Tema 1. Introducción al lenguaje de programación Python.

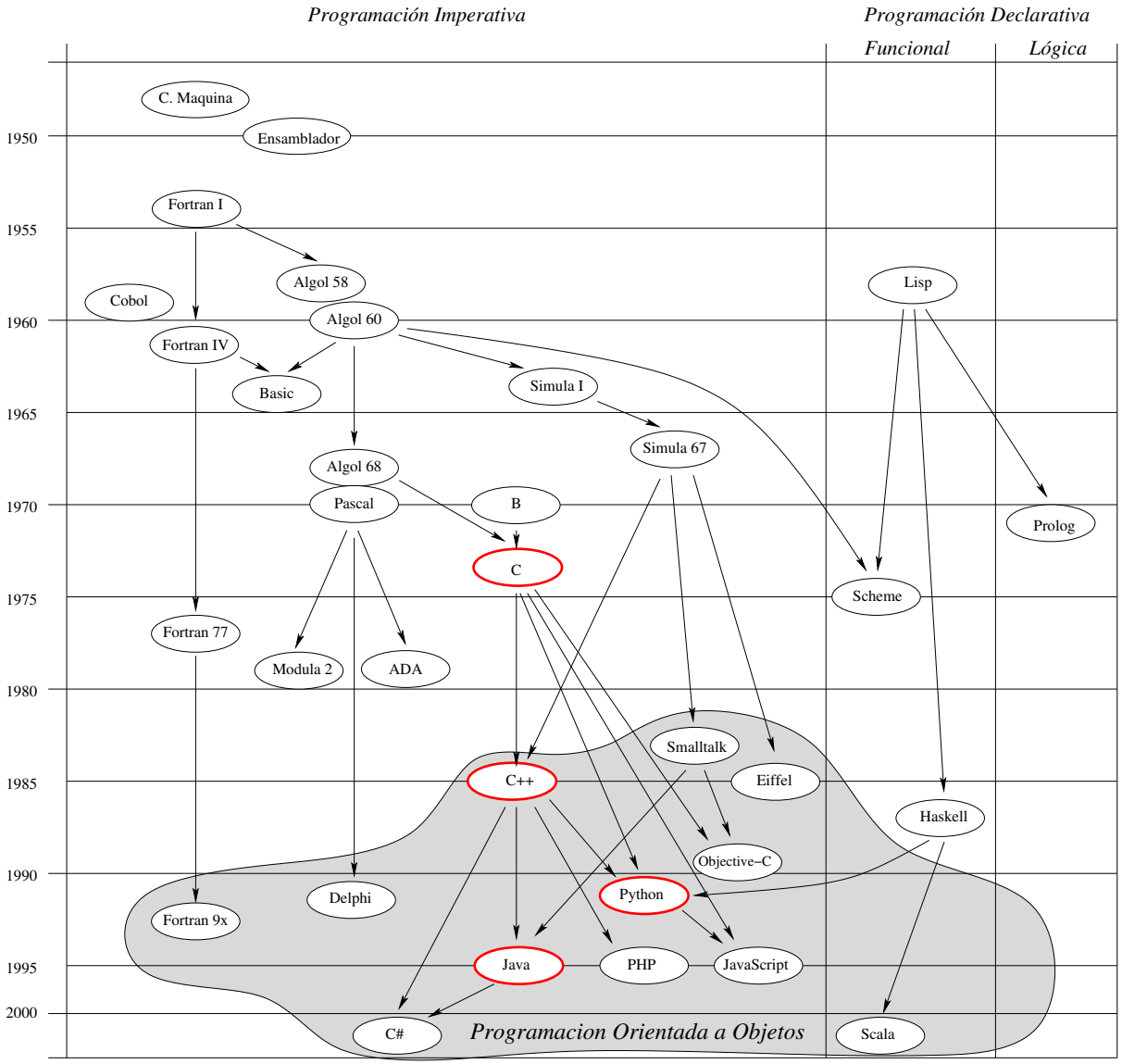
Vicente Benjumea García

Programación-II
Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

Tema 1. Introducción al lenguaje de programación Python.

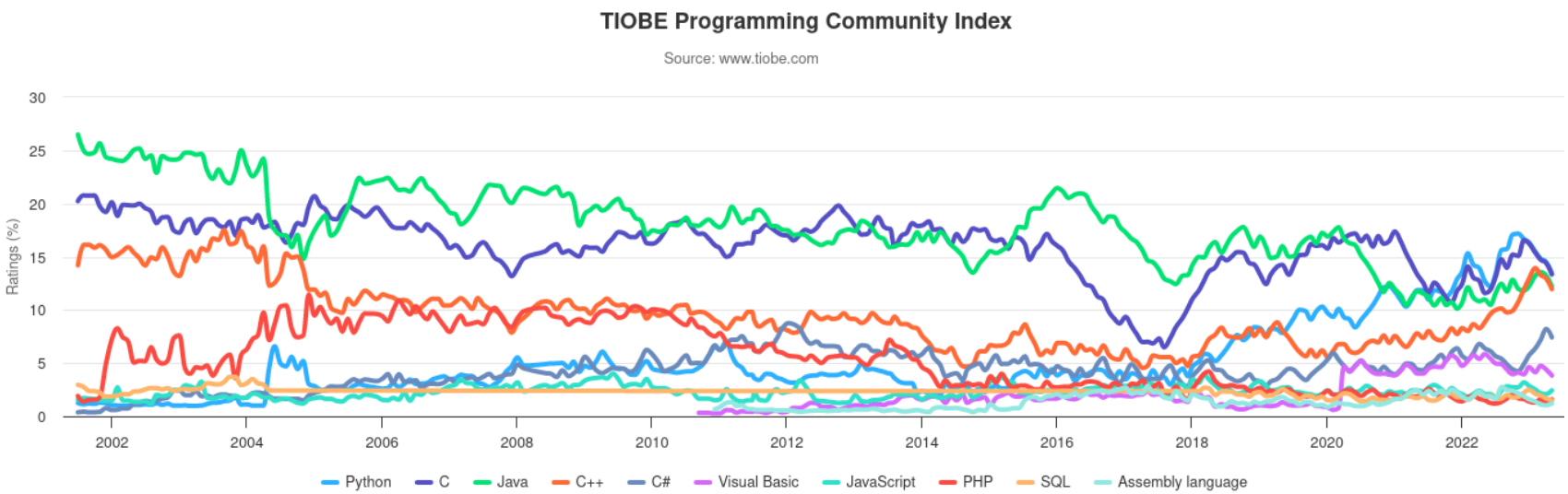
- Introducción histórica y evolución de los lenguajes de programación.
- Módulos, paquetes y programas en Python.
- Elementos del lenguaje.
- Entrada y salida de datos.
- Definición e invocación a funciones. Parámetros y argumentos.
- Anotaciones de tipo.
- Cadenas de caracteres.
- Tuplas.
- Listas.
- Conjuntos.
- Diccionarios.
- Funciones incorporadas.
- Módulo de funciones matemáticas.
- Comparación de rendimiento.
- Herramientas de desarrollo.

Introducción histórica y evolución de los lenguajes de programación



Introducción histórica y evolución de los lenguajes de programación

- Clasificación de los lenguajes más *populares* (2023) según *Tiobe* (`https://www.tiobe.com/tiobe-index/`):

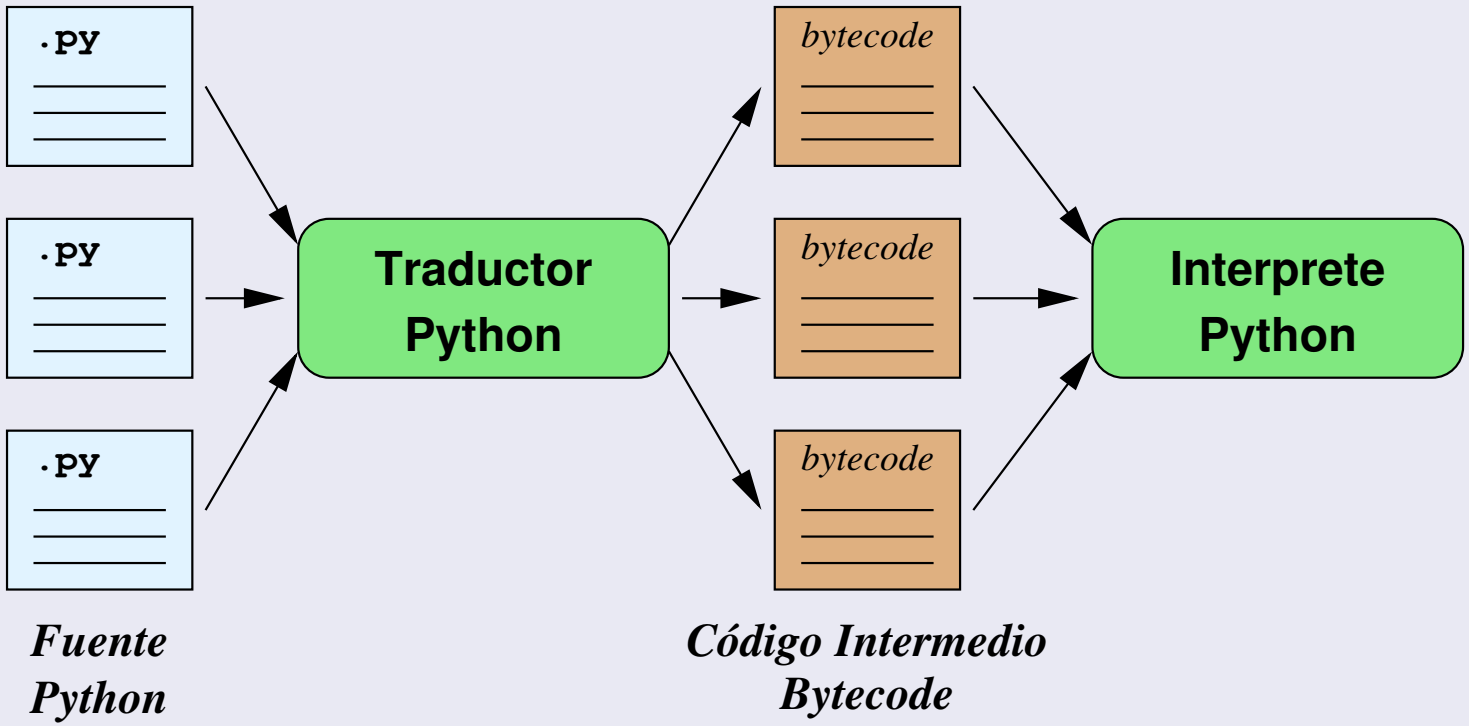


Introducción al lenguaje de programación Python (I)

- El lenguaje de programación *Python*, ha sido creado y desarrollado por *Guido van Rossum* a principios de 1990 (`https://www.python.org/`).
- Existen varias versiones del lenguaje de programación *Python*, siendo la versión 3.x la versión actual (desde 2008).
  - Es importante considerar que la versión 2.x de Python **no** es compatible con las versiones actuales de python (versión 3.x), por lo que un programa desarrollado en python 2.x no es válido en las versiones actuales.
- El lenguaje Python es un lenguaje *multi-paradigma*, proporciona soporte para los siguientes paradigmas de programación:
  - Programación estructurada.
  - Abstracción procedimental.
  - Programación modular.
  - Abstracción de datos.
  - Programación orientada a objetos.
  - Programación concurrente.

