paralela(self,p:Punto2D) -> Recta2D:
        return Recta2D.of(p,self.vector)

def ortogonal(self,p:Punto2D) -> Recta2D:
        return Recta2D.of(p,self.vector.ortogonal)
```

Punto2D

Veamos la implementación de un punto en el plano con dos propiedades básicas: x,y. Hemos de tener en cuenta que es un objeto geométrico que heredará de Objeto2D.

```
from __future__ import annotations
from math import sqrt, pi
from dataclasses import dataclass
from us.lsi.geometria.Cuadrante import Cuadrante
from us.lsi.geometria.Vector2D import Vector2D
from us.lsi.geometria.Objeto2D import Objeto2D
from us.lsi.tools import Draw

@dataclass(frozen=True,order=True)
class Punto2D(Objeto2D):
    x: float
    y: float
```

La forma de indicar que un tipo hereda de otro es indicando el tipo del que se hereda entre paréntesis tras el nombre de la clase.

Varios métodos de factoría:

```
@staticmethod
def origen() -> Punto2D:
      return Punto2D(0.,0.)

@staticmethod
def of(x:float,y:float) -> Punto2D:
      return Punto2D(x,y)

@staticmethod
def parse(linea:str) -> Punto2D:
      linea = linea[1:-1]
      x,y = linea.split(',')
      return Punto2D(float(x),float(y))
```

La representación:

```
def __str__(self) -> str:
      return '({0:.2f},{1:.2f})'.format(self.x,self.y)
```

Algunas propiedades: copia de un punto, conversión a vector.

```
@property
def copy(self: Punto2D) ->  Punto2D:
      return Punto2D(self.x,self.y)
@property
def vector(self: Punto2D) ->  Vector2D:
      return Vector2D(self.x,self.y)
```

La distancia a otro punto se determina por el teorema de pitágoras:

```
def distancia_a(self,p:Punto2D) -> float:
      dx = self.x-p.x;
      dy = self.y-p.y
      return sqrt(dx*dx+dy*dy)

@property
def distancia_al_origen(self: Punto2D) -> float:
      return self.distancia_a(Punto2D.origen())
```

Para el cálculo del cuadrante usamos la sentenica match

```
@property
def cuadrante(self) -> Cuadrante:
      match self:
            case Punto2D(x,y) if x >=0 and y >= 0 :
                  return Cuadrante.PRIMERO
            case Punto2D(x,y) if x <=0 and y >=0 :
                  return Cuadrante.SEGUNDO
            case Punto2D(x,y) if x <=0 and y <=0 :
                  return Cuadrante.TERCERO
            case _ :
                  return Cuadrante.CUARTO
```

El vector definido por dos puntos

```
def vector_to(self,p:Punto2D) -> Vector2D:
      return Vector2D.of(self.x-p.x,self.y-p.y)
```

Los puntos resultants de sumar o restar un vector a otro punto. Lo nombres de los métodos son los adecuados para poder usar luego los operadros +, -.

```
def __add__(self,v:Vector2D) -> Punto2D:
      return Punto2D.of(self.x+v.x,self.y+v.y)

def __sub__(self,v:Vector2D) -> Punto2D:
      return Punto2D.of(self.x-v.x,self.y-v.y)
```

Ahora no falta diseñar los métodos heredados de Objeto2D para darles la funcionalidad concreta para este tipo. En primer lugar *traslada*.

```
def traslada(self,v:Vector2D) ->Punto2D:
      return self.__add__(v)
```

Como vemos simplemente hemo sumado un vector al punto. La operación rotar un ángulo un punto con respecto a otro se implementa fácilemente usando el correspondiente método de los vectores.

```
def rota(self, p:Punto2D, angulo:float) -> Punto2D:
      v = self.vector_to(p).rota(angulo)
      return p + v
```

La homotecia de un factor dado con respecto a un punto la podemos implementar multiplicando el vector que se forma por el factor de escala.

```
def homotecia(self,p:Punto2D,factor:float) -> Punto2D:
      return p + p.vector_to(self) * factor
```

La proyección de un punto sobre una recta lo podemos conseguir escogiendo un punto *p* en la recta (r.punto) y sumándole la proyección sobre el vector de la recta del vector formado por el punto y p.

```
def proyecta_sobre_recta(self,r:'Recta2D') -> Punto2D:
      v1 = r.punto.vector_to(self).proyecta_sobre(r.vector)
      return r.punto + v1
```

El simétrico con respecto a una recta se consigue de forma similar.