Introducción al lenguaje de programación Python (II)

- Python es un lenguaje que sigue un esquema de traducción *mixto*.
  - Los programas con código fuente en Python se **traducen** a un código intermedio denominado **bytecode**.
  - El código intermedio *bytecode* se **interpreta** y ejecuta por la máquina virtual de python.



Introducción al lenguaje de programación Python (III)

Código fuente Python

```python
class Dato:
    """clase que almacena un determinado valor de tipo entero"""

    def __init__(self, valor: int, *args, **kwargs) -> None:
        """inicializa un objeto de la clase"""

        self.__valor = valor
        super().__init__(*args, **kwargs)

    def get_value(self) -> int:
        """devuelve el valor almacenado"""

        return self.__valor

    def set_value(self, valor: int) -> None:
        """modifica el valor almacenado"""

        self.__valor = valor

if __name__ == "__main__":
    import dis      # importa el desensamblador
    dis.dis(Dato)   # desensambla el código bytecode
```

Código bytecode

```
Disassembly of __init__:
          0 COPY_FREE_VARS       1

 4        2 RESUME               0

 7        4 LOAD_FAST            1 (valor)
          6 LOAD_FAST            0 (self)
          8 STORE_ATTR           0 (_Dato__valor)

 8       18 PUSH_NULL
         20 LOAD_GLOBAL          3 (NULL + super)
         32 PRECALL              0
         36 CALL                 0
         46 LOAD_ATTR            2 (__init__)
         56 LOAD_FAST            2 (args)
         58 BUILD_MAP            0
         60 LOAD_FAST            3 (kwargs)
         62 DICT_MERGE           1
         64 CALL_FUNCTION_EX     1
         66 POP_TOP
         68 LOAD_CONST           1 (None)
         70 RETURN_VALUE

Disassembly of get_value:
 10      0 RESUME                0

 13      2 LOAD_FAST             0 (self)
         4 LOAD_ATTR             0 (_Dato__valor)
        14 RETURN_VALUE

Disassembly of set_value:
 15      0 RESUME                0

 18      2 LOAD_FAST             1 (valor)
         4 LOAD_FAST             0 (self)
         6 STORE_ATTR            0 (_Dato__valor)
        16 LOAD_CONST            1 (None)
        18 RETURN_VALUE
```

Introducción al lenguaje de programación Python (IV)

Recursos disponibles en la web

- Web oficial de Python:
  `https://www.python.org/`
- Documentación oficial de Python:
  `https://docs.python.org/es/`
- Tutorial oficial de Python:
  `https://docs.python.org/es/3/tutorial/index.html`
- Manual de la biblioteca estándar:
  `https://docs.python.org/es/3/library/index.html`
- Referencia del lenguaje Python:
  `https://docs.python.org/es/3/reference/index.html`
- PEP 8 – Guía de Estilo para Código Python:
  `https://peps.python.org/pep-0008/`
- Guía de Estilo para Código Python de Google:
  `https://google.github.io/styleguide/pyguide.html`
- PEP 257 - Convenciones sobre los *DocStrings*:
  `https://peps.python.org/pep-0257/`
- Curso *"Introducción a la Programación con Python"* de Univ. Harvard:
  `https://cs50.harvard.edu/python/2022/`

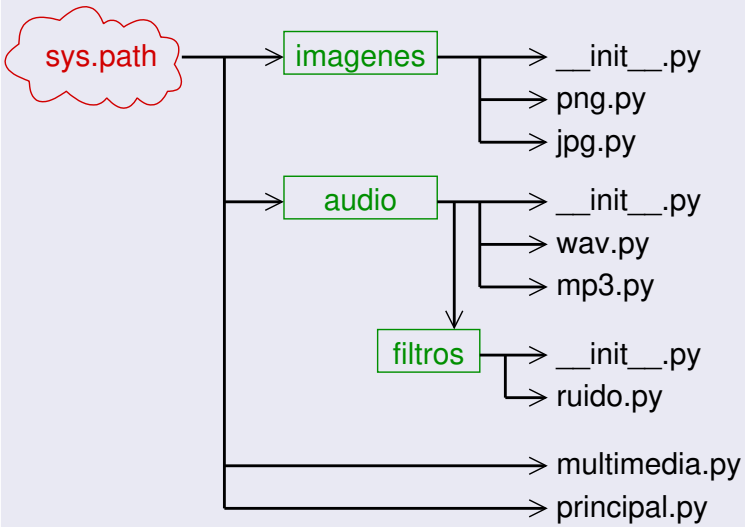
Módulos, paquetes y programas en Python

- Un **módulo** corresponde con un fichero que contiene código fuente en **Python**.
  - El nombre del fichero es el nombre del módulo, añadiendo la extensión **.py**.
- Un **módulo** puede *utilizar* (**importar**) otros módulos, para hacer uso de la funcionalidad, definiciones y características que ofrecen.
- La **biblioteca estándar** de Python proporciona una gran cantidad de módulos que facilitan el desarrollo de las diversas aplicaciones.
- Además, también existe la posibilidad de utilizar bibliotecas y módulos desarrollados por **terceras partes**.
- Un **programa** en Python se compone de uno o múltiples módulos, desarrollados por una o varias personas, incluyendo módulos proporcionados por la biblioteca estándar, así como por otras bibliotecas externas.
- El **módulo principal** contiene el código donde comienza la ejecución del programa.

- Cada módulo tiene su **espacio de nombres privado**, que contiene todas las definiciones realizadas dentro de cada módulo.
- Cuando un determinado módulo se ejecuta como **módulo principal**, la variable interna `__name__` toma el valor `"__main__"`.
  - Cuando se utiliza (importa) un determinado módulo, entonces la variable interna `__name__` toma el valor del nombre del módulo.

Módulos, paquetes y programas en Python

- Los **paquetes** permiten organizar lógicamente y agrupar módulos relacionados.
  - Un **paquete** integra múltiples módulos, relacionados entre sí, bajo una misma jerarquía de nombres separados por puntos.
  - Los **paquetes** permiten estructurar el *espacio de nombres* de módulos de Python, utilizando nombres de módulos y sub-módulos con puntos.
    - Por ejemplo, el nombre del módulo **A.B** representa al sub-módulo **B** en el paquete **A**.
- La jerarquía de nombres de paquetes separados por puntos permite acceder al fichero correspondiente al módulo especificado, según la jerarquía de carpetas, directorios y ficheros correspondientes.



Paquetes	Sub-módulos	Módulos
imagenes	png	imagenes.png
	jpg	imagenes.jpg
audio	wav	audio.wav
	mp3	audio.mp3
audio.filtros	ruido	audio.filtros.ruido
		multimedia
		principal

- `sys.path` es una variable que contiene el directorio del módulo principal, el directorio de trabajo, los directorios especificados en la variable de entorno `PYTHONPATH`, y los directorios del systema de python.

Intérprete Interactivo de Python

- En el intérprete interactivo de Python, es posible definir funciones, clases, variables, y ejecutar sentencias de Python.
- El comando `dir()` muestra todos los nombres accesibles desde el *espacio de nombres* actual.
- El comando `dir(`*nombre-de-módulo*`)` muestra todos los nombres accesibles desde el *espacio de nombres* del módulo especificado.
- El comando `help()` muestra la información de ayuda y documentación del módulo actual.
- El comando `help(`*item*`)` muestra la información de ayuda y documentación asociada al *item* especificado. *Item* puede ser un paquete, módulo, clase, función, tipo, variable, o cualquier entidad válida de python.
- Es útil para realizar pequeñas pruebas de código.

```
>>> import math
>>> dir()
[ ... ]
>>> dir(math)
[ ... ]
>>> help()
  [ ... ]
help> quit
>>> help(math)
  [ ... ]
>>>
```

```
>>> import math

>>> res = math.sin(math.pi/2) + math.cos(math.pi) + math.e**2
>>> print(res)

>>> factorial = 1
>>> for i in range(2, 7):
>>>     factorial *= i
>>> print(factorial)
>>>
```

Elementos de un Módulo en Python (I)

```python
"""
Este módulo presenta un ejemplo de los elementos
principales de un módulo Python
"""

__version__ = "2023.07.13"
__author__ = "John Doe"

from typing import Final, Optional, Any, Iterable
import math

# Esto es un comentario de línea
_CONSTANTES_GLOBALES: Final = 35  # comentario de línea


def calc_media(num1: float, num2: float, num3: float) -> float:
    """Calcula la media de los parámetros.

    Parámetros:
      - num1: valor numérico.
      - num2: valor numérico.
      - num3: valor numérico.
    Devuelve:
      - La media de num1, num2, num3
    """
    return (num1 + num2 + num3) / 3
```

Elementos de un Módulo en Python (II)

```python
class Dato:
    """clase que almacena un determinado valor de tipo entero"""

    def __init__(self, valor: int, *args, **kwargs) -> None:
        """Inicializa un objeto de la clase."""
        self.__valor = valor
        super().__init__(*args, **kwargs)

    def get_value(self) -> int:
        """Devuelve el valor almacenado."""
        return self.__valor

    def set_value(self, valor: int) -> None:
        """Modifica el valor almacenado."""
        self.__valor = valor


def main() -> None:
    dato = Dato(5)
    dato.set_value(7)
    print(dato.get_value())

    if (dato.get_value() > 3 and dato.get_value() < 9):
        print(calc_media(3.0, 5.0, 6.0))

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

PEP 8: Guía de Estilo para Código Python (I)

- Todos los módulos, todas las funciones y clases públicas del módulo, y todos los métodos públicos de una clase deben tener **Docstrings**.
  - Una *Docstring* es una *cadena de caracteres* literal que aparece como la primera sentencia en la definición de un módulo, una función, una clase, o un método de clase, entre `"""comillas dobles triples"""`, que puede contener múltiples líneas.

- `__version__` y `__author__` se definirán después de *docstring* y antes de las *importaciones*.
- Las **importaciones de módulos** se realizarán al principio del módulo, después del *docstring*.
  - *(1)* Módulos de la biblioteca estándar, *(2)* módulos externos, *(3)* módulos locales.
- Las **constantes** se definirán al principio del módulo, después de las *importaciones*.
- Las líneas tendrán un máximo de **79 caracteres** (opcionalmente un máximo de 99 caracteres).
- Se utilizarán **4 espacios** (sin tabuladores) por cada nivel de indentación.
  - Las **líneas de continuación**, dentro de paréntesis, corchetes y llaves, estarán verticalmente alineadas al primer elemento.
    - La línea se debe **romper** después de *coma*, o antes del *operador binario*.
- Las definiciones de **clases** y **funciones** estarán separadas por **dos líneas en blanco**.
- Las definiciones de **métodos** de clases estarán separadas por **una línea en blanco**.
- Los siguientes **operadores** estarán separados por un espacio delante y detrás (no = en args):

```
= += -= *= /= //= **= == < > != <= >= in is and or not
```

PEP 8: Guía de Estilo para Código Python (II)

Identificadores (nombres)

- **Paquetes**: palabras en minúsculas.
- **Módulos**: estilo **snake_case**, palabras en minúsculas, separados por _ si es necesario.
- **Tipos y Clases**: estilo **CamelCase**, cada palabra Capitalizada.
- **Excepciones**: estilo **CamelCaseError**, cada palabra Capitalizada, Error al final.
- **Constantes**: palabras en MAYÚSCULAS, separados por _ si es necesario.
- **Funciones**: estilo **snake_case**, palabras en minúsculas, separados por _ si es necesario.
- **Métodos**: estilo **snake_case**, palabras en minúsculas, separados por _ si es necesario.
- **Variables**: estilo **snake_case**, palabras en minúsculas, separados por _ si es necesario.
- En algunas situaciones, también es válido utilizar **mixedCase** para *funciones*, *métodos* y *variables*.
- A nivel de **módulo**, los identificadores **privados** comienzan por _.
- A nivel de **clase**, los identificadores **privados** comienzan por __.
- A nivel de **clase**, los identificadores **protegidos** comienzan por _.

- Los nombres que comienzan y terminan por __ están **reservados** para uso interno de Python, por ejemplo `__name__`, `__init__`, etc.

Importación de Módulos

- Un módulo debe **importar** a otro módulo cuando desea hacer uso de las funcionalidades proporcionadas por ese módulo.
- Hay diversos mecanismos para importar módulos:
  - **Importación** del módulo. Para utilizar cualquier símbolo definido en el módulo importado, el símbolo se debe cualificar con el nombre del módulo. Por ejemplo:

```
import math
res = math.sin(math.pi/2) + math.cos(math.pi) + math.e**2
```

  - **Importación** del módulo con **alias**. Para utilizar cualquier símbolo definido en el módulo importado con alias, el símbolo se debe cualificar con el nombre del alias del módulo. Por ejemplo:

```
import math as mm
res = mm.sin(mm.pi/2) + mm.cos(mm.pi) + mm.e**2
```

  - **Importación explícita de símbolos** del módulo. Los símbolos importados explícitamente se pueden utilizar directamente, sin cualificación. Por ejemplo:

```
from math import sin, cos, pi, e
res = sin(pi/2) + cos(pi) + e**2
```

  - **Importación explícita de símbolos** con **alias**. Los símbolos importados explícitamente con alias, se pueden utilizar directamente con su alias. Por ejemplo:

```
from math import sin as SIN, pi as PI
res = SIN(PI/2)
```

Elementos del lenguaje: Tipos

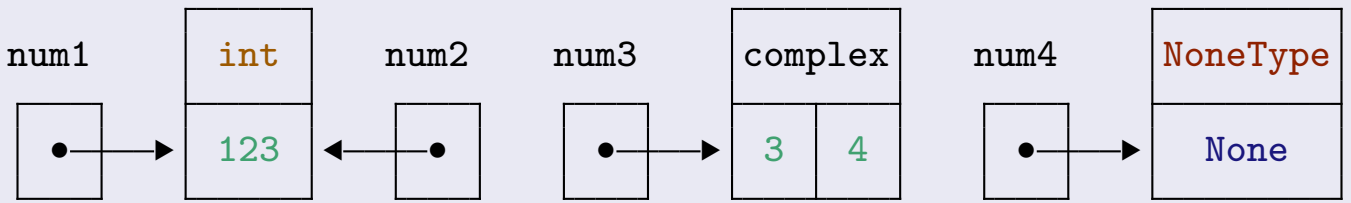
Tipos

- Existen los siguientes tipos predefinidos:
  - **bool** para valores lógicos (booleanos): `False`, `True`.
  - **int** para *números enteros*: `1`, `2`, `-35`, `1234`.
  - **float** para *números reales*: `3.1415`, `2.7182`, `12.345e+15`, `1.23e-5`.
  - **complex** para *números complejos*: `34+56j`, `-3-57j`.
  - **range** para *intervalos* de números enteros: `range(10)`, `range(1, 10)`, `range(1, 10, 2)`.
  - **str** para *cadenas de caracteres*: `"esto es una cadena de caracteres"`.
  - **tuple** para *tuplas* de elementos: `(1, 2, 3)`.
  - **list** para *listas* de elementos: `[1, 2, 3]`.
  - **set** para *conjuntos* de elementos: `{1, 2, 3}`.
  - **dict** para *diccionarios* de pares de elementos *clave:valor*: `{"x":1, "y":2, "z":3}`.

Elementos del lenguaje: Variables

Variables

- Las variables almacenan una **referencia a un objeto** que contiene el valor referenciado por la variable.
  - La *asignación* (=) **asigna la referencia al objeto** (al mismo objeto compartido).
- Las variables **no tienen** ningún **tipo** asociado, el **tipo** está **asociado** a los objetos que contienen los **valores**.
- Las variables son **polimórficas**, es decir, en un determinado momento pueden referenciar a un objeto que contiene un valor de un tipo, y en otro determinado momento pueden referenciar a otro objeto con un valor de un tipo diferente.
  - Se recomienda que las variables referencien a valores del mismo tipo, relacionados con la información que representa cada variable.
- Las variables **no se declaran**, las variables se **crean** automáticamente la primera vez que se utilizan, y se **destruyen** de automáticamente cuando no son necesarias.




```
num1 = 123
num2 = num1
num3 = 3+4j
num4 = None
```

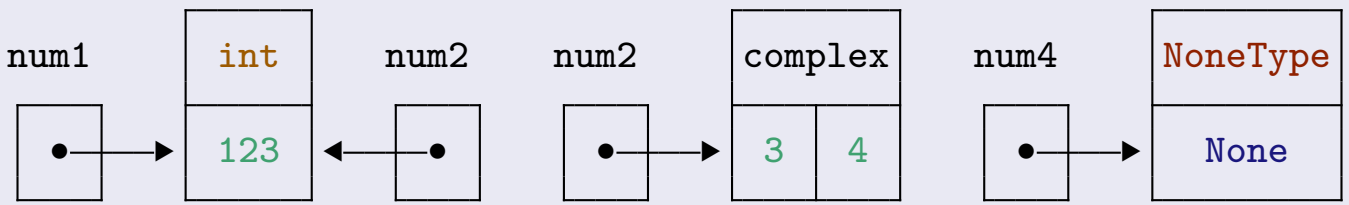
Elementos del lenguaje: Valores

Valores

- El valor **None** indica que una variable no referencia a ningún objeto.
- Los valores de los objetos de los tipos **bool**, **int**, **float**, **complex**, **str** y **tuple** son **inmutables**, es decir, no se puede cambiar el valor de los objetos, pero si se puede hacer que la variable referencie a otro objeto distinto con otro valor.
- Los valores se **crean** automáticamente la primera vez que se utilizan, y se **destruyen** de forma automática cuando ya no son necesarios.

Constantes simbólicas

- Las **constantes** simbólicas se definen como *variables*, al principio del módulo, con todas las **letras del nombre en mayúsculas**.
- Utilizaremos la anotación de tipo **Final** del módulo **typing** para indicar que un símbolo es constante, y no se debe modificar.



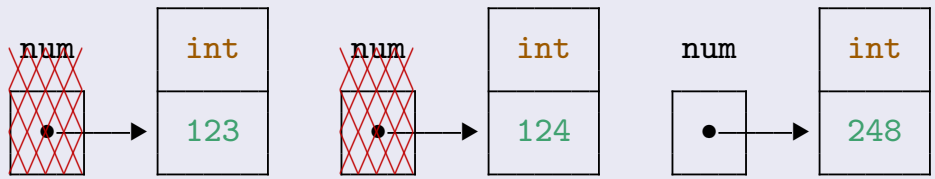

```
MAX: Final = 9
num1 = 123
num2 = num1
num2 = 3+4j
num4 = None
```

Elementos del lenguaje: Asignación

- Las variables almacenan una **referencia a un objeto** que contiene el valor referenciado por la variable.
  - La *asignación* **asigna** a una variable la **referencia a un objeto**, resultado de evaluar una **expresión**. La referencia anterior se pierde.

Operador	Descripción	Ejemplo	Equivalente a
`=`	Asignación	`num = expr`	
`+=`	Asignación de incremento.	`num += expr`	`num = num + ( expr )`
`-=`	Asignación de decremento.	`num -= expr`	`num = num - ( expr )`
`*=`	Asignación de multiplicación.	`num *= expr`	`num = num * ( expr )`
`/=`	Asignación de división real.	`num /= expr`	`num = num / ( expr )`
`//=`	Asignación de división entera.	`num //= expr`	`num = num // ( expr )`
`%=`	Asignación de modulo.	`num %= expr`	`num = num % ( expr )`
`**=`	Asignación de potencia.	`num **= expr`	`num = num ** ( expr )`

```
num = (120 + 3)    num += 1          num *= 2
```



Elementos del lenguaje. Expresiones

Expresiones

- Una **expresión** es una combinación de operadores aplicados sobre operandos.
  - Los operandos pueden ser valores literales, variables, e invocaciones a funciones y métodos.
  - La **precedencia** de los operadores determina el **orden de evaluación** de las subexpresiones. A igualdad de precedencia, se aplica **asociatividad**.