```
def simetrico_con_respecto_a_recta(self,r:'Recta2D') -> \
              Punto2D:
      p = self.proyecta_sobre_recta(r)
      return self + self.vector_to(p) * 2.
```

Segmento2D

Un segmento es un objeto geométrico, hereda por tanto de Objeto2D, definido por dos puntos. Su implementación mediante una clase es:

```
from __future__ import annotations
from dataclasses import dataclass
from us.lsi.geometria.Punto2D import Punto2D
from us.lsi.geometria.Vector2D import Vector2D
from us.lsi.geometria.Recta2D import Recta2D
from us.lsi.geometria.Objeto2D import Objeto2D
from us.lsi.tools import Draw
from  matplotlib.patches import Patch

@dataclass(frozen=True,order=True)
class Segmento2D(Objeto2D):
    p1: Punto2D
    p2: Punto2D
```

Con métodos de factoría y representación:

```
@staticmethod
def of_puntos(p1:Punto2D,p2:Punto2D) -> Segmento2D:
      return Segmento2D(p1,p2)

def __str__(self):
      return '({0},{1})'.format(str(self.p1),str(self.p2))
```

Y propiedades:

```
@property
def copy(self) -> Segmento2D:
      return Segmento2D(self.p1,self.p2)
```

```
@property
def vector(self) -> Vector2D:
      return self.p1.vector_to(self.p2)
```

```
@property
def modulo(self) -> float:
       return self.p1.distancia_a(self.p2)
```

Los métodos heredados de Objeto2D se implementan reutilizando los correspondientes de Punto2D.

```
def rota(self,p:Punto2D,angulo) -> Segmento2D:
        return Segmento2D.of_puntos(self.p1.rota(p,angulo),\
             self.p2.rota(p,angulo))
```

```
def traslada(self, v:Vector2D) -> Segmento2D:
       return Segmento2D.of_puntos(self.p1.traslada(v), \
             self.p2.traslada(v))
```

```
def homotecia(self, p:Punto2D, factor) -> Segmento2D:
       return Segmento2D.of(self.p1.homotecia(p,factor), \
             self.p2.homotecia(p,factor))
```

```
def proyecta_sobre_recta(self,r:Recta2D) -> Segmento2D:
        return Segmento2D.of(self.p1.proyecta_sobre_recta(r),\
             self.p2.proyecta_sobre_recta(r))
```

```
def simetrico_con_respecto_a_recta(self,r:Recta2D) ->
             Segmento2D:
        return Segmento2D.of(self.p1.simetrico(r), \
             self.p2.simetrico(r))
```

Circulo2D

El circulo es un objeto geométrico definido por un punto y un radio. No incluimos los import por ser similares a los tipos anteriores. El código completo se puede encontrar en el repositorio.

```
@dataclass(frozen=True,order=True)
class Circulo2D(Objeto2D):
    centro: Punto2D
    radio:float

    @staticmethod
    def of(centro: Punto2D, radio:float) -> Circulo2D:
        Preconditions.checkArgument(radio>=0, 'El radio debe\
ser mayor o igual a cero y es {0:.2f}'.format(radio))
        return Circulo2D(centro, radio)
    def __str__(self) -> str:
        return '({0},{1:.2f})'\
             .format(str(self.centro),self.radio)
```

En el método de factoría no aseguramos que el radio sea mayor o igual a cero. Algunas propiedades del tipo son:

```
@property
    def area(self) -> float:
        return pi*self.radio*self.radio

    @property
    def perimetro(self) -> float:
        return 2*pi*self.radio
```

Los métodos heredados de Objeto2D se implementan reutilizando los métodos de Punto2D.

```
def rota(self, p:Punto2D, angulo:float) -> Circulo2D:
        return Circulo2D.of(self.centro.rota(p,angulo),\
             self.radio)

def traslada(self, v: Vector2D) -> Circulo2D:
        return Circulo2D.of(self.centro.traslada(v),\
             self.radio)
```

```
def homotecia(self, p: Punto2D, factor:float) -> Circulo2D:
        return Circulo2D.of(self.centro.homotecia(p,factor),\
             self.radio*factor)

def simetrico_con_respecto_a_recta(self, r: Recta2D) ->\
      Circulo2D:
        return Circulo2D.of(self.centro.simetrico(r),
             self.radio)
```

La proyección dará lugar a un segmento cuyo centro estará en la proyección del centro del circulo y cuyo módulo es el dobel del radio.

```
def proyecta_sobre_recta(self, r: Recta2D) -> Segmento2D:
        c = self.centro.proyectaSobre(r)
        u = r.vector.unitario()
        return Segmento2D.of_puntos(c+u*self.radio,\
             c-u*(self.radio))
```

Poligono2D

El polígono es un objeto geométrico definido por una lista de puntos. No incluimos los import que pueden ser encontrados en el repositorio.

```
@dataclass(frozen=True,order=True)
class Poligono2D(Objeto2D):
    vertices: list[Punto2D]

    @staticmethod
    def of_vertices(vertices: list[Punto2D]) -> Poligono2D:
        return Poligono2D(vertices)
    def __str__(self) -> str:
        return '({0})'.format(','.join(str(p) \
             for p in self.vertices))
```

Podemos definir métodos de factoría para construir polígonos particulares: triángulos, triángulos equiláteros, cuadrados, rectángulos, etc.