```
res = 2 + 4 * -3**2 / 2      # valor = -16.0
```

Orden de precedencia y asociatividad de operadores

Operador	Descripción	Asoc
`(...) [...] {...}`	Paréntesis, tuplas, listas, conjuntos, diccionarios	No
`x[i], x[i:j], x(args), x.attr`	Indexación, segmentación, invocación, atributo	Izq
`x ** z`	Potencia	Dch
`+x, -x`	Positivo, negativo	Dch
`x * z, x / z, x // z, x % z`	Multiplicación, división real, división entera, resto	Izq
`x + z, x - z`	Suma, resta	Izq
`x < z, x <= z, x > z, x >= z, x != z, x == z`	Comparaciones	Sí
`x in z, x not in z, x is z, x is not z`	Comparaciones	No
`not x`	Negación lógica	Dch
`x and z`	Conjunción (Y) lógica	Izq
`x or z`	Disyunción (O) lógica	Izq
`- if - else -`	Expresión condicional	No

Operadores

Operador	Significado
`( expr )`	Paréntesis, modifica el orden de evaluación.
`- X`	Menos unario, negativo, cambio de signo.
`X ** Z`	Potencia aritmética.
`X * Z`	Multiplicación aritmética.
`X / Z`	División aritmética real.
`X // Z`	División aritmética entera.
`X % Z`	Resto de la división aritmética entera (módulo).
`X + Z`	Suma aritmética.
`X - Z`	Resta aritmética.
`X > Z`	Mayor, `True` si `X` mayor que `Z` , `False` en otro caso.
`X >= Z`	Mayor o igual, `True` si `X` mayor o igual que `Z` , `False` en otro caso.
`X < Z`	Menor, `True` si `X` menor que `Z` , `False` en otro caso.
`X <= Z`	Menor o igual, `True` si `X` menor o igual que `Z` , `False` en otro caso.
`X == Z`	Igualdad, `True` si `X` igual a `Z` , `False` en otro caso.
`X != Z`	Distinto, `True` si `X` distinto a `Z` , `False` en otro caso.
`X is Z`	Identidad, `True` si `X` idéntico a `Z` , `False` en otro caso.
`X is not Z`	No Identidad, `True` si `X` no idéntico a `Z` , `False` en otro caso.
`X in Z`	Pertenece, `True` si `X` pertenece a `Z` , `False` en otro caso.
`X not in Z`	No Pertenece, `True` si `X` no pertenece a `Z` , `False` en otro caso.
`not X`	Negación lógica (véase tabla de verdad)
`X and Z`	Conjunción (Y)/(AND) lógica (véase tabla de verdad).
`X or Z`	Disyunción (O)/(OR) lógica (véase tabla de verdad).

Elementos del lenguaje. Comparaciones

Comparación de Identidad

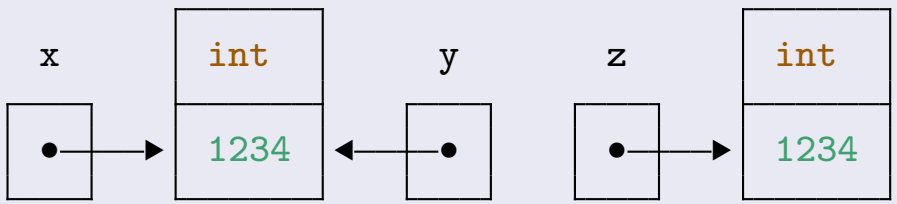
- El operador **is** devuelve `True` si las dos variables referencian al mismo objeto, y devuelve `False` en otro caso.
- El operador **is not** devuelve `True` si las dos variables referencian a objetos distintos, y devuelve `False` en otro caso.

```
>>> x = 1234

>>> y = x

>>> z = 1230 + 4
```

```
>>> x is y
True
>>> x is z
False
>>> y is z
False
>>> z is None
False
```

```
>>> x is not y
False
>>> x is not z
True
>>> y is not z
True
>>> z is not None
True
```

```
>>> id(x)
140062062210128
>>> id(y)
140062062210128
>>> id(z)
140062062210480
```



Elementos del lenguaje. Comparaciones

Comparación de Igualdad (equivalencia)

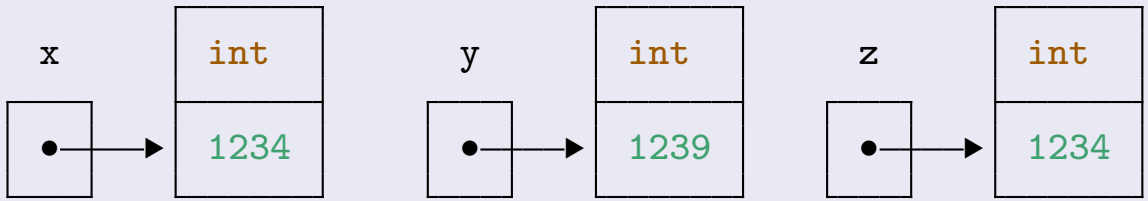
- El operador == devuelve `True` si los dos objetos referenciados tienen el mismo valor (son equivalentes), y devuelve `False` en otro caso.
- El operador != devuelve `True` si los dos objetos referenciados tienen valores distintos (no son equivalentes), y devuelve `False` en otro caso.

```
>>> x = 1234

>>> y = x + 5

>>> z = 1230 + 4
```

```
>>> x == y
False
>>> x == z
True
>>> y == z
False
>>> z == 1234
True
```

```
>>> x != y
True
>>> x != z
False
>>> y != z
True
>>> z != 1234
False
```

```
>>> x
1234
>>> y
1239
>>> z
1234
```



Elementos del lenguaje. Comparaciones

Comparación de Igualdad (equivalencia)

- El operador == devuelve `True` si los dos objetos referenciados tienen el mismo valor (son equivalentes), y devuelve `False` en otro caso.
- El operador != devuelve `True` si los dos objetos referenciados tienen valores distintos (no son equivalentes), y devuelve `False` en otro caso.

- Los números **reales** (`float`) tienen una representación **aproximada**, por lo que la comparación de igualdad (equivalencia) de números reales puede producir resultados inesperados.
  - Por ello, se recomienda comparar la equivalencia de números reales con la función `isclose(a, b)` del módulo `math`.

```
>>> x = ((3.0 * 0.1) / 3.0)

>>> y = x + 5

>>> z = (3.0 * (0.1 / 3.0))
```

```
>>> x == y
False
>>> x == z
False
>>> y == z
False
```

```
>>> math.isclose(x, y)
False
>>> math.isclose(x, z)
True
>>> math.isclose(y, z)
False
```

```
>>> x
0.10000000000000002
>>> y
5.1
>>> z
0.1
```

Expresión Condicional (operador condicional ternario)

```
op1 if cond else op3
```

- La expresión condicional (operador condicional ternario) `op1 if cond else op3` primero evalúa la condición (`cond`), si es verdadera, entonces el resultado de la expresión es el valor del primer operando (`op1`). En otro caso, el resultado es el valor del tercer operando (`op3`).
  - Se debe utilizar de forma simple, ya que puede dar lugar a expresiones complejas.

```
mayor = x if x > y else y    # Asigna el valor mayor de X e Y
menor = x if x < y else y    # Asigna el valor menor de X e Y
```

Elementos del lenguaje. Operadores lógicos

Tabla de verdad de los operadores lógicos

x	y	x and y	x or y	not x
False	False	False	False	True
False	True	False	True	True
True	False	False	True	False
True	True	True	True	False

- Los operadores lógicos (**and or**) se evalúan en **CORTOCIRCUITO**.
  - Cuando ya se conoce el resultado de la operación lógica trás la evaluación del primer operando, entonces el **segundo operando NO se evalúa**.

Operadores relacionales encadenados

- Los operadores relacionales (< <= > >= != ==) se pueden encadenar (equivalente a `and`), y se evalúan en cortocircuito:
  - Se debe utilizar de forma simple, ya que puede dar lugar a expresiones complejas.

```
(x <= y < z)     # equivalente a: (x <= y) and (y < z)
```

Elementos del lenguaje. Conversiones de Tipo

- En las expresiones, cuando se evalúa una operación aplicada sobre **operandos de tipos distintos**, se realiza una conversión de tipo **automática** al tipo simple más **amplio** de los operandos.
  - bool ▶ int ▶ float ▶ complex

- En la evaluación de expresiones, a veces nos puede interesar que un determinado operando o expresión sea **evaluado** a un valor con un **tipo distinto al suyo propio**. En estos casos, se realiza una **conversión explícita** de tipos (*casting*), precediendo el valor o expresión entre paréntesis por el símbolo del tipo al que queremos convertir:

```
a = int(True)          # produce el entero 1
b = bool(0)            # produce el valor lógico False

i = float(2)           # produce el real 2.0
j = int(3.6)           # produce el entero 3 (pierde decimales)

k = complex(2)         # produce el complejo 2+0j
k1 = k.real            # produce parte real
k2 = k.imag            # produce parte imaginaria
```

- `bool(valor)`: devuelve `False` si `valor` es alguno de `None`, `False`, `0`, `0.0`, `0j`, o *string*, *tupla*, *lista*, *conjunto* o *diccionario* vacía, y `True` en otro caso.

Elementos del lenguaje. Secuencia de Sentencias

Secuencia de Sentencias

- Cada sentencia debe aparecer en una línea independiente, indentada adecuadamente según el **nivel de anidamiento** en que se encuentre.
  - **4 espacios por cada nivel de anidamiento**.
- Cada sentencia termina en el **salto de línea** (**no** se utiliza el símbolo punto-y-coma como terminador de sentencias).
- Las **líneas de continuación**, dentro de paréntesis, corchetes y llaves, estarán verticalmente alineadas al primer elemento.
  - La línea se debe **romper** después de *coma*, o antes del *operador binario*.

```
if ((x < (minimo + maximo))
    and (z > maximo)):
    valor = funcion(arg1,
                    arg2,
                    arg3)
```

- La sentencia **pass** no hace nada, y se utiliza para indicar que en una determinada situación **no se debe hacer nada**.
  - Otras veces, también resulta de utilidad cuando todavía no hemos terminado el desarrollo de una parte del código.

Elementos del lenguaje. Sentencia de Selección `if`

Sentencia de Selección `if-elif-else`

- Ejecuta el bloque de sentencias correspondiente a la primera **expresión lógica de control** que sea evaluada a `True`, según el orden especificado.
- Si ninguna expresión lógica de control se evalúa a `True`, entonces se ejecuta el bloque de sentencias correspondiente a `else`, si existe.

Selección simple

```
if exp_lógica:
    sentencias
```

Selección doble

```
if exp_lógica:
    sentencias
else:
    sentencias
```

Selección múltiple

```
if exp_lógica:
    sentencias
elif exp_lógica:
    sentencias
...
elif exp_lógica:
    sentencias
else:
    sentencias
```

- El símbolo **dos-puntos** es importante.
- El bloque de sentencias debe tener una **intentación** de 4 espacios por cada nivel de anidamiento.
- La expresión lógica debe estar entre **paréntesis** para poder romper la línea.

Elementos del lenguaje. Sentencia de Selección `match`

Sentencia de Selección `match-case`

- Ejecuta el bloque de sentencias correspondiente, según el valor de una determinada **expresión**, comparada con diferentes **valores** especificados en cláusulas `case`, según el orden especificado.
- Puede dar lugar a situaciones complejas, sólo la utilizaremos en su formato más simple. No deben aparecer variables en las cláusulas `case`, sólo **valores**.

Selección `match`

```
match expresión:
    case valor:
        sentencias
    case valor | valor:
        sentencias
    ...
    case _: # al FINAL
        sentencias
```

Selección `match`

```
match contador:
    case 1:
        contador += 2
    case 9 | 10:
        contador = 0
    case _:
        contador += 1
```

Selección `match`

```
match nombre:
    case "ana":
        resultado = 1
    case "luis" | "eva":
        resultado = 2
    case _:
        resultado = 3
```

- El símbolo **dos-puntos** es importante.
- La claúsula `case` y el bloque de sentencias deben tener una **intentación** de 4 espacios por cada nivel de anidamiento.
- La expresión debe estar entre **paréntesis** para poder romper la línea.

Elementos del lenguaje. Sentencias de Iteración

Sentencias de Iteración

- El bucle **for** permite iterar, el bloque de sentencias, sobre **todos** los elementos del **iterable**. En cada iteración, la variable toma el siguiente valor del iterable.
- El bucle **while** permite iterar, el bloque de sentencias, mientras el valor de la **expresión lógica de control** es `True`.
- No existe sentencia **do-while**.

Iteración For

```
for variable in iterable:
    sentencias
```

Iteración While

```
while exp_lógica:
    sentencias
```

- El símbolo **dos-puntos** es importante.
- El bloque de sentencias debe tener una **intentación** de 4 espacios por cada nivel de anidamiento.
- La expresión lógica debe estar entre **paréntesis** para poder romper la línea.

- **Intervalos**, **cadenas de caracteres**, **tuplas**, **listas**, **conjuntos** y **diccionarios** son **iterables**.

Elementos del lenguaje. Intervalos iterables

Intervalos

- **range** es un iterable que permite generar secuencias de números, dentro de un determinado intervalo especificado por (inicio, límite e incremento).
- `range(limite)`: genera secuencia iterable de números desde `0` hasta `limite`, sin incluir `limite`.
- `range(inicio, limite)`: genera secuencia iterable de números desde `inicio` hasta `limite`, sin incluir `limite`.
- `range(inicio, limite, incr)`: genera secuencia iterable de números desde `inicio` hasta `limite`, sin incluir `limite`, incrementando en cada paso el valor de `incr`. El incremento puede ser negativo.

```
for i in range(5):
    print(i)              # 0 1 2 3 4

for i in range(1, 5):
    print(i)              # 1 2 3 4
```

```
for i in range(1, 5, 2):
    print(i)              # 1 3

for i in range(4, 0, -1):
    print(i)              # 4 3 2 1
```

Entrada y salida de datos

Salida de datos

- La función `print(*args, sep=' ', end='\n')` muestra en pantalla el valor de los argumentos. El espacio es el separador de argumentos por omisión. Al final, se muestra por omisión un salto de línea.

```
print("Resultado:", 1, 2, 3, (1+2+3)/3)             # muestra: Resultado: 1 2 3 2.0
print("Resultado:", 1, 2, 3, (1+2+3)/3, sep=", ")   # muestra: Resultado:, 1, 2, 3, 2.0
print("Resultado:", 1, end="")       # muestra: Resultado: 1 (sin salto de línea final)
```

Entrada de datos

- La función `input(mensaje='')` muestra en pantalla el valor del *mensaje de entrada*. Lee una línea de caracteres (de tipo **str**) de teclado y la devuelve (sin el salto de línea).
  - Si se desea leer un número (`int`, `float`, `complex`), la cadena de caracteres leída se debe convertir al tipo adecuado.

```
valor = input("Introduce frase: ")
valor = int(input("Introduce número entero: "))
valor = float(input("Introduce número real: "))
valor = complex(input("Introduce número complejo: "))
```

Entrada y salida de datos. Ejemplo 1

Programa que lee de teclado un número entero, y calcula el *factorial* de ese número, o muestra un mensaje de error si no es posible.

```
"""
Módulo principal que calcula el factorial de un número entero
"""

def main() -> None:
    """Lee un número de teclado y muestra factorial"""

    num = int(input("Introduce un número entero: "))
    if (num < 0):
        print("Error, número no válido")
    else:
        factorial = 1
        for i in range(2, num+1):
            factorial *= i
        print("El factorial es:", factorial)


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

Entrada y salida de datos. Ejemplo 2

Programa que lee de teclado dos números enteros, y calcula el *máximo común divisor* de ambos números, o muestra un mensaje de error si no es posible.

```
"""
Módulo principal que calcula el MCD de dos números enteros
"""

def main() -> None:
    """Lee dos números de teclado y muestra MCD de ambos"""

    num1 = int(input("Introduce un número entero: "))
    num2 = int(input("Introduce un número entero: "))
    if (num1 <= 0 or num2 <= 0):
        print("Error, números no válidos")
    else:
        while num1 != num2:
            if num1 > num2:
                num1 -= num2
            else:
                num2 -= num1
        print("El máximo común divisor es:", num1)


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

Definición e invocación a funciones

Definición e invocación de funciones

- Una **función** define un **bloque de código independiente**, que resuelve un determinado subproblema de forma parametrizada, y puede ser ejecutado (invocado) múltiples veces, aplicado a diferentes argumentos (valores).
- Todos los subprogramas son **funciones**, de tal forma que si no devuelven ningún valor explícitamente, entonces devuelven el valor `None` implícitamente.
- Se realiza paso de parámetros **por valor**, de la **referencia al objeto**.
- Utilizaremos **anotaciones de tipo** para los parámetros y el valor devuelto.
- Todas las **variables** utilizadas dentro de una función son **locales**.

Efectos de las funciones

- Una función **sólo puede tener los efectos** que sean especificados.

Anotaciones de tipo

- En la definición de una función, tanto los parámetros como el valor devuelto se pueden especificar con **anotaciones de tipo**, *pre/definidos*, o tipos especificados en el módulo **typing**: **Optional**, **Any**, **Union** (|), **Iterable**.

Definición e invocación a funciones. Ejemplo

Ejemplos:

1. Programa que muestra la media de dos números leídos de teclado.
2. Programa que lee de teclado un número entero, y calcula el *factorial* de ese número, o muestra un mensaje de error si no es posible.
3. Programa que lee de teclado dos números enteros, y calcula el *máximo común divisor* de ambos números, o muestra un mensaje de error si no es posible.

```python
def calc_media(n1: float, n2: float) -> float:
    """Calcula la media de los parámetros"""
    return (n1 + n2) / 2

def main() -> None:
    """Lee dos números de teclado y muestra la media"""

    n1 = float(input("Introduce número entero: "))
    n2 = float(input("Introduce número entero: "))
    res = calc_media(n1, n2)
    print("La media es:", res)
```

```python
if __name__ == "__main__":
    main()
```

Definición e invocación a funciones. Ejemplo

```
"""
Módulo principal que calcula el factorial de un número entero
"""
from typing import Optional

def factorial(num: int) -> Optional[int]:
    """Calcula el factorial"""
    res: Optional[int] = None
    if (num >= 0):
      res = 1
      for i in range(2, num+1):
          res *= i
    return res

def main() -> None:
    """Lee un número de teclado y muestra factorial"""
    num = int(input("Introduce número entero: "))
    res = factorial(num)
    if res is None:
        print("Error, número no válido")
    else:
        print("El factorial es:", res)

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

Definición e invocación a funciones. Ejemplo

```
"""
Módulo principal que calcula el MCD de dos números enteros
"""
from typing import Optional

def mcd(num1: int, num2: int) -> Optional[int]:
    """Calcula el MCD de dos números recibidos como parámetros"""
    res: Optional[int] = None
    if (num1 > 0 and num2 > 0):
        while num1 != num2:
            if num1 > num2:
                num1 -= num2
            else:
                num2 -= num1
        res = num1
    return res

def main() -> None:
    """Lee dos números de teclado y muestra MCD de ambos"""
    num1 = int(input("Introduce número entero: "))
    num2 = int(input("Introduce número entero: "))
    res = mcd(num1, num2)
    if res is None:
        print("Error, números no válidos")
    else:
        print("El máximo común divisor es:", res)
```

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

Definición e invocación a funciones. Parámetros y Argumentos

Parámetros con valores por omisión (defecto)

- Los parámetros pueden especificar el valor que deben tomar por omisión (defecto), en caso de que no reciban valores durante la invocación.
  - Deben aparecer al final de la lista de parámetros.

```
def mostrar_rango(inicio: int, limite: int,
                  prefijo: str = "[ ", separador: str = ", ", sufijo: str = " ]\n") -> None:
    pass  # definición en la siguiente página
```

Argumentos en la invocación a funciones

- Durante la invocación, los parámetros pueden recibir los valores según la **posición**, o pueden recibir valores según las **palabras clave** (*kwargs*).
  - Si algún parámetro no recibe valores en la invocación a la función, entonces tomará el valor por omisión (defecto), si está definido, en otro caso error.

```
mostrar_rango(0, 5, "<<< ", " @ ", " >>>\n")
mostrar_rango(0, 5)
mostrar_rango(0, 5, separador=" @ ")       # Argumentos con palabras clave (kwargs)
mostrar_rango(0, 5, prefijo="<<< ")        # Argumentos con nombre (kwargs)
mostrar_rango(0, 5, sufijo=" >>>\n")
mostrar_rango(0, 5, prefijo="<<< ", separador=" @ ", sufijo=" >>>\n")
mostrar_rango(0, 5, separador=" @ ", prefijo="<<< ", sufijo=" >>>\n")
mostrar_rango(inicio=0, limite=5, separador=" @ ", prefijo="<<< ", sufijo=" >>>\n")
mostrar_rango(separador=" @ ", prefijo="<<< ", sufijo=" >>>\n", inicio=0, limite=5)
```

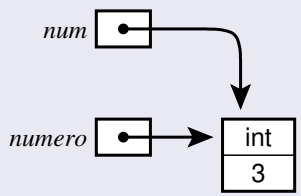
Definición e invocación a funciones. Parámetros y Argumentos

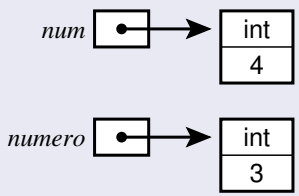
```
def mostrar_rango(inicio: int, limite: int,
                  prefijo: str = "[ ", separador: str = ", ", sufijo: str = " ]\n") -> None:
    """Muestra los números en el rango [inicio, limite)
    Parámetros
    ----------
    inicio: inicio del rango
    limite: limite superior del rango
    prefijo: prefijo del rango
    separador: separador entre valores
    sufijo: sufijo del rango
    """
    print(prefijo, end="")
    if inicio < limite:
        print(inicio, end="")
        for valor in range(inicio+1, limite):
            print(separador, end="")
            print(valor, end="")
    print(sufijo, end="")

def main() -> None:
    mostrar_rango(0, 5, "<<< ", " @ ", " >>>\n")                                  # <<< 0 @ 1 @ 2 @ 3 @ 4 >>>
    mostrar_rango(0, 5)                                                           # [ 0, 1, 2, 3, 4 ]
    mostrar_rango(0, 5, separador=" @ ")                                          # [ 0 @ 1 @ 2 @ 3 @ 4 ]
    mostrar_rango(0, 5, prefijo="<<< ")                                           # <<< 0, 1, 2, 3, 4 ]
    mostrar_rango(0, 5, sufijo=" >>>\n")                                          # [ 0, 1, 2, 3, 4 >>>
    mostrar_rango(0, 5, prefijo="<<< ", separador=" @ ", sufijo=" >>>\n") # <<< 0 @ 1 @ 2 @ 3 @ 4 >>>
    mostrar_rango(0, 5, separador=" @ ", prefijo="<<< ", sufijo=" >>>\n") # <<< 0 @ 1 @ 2 @ 3 @ 4 >>>
    mostrar_rango(separador=" @ ", prefijo="<<< ", sufijo=" >>>\n", inicio=0, limite=5) # <<< 0 @ 1 @ 2
if __name__ == "__main__":
    main()
```

Paso de parámetros por valor de la referencia al objeto

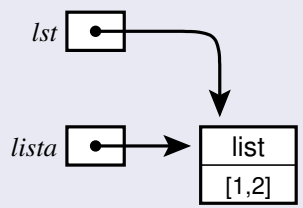
- El paso de parámetros a funciones se realiza mediante **paso por valor de la referencia** al objeto.
  - Dentro de la función se puede acceder al valor del objeto a través de la referencia.
  - Dentro de la función se puede modificar el valor del objeto, **si es mutable**, a través de la referencia.
  - Dentro de la función, la modificación de la referencia recibida como parámetro, para que referencie a otro objeto, **no modifica el argumento en la invocación**, y el nuevo objeto referenciado no será visto en el exterior de la función.
  - En prácticas, controles y exámenes, las funciones y métodos **no deben modificar** los valores de los parámetros que reciben, salvo que el enunciado especifique lo contrario.