```
@staticmethod
    def triangulo(p1:Punto2D, p2:Punto2D, p3:Punto2D) ->\
             Poligono2D:
        return Poligono2D.of([p1, p2, p3])
```

```
@staticmethod
def triangulo_equilatero(p1:Punto2D, lado:Vector2D)-> \
             Poligono2D:
        return Poligono2D.of([p1, p1+lado, \
             p1+lado.rota(pi/3)])
```

```
@staticmethod
def cuadrado(p:Punto2D,lado:Vector2D) -> Poligono2D:
        p1 = p+lado
        lo = lado.ortogonal
        p2 = p1+lo
        p3 = p+lo
        return Poligono2D.of([p,p1,p2,p3])
```

```
@staticmethod
def rectangulo(p:Punto2D, base:Vector2D, altura:float) -> \
             Poligono2D:
        p1 = p+base
        p2 = p+base+base.ortogonal.unitario*altura
        p3 = p+base.rota(pi/2).unitario*altura
        return Poligono2D.of([p,p1,p2,p3])
```

```
@staticmethod
def rectanguloHorizontal(x_min:Punto2D, x_max:Punto2D,\
             y_min:Punto2D, y_max:Punto2D) -> Poligono2D:
        p0 = Punto2D.of(x_min, y_min)
        p1 = Punto2D.of(x_max, y_min)
        p2 = Punto2D.of(x_max, y_max)
        p3 = Punto2D.of(x_min, y_max)
        return Poligono2D.of([p0,p1,p2,p3])
```

Veamos algunas de sus propiedades: copy, area, perímetro, número de vertices, vertice(i), lado(i), diagonal(i). El vertice(i) es un Punto2D, el lado(i) y la diagonal(i) ambos Vector2D.

```
def vertice(self,i)-> Punto2D:
        Preconditions.checkElementIndex(i, self.n)
        return self.vertices[i]

def lado(self,i:int) -> Vector2D:
        Preconditions.checkElementIndex(i, self.n);
        return self.vertice(i)\
             .vector_to(self.vertice((i+1)%self.n))

def diagonal(self,i:int,j:int) -> Vector2D:
        Preconditions.checkElementIndex(i, self.n);
        Preconditions.checkElementIndex(j, self.n);
        return self.vertice(i).vector_to(self.vertice(j))
```

Ahora podemos calcular el perímetro como la suma de los módulos de los lados y el área como la suma de los triángulos definidos por dos diagonales consecutivas. La copia se hace de la manera usual.

```
@property
def copy(self) -> Poligono2D:
       return Poligono2D.of(self.vertices)

@property
def area(self) -> float:
       area = sum(self.diagonal(0,i)\
       .multiply_vectorial_2d(self.diagonal(0,i+1)) \
       for i in range(1,self.n-1))
       return area/2

@property
def perimetro(self)->float:
       return sum(self.lado(i).modulo for i in range(self.n))
```

Los métodos heredados de Objeto2D pueden ser implementados basados en los de Punto2D.

```
def rota(self, p:Punto2D, angulo:float) -> Poligono2D:
        return Poligono2D.of([x.rota(p,angulo) for x in
self.vertices])
```

```
def traslada(self, v:Vector2D) -> Poligono2D:
        return Poligono2D.of([x.traslada(v)
              for x in self.vertices])
```

```
def homotecia(self, p:Punto2D, factor:float) -> Poligono2D:
        return Poligono2D.of([x.homotecia(p,factor) \
              for x in self.vertices])
```

```
def proyecta_sobre_recta(self,r:Recta2D) -> set[Punto2D]:
        return Poligono2D.of([x.proyecta_sobre_recta(r) \
              for x in self.vertices])
```

```
def simetrico_con_respecto_a_recta(self, r:Recta2D) ->\
              Poligono2D:
        return Poligono2D.of(\
              [x.simetrico_con_respecto_a_recta(r)\
              for x in self.vertices])
```

Agregado2D

Un agregado será una colección de objetos geométricos de tipo Objeto2D. Esto implica que un agregado pued contener objetos simples u otros agregados.

Este tipo implement a los métodos heredados de Objeto2D.

```
@dataclass(frozen=False)
class Agregado2D(Objeto2D):
    objetos: list[Objeto2D]

    @staticmethod
    def of(objetos: list[Objeto2D]) -> Agregado2D:
        return Agregado2D(objetos)

    def __str__(self) -> str:
        return '({0})'.format(','.join(str(p) \
              for p in self.objetos))
```

El tipo Agregado2D es inmutable pero tiene una propiedad mutable puesto que objetos es un a lista.

Esto hace que podamos modificar esa propiedad del agregado diseñando métodos para añadir objetos geométricos.

Si quisiéramos hacer el tipo completamente inmutable deberíamos hacer privada la propiedad objetos, no permitir métodos que la mofiquen y cuando se quiera consultar devolver una copia. Si permitimos que se pueda modificar la propiedad objetos diseñamos métodos para añadir uno o varios.

```
def add(self,objeto:Objeto2D)->None:
        self.objetos.append(objeto)

def add_list(self,objetos:list[Objeto2D])->None:
        for e in objetos:
            self.objetos.append(e)
```

Diseñamos los métodos heredados de Object2D invocando a los respectivos métodos de los objetos en el agregado. Estos métodos tienen una implementación específica para cada tipo de objetos.