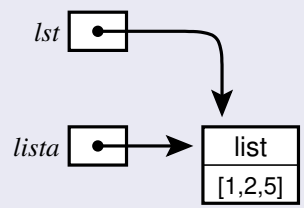
```python
def incrementar(num: int) -> None:
    num = num + 1   # num += 1

def main() -> None:
    numero = 3
    incrementar(numero)
    print(numero)   # 3
```






```python
def anyadir(lst: list[int]) -> None:
    lst.append(5)

def main() -> None:
    lista = [1, 2]
    anyadir(lista)
    print(lista)   # [1, 2, 5]
```





Definición e invocación a funciones. Anotaciones de tipo (I)

- En la definición de una función, tanto los parámetros como el valor devuelto se pueden especificar con **anotaciones de tipo**, *pre/definidos*, o tipos especificados en el módulo **typing**: **Optional**, **Any**, **Union** (|), **Iterable**.
- Las anotaciones de tipo son **opcionales**, el intérprete de Python no las comprueba.
- Sin embargo, los analizadores de código (**Pylint** y **MyPy**) pueden realizar un análisis más detallado y mostrar algunas posibles situaciones de error.

```
def mostrar_rango(inicio: int, limite: int,
                  prefijo: str = "[ ", separador: str = ", ", sufijo: str = " ]\n") -> None:
    pass # definición en la página anterior

def main() -> None:
    mostrar_rango("<<< ", " @ ", " >>>\n", 0, 5)

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

Analisis proporcionado por el asistente

```
Line 5 : Argument 1 to "mostrar_rango" has incompatible type "str"; expected "int"  [arg-type]
Line 5 : Argument 2 to "mostrar_rango" has incompatible type "str"; expected "int"  [arg-type]
Line 5 : Argument 4 to "mostrar_rango" has incompatible type "int"; expected "str"  [arg-type]
Line 5 : Argument 5 to "mostrar_rango" has incompatible type "int"; expected "str"  [arg-type]
```

Definición e invocación a funciones. Anotaciones de tipo (II)

- En la definición de una función, tanto los parámetros como el valor devuelto se pueden especificar con **anotaciones de tipo**, *pre/definidos*, o tipos especificados en el módulo **typing**: **Optional**, **Union** (|), **Iterable**.

```
from typing import Final, Optional, Iterable

CONSTANTE: Final = 123

def funcion(a, b: tuple[int, float], c: list[int], d: set[float], e: dict[str, int],
            f: Optional[int], g: (int|float), h: Iterable[int]) -> None:
    print(a, b, c, d, e, f, g, h)

def main() -> None:
    funcion("hola", (23, 3.14), [34], {3.14}, {"x": 45}, None, 3.14, range(10))
    funcion((23, 3.14), [34], {3.14}, {"x": 45}, None, 3.14, range(10), "hola")

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

Analisis proporcionado por el asistente

```
Line 11 : Argument 2 to "funcion" has incompatible type "List[int]"; expected "Tuple[int, float]"  [arg-type]
Line 11 : Argument 3 to "funcion" has incompatible type "Set[float]"; expected "List[int]"  [arg-type]
Line 11 : Argument 4 to "funcion" has incompatible type "Dict[str, int]"; expected "Set[float]"  [arg-type]
Line 11 : Argument 5 to "funcion" has incompatible type "None"; expected "Dict[str, int]"  [arg-type]
Line 11 : Argument 6 to "funcion" has incompatible type "float"; expected "Optional[int]"  [arg-type]
Line 11 : Argument 7 to "funcion" has incompatible type "range"; expected "Union[int, float]"  [arg-type]
Line 11 : Argument 8 to "funcion" has incompatible type "str"; expected "Iterable[int]"  [arg-type]
```

Anotaciones de tipo en funciones y variables

- Se recomienda especificar **anotaciones de tipo** en aquellas situaciones en las que el analizador de código no es capaz de deducir el tipo por sí mismo.
  - En los **parámetros** de funciones y en el tipo del valor **devuelto** por la función.
  - En la **primera asignación** a una variable, cuando el valor asignado **no** permite deducir el *tipo completo* de la variable.

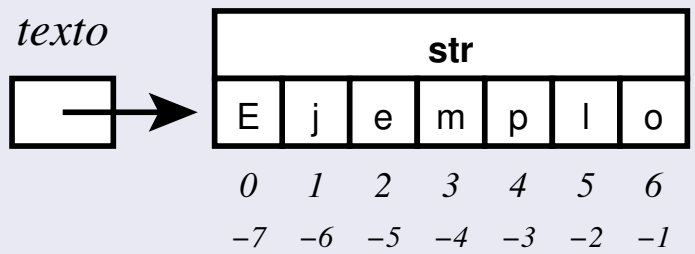
```
def funcion(num1: int) -> Optional[float]: # Es necesario anotar el tipo de parámetros y tipo del valor devuelto
    cnt = 0                                # El valor cero permite deducir que el tipo de cnt es int

    resultado: Optional[float] = None      # El valor None no permite deducir el tipo completo de resultado
    if num1 > 0:
        resultado = num1 / 2

    lista1 = [1, 2, 3]                     # El valor [1,2,3] permite deducir que el tipo de lista1 es list[int]

    lista2: list[str] = list()             # El valor list() no permite deducir el tipo completo de lista2
    lista2.append("error")

    conjunto1 = { "hola", "adios" }        # El valor permite deducir que el tipo de conjunto1 es set[str]

    conjunto2: set[int] = set()            # El valor set() no permite deducir el tipo completo de conjunto2
    conjunto2.add(3)

    dicc1 = {"a":1, "b":2}                 # El valor permite deducir que el tipo de dicc1 es dict[str,int]

    dicc2: dict[str,int] = dict()          # El valor dict() no permite deducir el tipo completo de dicc2
    dicc2["c"] = 3

    return resultado
```

Cadenas de caracteres

- El tipo **str** permite representar y manipular las cadenas de caracteres (**string**).
- Los objetos de tipo **str** son **INMUTABLES**, por lo que una vez creado, no es posible modificarlo. No obstante, sí se puede modificar la variable que referencia al objeto.
- Las **cadenas de caracteres** literales se representan como secuencias de caracteres Unicode entre comillas dobles (también es válido entre comillas simples, o triples comillas).

```
texto = "Ejemplo"
texto = "Ejemplo de cadena de caracteres"
texto = 'Otro ejemplo de cadena de caracteres'
texto = """Otro ejemplo de cadena de caracteres"""   # multi-línea
texto = '''Otro ejemplo de cadena de caracteres'''   # multi-línea
```

```
texto = input("Introduce texto: ")
print(texto)
```



Cadenas de caracteres con formato (I)

Cadenas de caracteres con formato

- Las *cadenas de caracteres con formato* (**f-strings**) permiten construir cadenas de caracteres, utilizando valores de otras variables, aplicando el formato especificado (*anchura de campo*, *justificación*, *completado con ceros a la izquierda*, y precisión). Por ejemplo:

```
nombre = "Juan Luis"
edad = 20
persona = f"Nombre: {nombre}; Edad: {edad}"
print(persona)                                  # Nombre: Juan Luis; Edad: 20
print(f"Nombre: {nombre}; Edad: {edad}")       # Nombre: Juan Luis; Edad: 20
```

Anchura de campo y justificación

```
print(f"Nombre: [{nombre:12}]; Edad: [{edad:4}]")    # Nombre: [Juan Luis   ]; Edad: [  20]
print(f"Nombre: [{nombre:>12}]; Edad: [{edad:<4}]") # Nombre: [   Juan Luis]; Edad: [20  ]
```

Anchura de campo variable

```
nombre = "Juan Luis"
anchura_de_campo = 12
print(f"Nombre: [{nombre:{anchura_de_campo}}]")      # Nombre: [Juan Luis   ]
```

Cadenas de caracteres con formato (II)

Anchura de campo y completado con ceros a la izquierda

```
dia = 4
mes = 7
anyo = 2013

print(f"FNac: {dia}/{mes}/{anyo}")             # FNac: 4/7/2013
print(f"FNac: {dia:2}/{mes:2}/{anyo:4}")       # FNac:  4/ 7/2013
print(f"FNac: {dia:02}/{mes:02}/{anyo:04}")    # FNac: 04/07/2013
```

Anchura de campo, precisión, notacion general, fija y científica

```
valor = (4+5+1)/3

print(f"Valor: {valor}")        # Valor: 3.3333333333333335
print(f"Valor: {valor:.4}")     # Valor: 3.333
print(f"Valor: {valor:8.4}")    # Valor:    3.333
print(f"Valor: {valor:08.4}")   # Valor: 0003.333

print(f"Valor: {valor:.4g}")     # Valor: 3.333   fija: (1e-5 < |valor| < 10**prec)
print(f"Valor: {valor:.4f}")     # Valor: 3.3333
print(f"Valor: {valor:.4e}")     # Valor: 3.3333e+00    (prec.def 6)
```

Cadenas de caracteres con formato (III)

Representación alternativa

```
nombre = "Jose Luis"

print(f"Nombre: {nombre}")    # -> print("Nombre: "+str(nombre))  -> Nombre: Jose Luis
print(f"Nombre: {nombre!r}") # -> print("Nombre: "+repr(nombre)) -> Nombre: 'Jose Luis'
```