```
def copy(self):
        return Agregado2D.of([e.copy for e in self.objetos])

def rota(self, p:Punto2D, angulo:float) -> Agregado2D:
        return Agregado2D.of([x.rota(p,angulo) \
             for x in self.objetos])

def traslada(self, v:Vector2D) -> Agregado2D:
        return Agregado2D.of([x.traslada(v) \
             for x in self.objetos])
```

```
def homotecia(self, p:Punto2D, factor:float) -> Agregado2D:
        return Agregado2D.of([x.homotecia(p,factor) for x in
self.objetos])
```

```
def proyecta_sobre_recta(self,r:Recta2D) -> Agregado2D:
        return Agregado2D.of([x.proyecta_sobre_recta(r) \
             for x in self.vertices])
```

```
def simetrico_con_respecto_a_recta(self, r:Recta2D) ->
              Agregado2D:
        return Agregado2D.of([x.simetrico(r) \
              for x in self.objetos])
```

Todos estos tipos pueden usarse de la forma:

```
p = Punto2D.of(1., -1.)
pr = Punto2D.of(0., 0.)
vr = Vector2D.of(1,1)*5.
r = Recta2D.of(pr, vr)
p2 = p.proyecta_sobre_recta(r)
p3 = p.simetrico_con_respecto_a_recta(r)
pol = Poligono2D.cuadrado(Punto2D.of(3.,4.),\
              Vector2D.of(1,-2))
pol2 = pol.simetrico_con_respecto_a_recta(r)
s = Segmento2D.of_puntos(pr, pr + vr)
a = Agregado2D.of([p,p2,p3,s,pol])
a.add(pol)
b = a.rota(Punto2D.origen(),-pi/3)
```

Representación gráfica de los objetos geométricos

La representación gráfica de estos objetos podemos hacerla usando el tipo Patch de la librería *matplotlib.patches* de Python. La idea es asociar a cada objeto geométrico un Patch y posteriormente mostrar todos esos patches. Podemos diseñar funciones para este propósito en el módulo Draw.

```
def shape_circle(center:tuple[float],radio:float,fill=None)\
      ->Patch:
    return plt.Circle(center,radio, fill=fill, color=color)
```

```
def shape_multiline(points:list[tuple[float,float]],\
      closed=None, fill=None)->Patch:
    return plt.Polygon(points, closed=closed, fill=fill, \
      edgecolor=color)
```

Disponiendo de estas funciones completaremos los métodos shape de cada uno de los objetos geométricos:

```
#Punto2D
@property
    def shape(self)->Patch:
        return Draw.shape_circle([self.x,self.y],
             radio=0.05,fill=True)
```

```
#Segmento2D
@property
    def shape(self)->Patch:
        return shape_multiline([[self.p1.x,self.p1.y],
             [self.p2.x,self.p2.y]],closed=False)
```

```
#Circulo2D
@property
    def shape(self)->list[Patch]:
        return Draw.shape_circle(\
             [self.center.x,self.center.y],\
             self.radio,fill=False)
```

```
#Poligono2D
@property
    def shape(self)->Patch:
        return Draw.shape_multiline([[p.x,p.y] for p in
self.vertices],closed=True)
```

```
#Agregado2D
@property
    def shape(self):
        return [p.shape for p in self.objetos]
```

La tarea la completamos diseñando una función que dibuja un conjunto de patches.

```
def draw_shapes(shapes:list[Patch]):
    plt.axes()
    for f in shapes:
        plt.gca().add_patch(f)
    plt.axis('scaled')
    plt.show()
```

Montecarlo

Usaremos ahora el método de Montecarlo para estimar la probabilidad de victoria de una determinada mano de póker. El póker es un juego de cartas de apuestas, en el que el jugador que tiene la mejor combinación de cartas gana toda la cantidad aspostada en una mano. Tiene elementos de muchos juegos antiguos pero su expansión, con reglas parecidas a las que se usan hoy en día, tuvo lugar desde Nueva Orleans en el siglo XIX. Se juega con la denominada baraja inglesa que tiene trece cartas de cuatro palos distintos. Las trece cartas ordenadas de menor a mayor según su valor son:

```
['2','3','4','5','6','7','8','9','10','J','Q','K','A']
```

No hay diferencia de valor entre los cuatro palos: *Tréboles, Corazones, Picas, Diamantes.*

El póker se basa en un *ranking* bastante heterogéneo de jugadas. En la siguiente tabla se muestran las jugadas con las que trabajaremos en este ejercicio, junto con sus descripciones y la probabilidad de cada combinación. Consideraremos una baraja de 52 cartas sin comodines, por lo que la jugada del *repóker* no será posible:

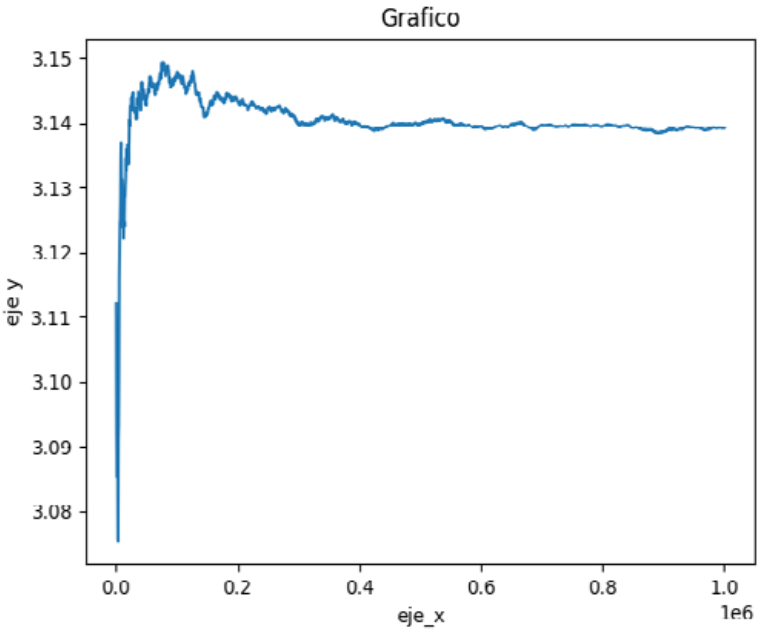
Jugada		Descripción	Probabilidad
Escalera real	ER	Cinco cartas seguidas del mismo palo del 10 al as	4 de 2.598.960 ($1{,}539 \cdot 10^{-4}$ %)
Escalera de color	EC	Cinco cartas consecutivas del mismo palo	36 de de 2.598.960 ($1{,}385 \cdot 10^{-3}$ %)
Póker	P	Cuatro cartas iguales en su valor	624 de 2.598.960 ($2{,}4 \cdot 10^{-2}$ %)
Full	F	Tres cartas iguales en su valor (tercia), más otras dos iguales en su valor (pareja)	3.744 de 2.598.960 (0,144 %)
Color	C	Cinco cartas del mismo palo, sin ser necesariamente consecutivas	5.108 de 2.598.960 (0,196 %)
Escalera	E	Cinco cartas consecutivas sin importar el palo	10.200 de 2.598.960 (0,392 %)
Trío	T	Tres cartas iguales de valor	54.912 de 2.598.960 (2,111 %)
Doble pareja	D	Dos pares de cartas del mismo valor (par y par)	123.552 de 2.598.960 (4,759 %)
Pareja	P	Dos cartas del mismo valor (y tres diferentes)	1.098.240 de 2.598.960 (42,257 %)
Carta alta	C	Gana quien tiene la carta más alta	1.302.540 de 2.598.960 (50,117 %

El objetivo del problema estimar en que propoción una mano gana a otra escogida aletoriamente. Para ello usaremos el método de Montecarlo.

Este funcionamiento de este método podemos comprenderlo estimando el área de un círculo por este método. Para ello sorteamos dos números

reales en los intervalos (-1.,1.). Todos los puntos caen en un cuadrado de área 1. El número de puntos que están a un distancia del (0.,0.) < 1 forman un círculo cuyo área queremos estimar. Sea f la proporción de puntos que están a una distancia del centro igual o menor de 1. El área del cuadrado es 4 luego la estimación del área del circulo es 4f.

Como el radio del circulo es 1 su área es π por lo que el método nos puede servir también para hacer una estimación de π. Si repetimos el cálculo para distintos valores de n cada vez más grandes vemos como el resultado se va aproximando a π.



El código para una aproximación es:

```
def montecarlo(n:int):
    puntos = (Punto2D.of(random.uniform(-1.,1.),
              random.uniform(-1.,1.)) for _ in range(n))
    pt=0
    pi = 0
    for p in puntos:
        pt = pt +1
        if p.distancia_al_origen <= 1:
            pi = pi +1
    return 4*pi/pt
```

Una aproximación con n=`1000000` es `3.138704`.

Tipos

Valor: enum [2,3,4,5,6,7,8,9,10,J,Q,K,A]

Palo: enum [T,C,P,D]

Jugada: enum [C,P,D,T,E,C,F,P,EC,ER]

Carta:

Propiedades:

- *Valor*: Integer en [0..13), básica, ordenadas de mayor a menor valor
- *Palo*: Integer en [0..4), básica
- *Id*: Integer en [0..53), derivada, *id = palo*4+valor*;
- *NombresDeValores*: List<String>, compartida *NombresDePalos: List<Character>, compartida*,

Representación: "VP".

Factoría:

- Dado el id
- Dado valor y palo
- Dada su representación textual

Orden natural: por valor

Mano:

Propiedades:

- *Cartas*: List<Carta>, tamaño *numeroDeCartas, ordenadas por valor*
- *NumeroDeCartas*, Integer, compartida, 5
- *Son5ValoresConsecutivos: Boolean, derivada*
- *FrecuenciasDeValores: Map<Integer,Integer>, derivada*
- *ValoresOrdenadosPorFrecuencias(): List<Integer>, de mayor a menor*
- *ValorMasFrecuente: Integer, derivada, el valor que más se repite*
- *FrecuenciasDePalos: Map<Integer,Integer>,*
- *PalosOrdenadosPorFrecuencias: List<Integer>, de mayor a menor*

- *EsColor: Boolean, derivada*
- *EsEscalera: Boolean, derivada*
- *EsPoker: Boolean, derivada*
- *EsEscaleraDeColor: Boolean, derivada*
- *EsFull: Boolean, derivada*
- *EsTrio, Boolean, derivada*
- *EsDoblePareja: Boolean, derivada*
- *EsPareja:Boolean, derivada*
- *EsEscaleraReal:Boolean, derivada*
- *EsMano:Boolean, , derivada*
- *Jugada:Jugada, derivada*
- *Fuerza: Double, derivada, la probabilidad de ganar a un mano aleatoria*

Representación: Representación de las cartas entre paréntesis y separadas por comas

Factoría:

- *Random*, selecciona 5 cartas sin repetición al azar
- Dada un Set<Carta> de tamaño 5
- Dada la representación textual de una mano

Orden Natural: Por tipo de jugada y si es la misma por el valor que más se repite

El tipo Carta lo inplemententamos en una clase.