Representación de llaves y comillas

```
nombre = "Jose Luis"

print(f"{{Nombre}}: {{{nombre}}}")       # -> {Nombre}: {Jose Luis}
print(f"'Nombre': {len(nombre)}")        # -> 'Nombre': 9
print(f"'Nombre': {len('Jose Luis')}")  # -> 'Nombre': 9
```

Cadenas de caracteres. Operaciones

Secuencias iterables

- Los objetos del tipo **str** son *secuencias iterables*. Se puede utilizar el bucle **for** para iterar sobre todos los elementos:

```
nombre = "Jose Luis"
for caracter in nombre:
    print(caracter, end=' ')            # J o s e   L u i s
print()
```

- `enumerate(iterable, start=0)`: devuelve un *iterador-enumerador* sobre el *iterable*. Por cada elemento del iterable, genera una tupla con el contador y el elemento.

```
nombre = "Jose Luis"
for (idx, caracter) in enumerate(nombre):
    print(idx, caracter, end=' ')    # 0 J 1 o 2 s 3 e 4   5 L 6 u 7 i 8 s
print()
```

Cadenas de caracteres. Operaciones

Acceso a los elementos del string (secuencia)

- `len(string)`: devuelve el número de caracteres del *string*.
- `string[idx]`: devuelve el carácter (de tipo str) de la posición *idx* del *string*.
  - El primer elemento tiene índice **0**. El último elemento tiene índice **len(string)-1**.
  - El indice negativo es equivalente a indexar desde el final (**len(string)-idx**). Por ejemplo **string[-3]** es equivalente a **string[len(string)-3]**.
- `string[inicio:fin]`: devuelve el **sub-string** (de tipo str) desde la posición *inicio* hasta la posición *fin*, sin incluirla.
- `string[inicio:fin:paso]`: devuelve el **sub-string** (de tipo str) desde la posición *inicio* hasta la posición *fin*, sin incluirla, con el incremento especificado.

```
nombre = "Jose Luis"
idx = 0
while idx < len(nombre):
    print(nombre[idx], end=' ')          # J o s e   L u i s
    idx += 1
print()

print(nombre[1:-2])                      # ose Lu
print(nombre[1:-2:2])                    # oeL
print(nombre[3:])                        # e Luis
print(nombre[:6])                        # Jose L
print(nombre[::2])                       # Js us
print(nombre[::-1])                      # siuL esoJ
```

Cadenas de caracteres. Operaciones

Conversiones

- `int(valor)`: devuelve el número entero correspondiente a *valor*.
- `float(valor)`: devuelve el número real correspondiente a *valor*.
- `complex(valor)`: devuelve el número complejo correspondiente a *valor*.
- `str(valor)`: devuelve la cadena de caracteres (string) correspondiente a *valor*.
- `repr(valor)`: devuelve la cadena de caracteres alternativa correspondiente a *valor*.

- `chr(numero)`: devuelve el carácter (*string unicode*) correspondiente al número entero.
- `ord(caracter)`: devuelve el número entero corresp. al *carácter* (*string unicode*).

```
numero = int(" 1234 ")              # lanza excepción ValueError si error
numero = float(" 1234.56e+10 ")     # lanza excepción ValueError si error
numero = complex(" 5+7j ")          # lanza excepción ValueError si error

string = str(False)
string = str(True)
string = str(1234)
string = str(1234.56e+10)
string = str(5+7j)

numero = ord("a")   # 97            # lanza excepción TypeError si error
string = chr(65)    # "A"           # lanza excepción TypeError si error
```

Cadenas de caracteres. Operaciones

Búsquedas

- `string.count(substr [, start [, end ]] )`: devuelve la cantidad de veces que *substr* se repite en *string*, desde *start* hasta *end*.
- `string.find(substr [, start [, end ]] )`: devuelve el índice de la primera ocurrencia de *substr* dentro de *string*, desde *start* hasta *end*. Devuelve -1 si no encontrado.
- `string.rfind(substr [, start [, end ]] )`: devuelve el índice de la última ocurrencia de *substr* dentro de *string*, desde *start* hasta *end*. Devuelve -1 si no encontrado.

```
texto = "el perro está en el monte"

cuenta = texto.count("el")          # cuenta = 2

idx = texto.find("el")              # idx = 0
idx = texto.rfind("el")             # idx = 17

idx = texto.find("pe")              # idx = 3
idx = texto.rfind("pe")             # idx = 3

idx = texto.find("x")               # idx = -1
idx = texto.rfind("x")              # idx = -1
```

Cadenas de caracteres. Operaciones

Comparaciones

- `==` `!=` `>` `>=` `<` `<=`: devuelven el resultado de las comparaciones lexicográficas.
- `substr in string`: devuelve `True` si *substr* está incluido en *string*.
- `substr not in string`: devuelve `True` si *substr* no está incluido en *string*.
- `string.startswith(prefix [, start [, end ]] )`: devuelve `True` si *string* comienza por *prefix*, desde *start* hasta *end*.
- `string.endswith(suffix [, start [, end ]] )`: devuelve `True` si *string* termina en *suffix*, desde *start* hasta *end*.

```
texto = "el perro está en el monte"
texto1 = "mantuvo"
texto2 = "mantiene"

ok = "perro" in texto                # True
ok = "gato" not in texto             # True
ok = texto.startswith("el")          # True
ok = texto.endswith("monte")         # True
ok = texto1 == texto2                # False
ok = texto1 != texto2                # True
ok = texto1 > texto2                 # True
ok = texto1 >= texto2                # True
ok = texto1 < texto2                 # False
ok = texto1 <= texto2                # False
```

Cadenas de caracteres. Operaciones

Comparaciones

- `string.isdecimal()`: devuelve `True` si todos los caracteres de *string* son dígitos numéricos *(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)*.
- `string.islower()`: devuelve `True` si no hay letras mayúsculas incluidas en *string*, y al menos hay una letra minúscula.
- `string.isupper()`: devuelve `True` si no hay letras minúsculas incluidas en *string*, y al menos hay una letra mayúscula.
- `string.isalpha()`: devuelve `True` si todos los caracteres de *string* son letras.
- `string.isalnum()`: devuelve `True` si todos los caracteres de *string* son letras o dígitos.

```
ok = "123".isdecimal()              # True
ok = "123 ".isdecimal()             # False
ok = "-123".isdecimal()             # False
ok = "123abc".islower()             # True
ok = "123Abc".islower()             # False
ok = "123ABC".isupper()             # True
ok = "123aBC".isupper()             # False
ok = "ABCabc".isalpha()             # True
ok = "ABC abc".isalpha()            # False
ok = "123ABCabc".isalnum()          # True
ok = "123ABC abc".isalnum()         # False
```

Cadenas de caracteres. Operaciones

Transformaciones

- `string.capitalize()`: devuelve una nueva cadena, con el primer carácter mayúscula y el resto minúsculas.
- `string.lower()`: devuelve una nueva cadena, con todos los caracteres minúsculas.
- `string.upper()`: devuelve una nueva cadena, con todos los caracteres mayúsculas.

- `string.strip(chars=" \t\n\r\f\v")`: devuelve una nueva cadena, eliminado por el principio y por el final, cualquiera de los caracteres que aparecen en *chars*.
- `string.lstrip(chars=" \t\n\r\f\v")`: devuelve una nueva cadena, eliminado por el principio, cualquiera de los caracteres que aparecen en *chars*.
- `string.rstrip(chars=" \t\n\r\f\v")`: devuelve una nueva cadena, eliminado por el final, cualquiera de los caracteres que aparecen en *chars*.

```
texto = "pepe".capitalize()              # "Pepe"
texto = "PEPE".lower()                   # "pepe"
texto = "pepe".upper()                   # "PEPE"

texto = "   esto es texto   ".strip()    # "esto es texto"
texto = "   esto es texto   ".lstrip()   # "esto es texto   "
texto = "   esto es texto   ".rstrip()   # "   esto es texto"
```

Cadenas de caracteres. Operaciones

Transformaciones

- `string.replace(old, new, count=-1)`: devuelve una nueva cadena, reemplazando todas (como máximo *count*) las ocurrencias de *old* por *new*.
- `string.removeprefix(prefix)`: devuelve una nueva cadena, eliminado *prefix* del inicio, si está.
- `string.removesuffix(suffix)`: devuelve una nueva cadena, eliminado *suffix* del final, si está.

```
texto = "tuvo un coche que mantuvo"
texto2 = texto.replace("tuvo", "tiene") # "tiene un coche que mantiene"
texto3 = texto.removeprefix("tu")       # "vo un coche que mantuvo"
texto3 = texto.removesuffix("vo")       # "tuvo un coche que mantu"
```

Cadenas de caracteres. Operaciones

Concatenación

- El operador `+` crea una nueva cadena resultado de concatenar (unir) dos cadenas de caracteres (se debe convertir a **str** si no es cadena de caracteres).
- El operador `+=` crea una nueva cadena resultado de concatenar (unir) la cadena actual con otra cadena de caracteres (se debe convertir a **str** si no es cadena de caracteres) (`string = string + otro_string`).

- `separador.join(iterable_de_str)`: devuelve una nueva cadena de caracteres (de tipo str) resultado de unir todas las cadenas de caracteres del *iterable*, separadas por el *separador* (de tipo str) especificado.

```
nombre = "Jose" + " Luis " + str(234)    # concatenacion de strings
nombre += " Lopez Vazquez"               # nombre = nombre + " Lopez Vazquez"
print(nombre)                            # Jose Luis 234 Lopez Vazquez

lista = ["fresa", "manzana", "naranja"]
texto = "; ".join(lista)                 # "fresa; manzana; naranja"
texto = "-".join("hola")                 # "h-o-l-a"
```

Cadenas de caracteres. Operaciones

División

- `string.split(sep=None, maxsplit=-1)`: devuelve una nueva lista con todas (como máximo *maxsplit+1*) las subcadenas, resultado de dividir el *string* con el *separador* especificado. Si *seperador* es `None`, entonces utiliza **espacios** como separador.
- `string.splitlines()`: devuelve una nueva lista con todas las subcadenas, resultado de dividir el *string* por los **saltos de línea**.

```
texto = "  fresa   manzana   naranja  "
lista = texto.split()                    # ["fresa", "manzana", "naranja"]

texto = "fresa; manzana; naranja"
lista = texto.split("; ")                # ["fresa", "manzana", "naranja"]

texto = "  fresa ; manzana ; naranja  "
lista = texto.split(";")                 # ["  fresa ", " manzana ", " naranja  "]

texto = "fresa\nmanzana\nnaranja"
lista = texto.splitlines()               # ["fresa", "manzana", "naranja"]

texto = "fresa \n manzana \n naranja \n"
lista = texto.splitlines()               # ["fresa ", " manzana ", " naranja "]
```