```
from __future__ import annotations
from us.lsi.tools import Preconditions
from dataclasses import dataclass

nombre_valores = \
       ["2","3","4","5","6","7","8","9","10","J","Q","K","A"]
symbols_palos = ['C', 'H', 'S', 'D']
nombre_palos = ["clubs","hearts","spades","diamonds"]

@dataclass(frozen=True,order=True)
class Card:
    palo:int  # [0,4)
    valor:int # [0,14)
    _ide:int = None # palo*4+valor # [0,52)
```

La representación y la propiedad id que calculamos una sola vez.

```
def __str__(self):
        return '{0}{1}'\
       .format(nombre_valores[self.valor],\
       symbols_palos[self.palo])
```

```
@property
def id(self) -> int:
        if self._id is None:
            self._id = self.palo*4+self.valor
        return self._id
```

Los métodos de factoría:

```
@staticmethod
def of(palo,valor):
        Preconditions.checkArgument(\
       valor >= 0 and valor <14 and palo >=0 and palo < 52,\
       "No es posible valor = %d, palo = %d"\
       .format(valor,palo))
        return Card(palo,valor)
```

```
@staticmethod
    def of_id(ide):
        Preconditions.checkArgument(\
       ide >= 0 and ide < 52, \
       "No es posible {0:d}".format(ide))
        palo = ide % 4
        valor = ide % 13
        return Card(palo,valor)
```

```
@staticmethod
def of_text(text:str)->Card:
        p = text[len(text)-1]
        v = text[0:len(text)-1]
        palo = symbols_palos.index(p)
        valor = nombre_valores.index(v)
        return Card.of(palo, valor)
```

Ahora podemos crear cartas:

```
print(Card.of_id(34))
print(Card.of_text('10S'))
```

Veamos ahora el tipo Mano. Una mano es un conjunto des cinco cartas elegidas al azar. Tiene la propiedad básica *cartas* que es una lista. Junto con ella hay dos propiedades relevantes: jugada y fuerza. La primera es el tipo de jugada que está asociada a la mano. La segunda es la fuerza de la mano, es decir la probabilidad de ganar a una mano aleatoria. Para el cála culo de ambas hay que hacer cálculos intermedios: frecuencias de valores, valores ordenados por frecuencias, frecuencias de palos, palos ordenados por frecuencias, son 5 valores consecutivos. Veamos la implementación de la clase y de estos métodos intermedios que implementamos para hacer el cálculo una sola vez.

```
nombres_jugadas: list[str]=['EscaleraReal','EscaleraDeColor',\
       'Poker','Full','Color', 'Escalera','Trio',
       'DoblePareja','Pareja','CartaMasAlta']
numero_de_cartas: int = 5

@total_ordering
class Mano:
    def __init__(self, cartas):
        self._cartas: list[Card] = cartas
        self._frecuencias_de_valores: dict[int,int] = None
        self._valores_ordenados_por_frecuencias: list[int] = \
             None
        self._son_5_valores_consecutivos: bool = None
        self._frecuencias_de_palos: dict[int,int] = None
        self._palos_ordenados_por_frecuencias: list[int] = \
                    None
        self._jugada: int = None
        self._fuerza: float = None
```

```
@property
def frecuencias_de_valores(self) -> dict[int,int]:
        if not self._frecuencias_de_valores:
            self._frecuencias_de_valores = \
              frequencies(self._cartas,fkey=lambda c:c.valor)
        return self._frecuencias_de_valores
```

```
@property
def valores_ordenados_por_frecuencias(self) -> list[int]:
        if not self._valores_ordenados_por_frecuencias:
            ls  = [e for e in \
                    self.frecuencias_de_valores.items()]
            ls.sort(key = lambda e: e[1], reverse= True)
            ls = [e[0] for e in ls]
            self._valores_ordenados_por_frecuencias = ls
        return self._valores_ordenados_por_frecuencias
```

```
@property
def frecuencias_de_palos(self) -> dict[int,int]:
        if not self._frecuencias_de_palos:
            self._frecuencias_de_palos = \
             frequencies(self._cartas,fkey=lambda c:c.palo)
        return self._frecuencias_de_palos
```

```
@property
def palos_ordenados_por_frecuencias(self) -> list[int]:
        if not self._palos_ordenados_por_frecuencias:
            ls  = [e for e in \
                    self.frecuencias_de_palos.items()]
            ls.sort(key = lambda e: e[1], reverse= True)
            ls = [e[0] for e in ls]
            self._palos_ordenados_por_frecuencias = ls
        return self._palos_ordenados_por_frecuencias
```

```
@property
def son_5_valores_consecutivos(self) -> bool:
        if not self._son_5_valores_consecutivos:
            ls = self.valores_ordenados_por_frecuencias
            self._son_5_valores_consecutivos = False
            if len(ls) == 5:
                self._son_5_valores_consecutivos = \
             all(ls[x+1]-ls[x]==1 for x in range(0,len(ls)-1))
        return self._son_5_valores_consecutivos
```

Usando esos métodos podemos impletar los métodos que definen el tipo de jugada.

```
@property
def es_color(self)-> bool:
        pal0 = self.palos_ordenados_por_frecuencias[0]
        return self.frecuencias_de_palos[pal0] == 5
```

```
@property
def es_escalera(self)-> bool:
        return self.son_5_valores_consecutivos
```

```
@property
def es_poker(self)-> bool:
        val0 = self.valores_ordenados_por_frecuencias[0]
        return self.frecuencias_de_valores[val0] == 4
```

```
@property
def es_escalera_de_color(self)-> bool:
        pal0 = self.palos_ordenados_por_frecuencias[0]
        return self.son_5_valores_consecutivos and \
             self.frecuencias_de_palos[pal0] == 5
```

```
@property
def es_full(self) -> bool:
        r = False
        if len(self.valores_ordenados_por_frecuencias) >= 2 :
            val0 = self.valores_ordenados_por_frecuencias[0]
            val1 = self.valores_ordenados_por_frecuencias[1]
            r = self.frecuencias_de_valores[val0] == 3 and \
              self.frecuencias_de_valores[val1] == 2
        return r
```

```
@property
def es_trio(self)-> bool:
        val0 = self.valores_ordenados_por_frecuencias[0]
        return self.frecuencias_de_valores[val0] == 3
```

```
@property
def es_doble_pareja(self)-> bool:
        r = False
        if len(self.valores_ordenados_por_frecuencias) >= 2 :
            val0 = self.valores_ordenados_por_frecuencias[0]
            val1 = self.valores_ordenados_por_frecuencias[1]
            r = self.frecuencias_de_valores[val0] == 2 and \
              self.frecuencias_de_valores[val1] == 2
        return r
```

```
@property
def es_pareja(self)-> bool:
        val0 = self.valores_ordenados_por_frecuencias[0]
        return self.frecuencias_de_valores[val0] == 2
```

```
@property
def es_escalera_real(self)-> bool:
        return self.es_escalera_de_color and \
             {x.valor for x in self.cartas}.contains(12)
```

```
@property
def es_carta_mas_alta(self)-> bool:
        return True
```

Para decidir el tipo de juagada debemos ejecutar secuencialmente los predicados anteriores. El primer predicado que devuelva verdadero decidirá el tipo de jugada. Una manera compacta de hacer esto es definir una lista con las propiedades boolenas. Recorrerla y encontrar el primer índice que es verdadero.

```
@property
def predicados_jugadas(self) -> list[bool]:
        return  [self.es_escalera_real, \
             self.es_escalera_de_color, \
             self.es_poker, \
             self.es_full, \
             self.es_color, \
             self.es_escalera, \
             self.es_trio, \
             self.es_doble_pareja, \
             self.es_pareja, \
             self.es_carta_mas_alta]
```

```
@property
def jugada(self) -> int:
        if not self._jugada:
            self._jugada = index_bool(self.predicados_jugadas)
        return self._jugada
```

La función index_bool recorre un iterable y en cuentra la posición del primero verdadero. Una jugada es mejor que otra cuando tiene asociado un entero mayor. Con este criterio definimos una relación de orden total entre jugadas que implementamos con los métodos *__eq__, __ne__, __lt__*.

```
def __eq__(self,mano):
        return self.jugada == mano.jugada and \
            self.valores_ordenados_por_frecuencias[0] == \
                    mano.valores_ordenados_por_frecuencias[0]
```

```
def __lt__(self,mano):
        r = False
        if  self.jugada > mano.jugada:
            return True
        if self.jugada == mano.jugada and \
            self.valores_ordenados_por_frecuencias[0] <
             mano.valores_ordenados_por_frecuencias[0]:
            return True
        return r
```

Por último podemos implementar el cálculo de la fuerza de una mano. Lo hacemos por el método de Montecarlo. Es decir es el porcentaje de veces que gana a una mano escogida al azar.

```
def fuerza(self, n=5000) -> float:
        if self._fuerza:
            return self._fuerza
        gana = 0;
        pierde = 0;
        empata = 0;
        for _ in range(n):
            mr = Mano.random()
            if self < mr :
                pierde = pierde+1
            elif self > mr:
                gana = gana +1
            elif self == mr:
                empata = empata+1
        self._fuerza = gana/(gana+pierde+empata)
        return self._fuerza
```

Junto a lo anterior necesitamos los métodos de factoría, la representación y algunas propiedades.

```
@staticmethod
def of(cartas):
        return Mano(cartas)
```

```
@staticmethod
def of_text(txt):
        txt = txt[1:len(txt)-1]
        partes = txt.split(",")
        cartas = [Card.of_text(x) for x in partes]
        return Mano.of(cartas)
```

```
@staticmethod
def random():
        cartas = []
        for _ in range(numero_de_cartas):
            n = random.randint(0,51)
            card = Card.of_id(n)
            cartas.append(card)
        return Mano(cartas)
```

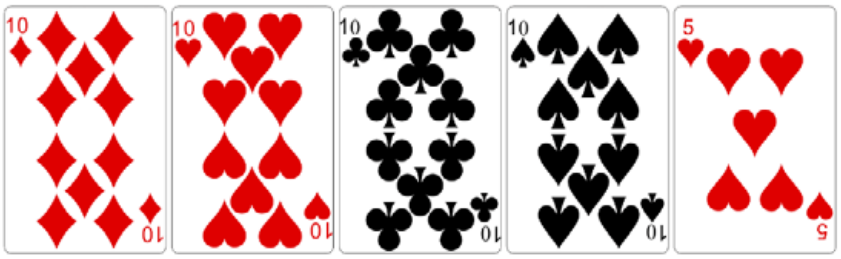
```
@property
def cartas(self) -> list[Card]:
        return self._cartas
```

```
def __str__(self):
        mano = joining((c for c in self.cartas),\
             separator=',',prefix='[',suffix=']')
        return '{}={}={}'.format(mano,self.nombre_de_jugada,\
             str(self.fuerza()))
```

Lo anterior podemos usarlo en la forma:

```
m1 = Mano.random()
m2 = Mano.of_text('[7H,8H,3C,3S,6H]')
m3 = Mano.of_text('[10D,10H,10C,10S,5H]')
print(m1)
print(m2)
print(m3)
print(m1 < m2)
```

Los cálculos anteriores pueden ser visualizados en ficheros html como el siguiente. El código necesario puede encontrarse en el repositorio.



Su tipo es = Poker

Su fuerza es = 0.9992

Herramientas

En este capítulo incluimos ejemplos de herramientas útiles en Pyhton.

1. *Diseñar una función que devuelva el directorio raiz del proyecto actual*

```
def root_project():
    return sys.path[1]
```

2. *Diseñar una función que devuelva la dirección absoluta de un fichero dada su dirección relativa la proyecto actual*

```
def absolute_path(file:str)->str:
    return root_project()+file
```

3. *Diseñar una función que nod devuelva el directorio actual*

```
def dir_path()->str:
    return os.getcwd()
```

4. *Diseñar una función que nos indique si un fichero existe o no*

```
def existeFichero(filePath)->bool:
    return os.path.isfile(filePath)
```

5. *Diseñar un función que nos devuelva el encondin de un fichero*

```
def encoding(file:str)->str:
    checkArgument(existeFichero(file),\
              'El fichero {} no existe'.format(file))
    with open(file,"rb") as f:
        data = f.read()
        enc = chardet.detect(data)
        return enc['encoding']
```

6. *Diseñar una función que sustituya un conjunto de identificadores delimitados por {} por cadenas de texto asociados mediante un diccionario*

```
def transform(inText:str,reglas:dict[str,str]) -> str:
    outText = inText
    for e,s in reglas.items():
        outText = re.sub(r'\{'+e+'\}',s,outText)
    return outText
```

Bibliografía

Varios enlaces a material de Pyhton y un libro.

1. Python 3 tutorial:

   https://docs.python.org/3/tutorial/

2. Otro tutorial dePython:

   https://python-course.eu/python3_course.php

3. The Python Standard Library

   https://docs.python.org/3/library/

4. Libraries in Python

   https://www.geeksforgeeks.org/libraries-in-python/

5. Python 3: Los fundamentos del lenguaje (2ª edición). Sébastien Chazallet. Ediciones ENI, 2016. ISBN: 409-00614-2

Miguel Toro es doctor en Ingeniería Industrial por la Universidad de Sevilla, institución en la que desarrolla su labor docente e investigadora como catedrático del Departamento de Lenguajes y Sistemas Informáticos y en la que ejerce también como director del Instituto de Ingeniería Informática.

Ha ocupado diversos cargos de responsabilidad, entre los que cabe señalar los siguientes: director de la Oficina de Transferencia de Resultados de la Investigación (OTRI), director general de Investigación, Tecnología y Empresa de la Junta de Andalucía, presidente de Sistedes (Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software) y presidente de la Sociedad Científica Informática de España (SCIE), que engloba a los informáticos de las universidades españolas.

Colabora asiduamente con varias agencias nacionales de evaluación universitaria; en este sentido, ha tenido un papel activo en la Agencia Andaluza de Evaluación de la Calidad y Acreditación Universitaria (AGAE) y en el consejo asesor de la Agencia Nacional de Evaluación de la Calidad y Acreditación (ANECA).

En su extensa trayectoria profesional cuenta con varios reconocimientos: Premio Fama de la Universidad de Sevilla, Premio Sistedes de la Sociedad Nacional de Ingeniería del Software y Tecnologías de Desarrollo de Software (en reconocimiento a su labor de promoción y consolidación de la Informática en España) y Premio Nacional de Informática José García Santesmases a la trayectoria profesional, otorgado por la Sociedad Científica Informática de España.

Este volumen está dirigido a los alumnos de primero de los grados en Ingeniería Informática, especialmente a aquellos que cursan la asignatura Fundamentos de Programación, dado que se exponen aquí los conceptos básicos de la primera parte de esta materia, desglosados en los siguientes puntos:

— Introducción a Python. Expresiones y tipos básicos
— Sentencias de control de flujo y abstracción funcional
— Programación Orientada a Objetos. Diseño de tipos de datos
— Tipos de agregados de datos
— Esquemas secuenciales: estilo imperativo, comprensión de colecciones y generadores, iteradores y sus operadores

Editorial Universidad de Sevilla