Cadenas de caracteres. Ejemplo

- Programa que lee una frase, compuesta de palabras separadas por espacios, y crea otra frase donde para cada palabra, si comienza por vocal, reemplaza la vocal por el símbolo `"#"`, si comienza por mayúsculas, se convierte la palabra a mayúsculas, si comienza por minúsculas, se convierte la palabra a minúsculas, separando las letras con el símbolo `"-"`, si es un número, se múltiplica el reverso de su valor por 2. Por ejemplo:

```
Introduce una frase: Hola, hOY es UN Gran 12abCDE 123 432
Resultado: HOLA, h-o-y #s #N GRAN 12abCDE 642 468
```

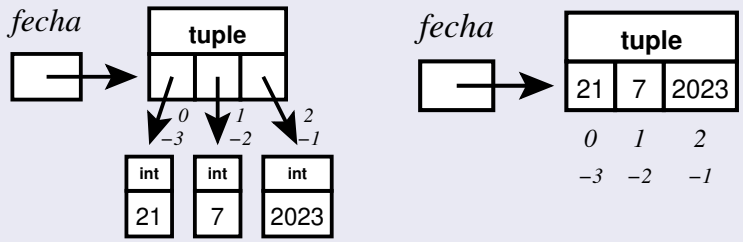
```python
def procesar_frase(frase: str) -> str:
    resultado = ""
    for palabra in frase.split():
        if len(palabra) > 0:
            if palabra[0].lower() in "aeiouáéíóú":
                pal = "#" + palabra[1:]
            elif palabra[0].isupper():
                pal = palabra.upper()
            elif palabra[0].islower():
                pal = "-".join(palabra.lower())
            elif palabra.isdecimal():
                pal = str(2*int(palabra[::-1]))
            else:
                pal = palabra
            resultado += " " + pal
    return resultado.strip()
```

```python
def main() -> None:
    frase = input("Introduce una frase: ")
    # print("Resultado:", procesar_frase(frase))
    print(f"Resultado: {procesar_frase(frase)}")

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

Tuplas

- Una tupla es una secuencia iterable e **inmutable** de elementos (valores), que se utiliza como **agrupación** de valores. Su tipo es **tuple**.

```
fecha = (21, 7, 2023)          # empaquetado
fecha = (21, "Julio", 2023)    # empaquetado
print(fecha)                   # (21, 'Julio', 2023)
(dia, mes, anyo) = fecha       # desempaquetado
(dia, mes, anyo) = (21, 7, 2023) # asig. múltiple
(x, y) = (y, x)                  # asig. múltiple (intercambio de valores)

z = (1,)                       # tupla con un único elemento -> la coma es necesaria
```



- Aunque una tupla es *inmutable*, puede contener elementos *mutables*, que sí son *modificables*, aunque no son *reemplazables*.

- Se puede crear una tupla con los valores de un **iterable**:

```
tupla = tuple(iterable)

tupla = tuple(range(3))
tupla = tuple((2, 3, 4))
tupla = tuple([2, 3, 4])
```

Tuplas por comprensión (tuple comprehensions)

- Las **tuplas por comprensión** permiten crear tuplas de forma concisa, iterando sobre los elementos de un *iterable*, seleccionando elementos según alguna condición, y transformando los elementos seleccionados:

```
tupla = tuple(x for x in range(10))          # (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
tupla = tuple(x**2 for x in (1, 2, 3))       # (1, 4, 9)
tupla = tuple(x+5 for x in (1, 2, 3))        # (6, 7, 8)

tupla = tuple(x for x in range(10) if x % 2 == 0) # (0, 2, 4, 6, 8)

tupla = tuple((i, x) for (i, x) in enumerate((1, 2, 2)) if x == 2) # ((1, 2), (2, 2)

tupla = tuple((x, y) for x in (1, 2, 3) for y in (3, 1, 4) if x != y)
      # ((1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4))
```

Tuplas

Secuencias iterables

- Se puede utilizar el bucle **for** para iterar sobre todos los elementos de una secuencia iterable:

```
fecha = (21, 7, 2023)
for elemento in fecha:
    print(elemento, end=' ')
print()
```

```
fecha = (21, "Julio", 2023)
for (idx, elemento) in enumerate(fecha):
    print(idx, elemento, end=' ')
print()
```

Pertenencia de miembro

- `elemento in tupla`: devuelve `True` si *elemento* es miembro de *tupla*.
- `elemento not in tupla`: devuelve `True` si *elemento* no es miembro de *tupla*.

```
fecha = (21, "Julio", 2023)
ok = 3 in fecha              # True
ok = "Julio" in fecha        # True
ok = 2023 in fecha           # True
ok = "Enero" not in fecha    # True
```

Tuplas

Acceso a los elementos de la tupla (secuencia)

- `len(tupla)`: devuelve el número de elementos de la *tupla*.
- `tupla[idx]`: devuelve el elemento de la posición *idx* de la *tupla*.
  - El primer elemento tiene índice **0**. El último elemento tiene índice **len(tupla)-1**.
  - El indice negativo es equivalente a indexar desde el final (**len(tupla)-idx**). Por ejemplo **tupla[-3]** es equivalente a **tupla[len(tupla)-3]**.
- `tupla[inicio:fin]`: devuelve la **sub-tupla** (de tipo tuple) desde la posición *inicio* hasta la posición *fin*, sin incluirla.
- `tupla[inicio:fin:paso]`: devuelve la **sub-tupla** (de tipo tuple) desde la posición *inicio* hasta la posición *fin*, sin incluirla, con el incremento especificado.

```
fecha = (21, 7, 2023)
x = len(fecha) # 3
x = fecha[0]   # 21
x = fecha[1]   # 7
x = fecha[2]   # 2023
```

```
x = fecha[0:2]  # (21, 7)
x = fecha[1:-1] # (7,)
x = fecha[:2]   # (21, 7)
x = fecha[1:]   # (7, 2023)
x = fecha[::2]  # (21, 2023)
```

Tuplas

Comparaciones

- `==` `!=` `>` `>=` `<` `<=`: devuelven el resultado de las comparaciones lexicográficas.

Concatenación

- El operador `+` crea una nueva tupla resultado de concatenar otras dos tuplas.
- El operador `+=` crea una nueva tupla resultado de concatenar la tupla actual con otra tupla (`tupla = tupla + otra_tupla`).

```
x = (1, 2, 3) + (4, 5) # (1, 2, 3, 4, 5)
x += (6,)              # (1, 2, 3, 4, 5, 6)
# la tupla con 1 elemento debe tener una coma
```

```
(1, 2, 3) == (1, 2)           # False
(1, 2, 3) != (1, 2)           # True
(1, 2, 3) >= (1, 2)           # True
(1, 2, 3) > (1, 2)            # True
(1, 1, 3) > (1, 2)            # False
(1, 2, 3) <= (1, 2)           # False
(1, 2, 3) < (1, 2)            # False
(1, 2, 3, 4) == (1, 2, 3, 4) # True
(1, 2, 3, 4) == (1, 2, 5, 4) # False
(1, 2, 3, 4) != (1, 2, 5, 4) # True
(1, 2, 3, 4) >= (1, 2, 5, 4) # False
(1, 2, 3, 4) > (1, 2, 5, 4)  # False
(1, 2, 3, 4) <= (1, 2, 5, 4) # True
(1, 2, 3, 4) < (1, 2, 5, 4)  # True
```

Tuplas. Ejemplo

Programa de notas de alumnos

- Lee el **nombre y la nota** de cada alumno y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual a 5.

```python
from typing import Final, Optional

_UMBRAL_APROBADO: Final = 5

def leer_alumno() -> Optional[tuple[str,float]]:
    """Lee los datos de un alumno."""
    res: Optional[tuple[str,float]] = None
    nombre = input("Introduce Nombre: ")
    if len(nombre) > 0:
        nota = float(input("Introduce nota: "))
        res = (nombre, nota)
    return res

def mostrar_calificacion(alumno: tuple[str,float]) -> None:
    """Muestra la calificación de un alumno."""
    if alumno[1] >= _UMBRAL_APROBADO:
        print(f"Alumno: {alumno[0]} Aprobado")
    else:
        print(f"Alumno: {alumno[0]} Suspenso")

def main() -> None:
    """Lee los datos de varios alumnos
    y muestra calificaciones."""
    alumno = leer_alumno()
    while alumno is not None:
        mostrar_calificacion(alumno)
        alumno = leer_alumno()

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

Tuplas con nombres: NamedTuple

- Podemos definir un **nuevo tipo tupla** (con un nuevo nombre de tipo), y podemos especificar el **nombre** con el que acceder a los elementos (además del acceso por índice), así como su tipo.
  - Las tuplas con nombres (`NamedTuple`) son también **tuplas inmutables**, y todas las operaciones de tuplas están incluidas.

```python
from typing import NamedTuple  # NamedTuple se debe importar de typing

class Fecha(NamedTuple):
    """Fecha"""
    dia: int
    mes: str
    anyo: int = 2000  # elms con valores por defecto al final

# creación de nuevos objetos tupla (de tipo tupla Fecha)
fecha = Fecha(21, "Julio")  # anyo = 2000
fecha = Fecha(21, "Julio", 2023)
fecha = Fecha(dia=21, mes="Julio", anyo=2023)

print(fecha)    # muestra Fecha(dia=21, mes='Julio', anyo=2023)
# se puede acceder a los elementos por nombre
print(f"{fecha.dia}/{fecha.mes}/{fecha.anyo}")  # muestra 21/Julio/2023
# se puede acceder a los elementos por índice, como una tupla normal
print(f"{fecha[0]}/{fecha[1]}/{fecha[2]}")      # muestra 21/Julio/2023
```

NamedTuple. Ejemplo

Programa de notas de alumnos

- Lee el **nombre y la nota** de cada alumno y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual a 5.

```python
from typing import Final, Optional, NamedTuple

_UMBRAL_APROBADO: Final = 5

class Alumno(NamedTuple):
    nombre: str
    nota: float

def leer_alumno() -> Optional[Alumno]:
    """Lee los datos de un alumno."""
    res: Optional[Alumno] = None
    nombre = input("Introduce Nombre: ")
    if len(nombre) > 0:
        nota = float(input("Introduce nota: "))
        res = Alumno(nombre, nota)
    return res
```

```python
def mostrar_calificacion(alumno: Alumno) -> None:
    """Muestra la calificación de un alumno."""
    if alumno.nota >= _UMBRAL_APROBADO:
        print(f"Alumno: {alumno.nombre} Aprobado")
    else:
        print(f"Alumno: {alumno.nombre} Suspenso")

def main() -> None:
    """Lee los datos de varios alumnos
    y muestra calificaciones."""
    alumno = leer_alumno()
    while alumno is not None:
        mostrar_calificacion(alumno)
        alumno = leer_alumno()

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

Datos mutables: @dataclass

- Podemos definir un **nuevo tipo**, como **agrupación mutable de datos** (con un nuevo nombre de tipo), y podemos especificar el **nombre** con el que acceder a los elementos, así como su tipo.

```python
from dataclasses import dataclass

@dataclass(order=True)
class Producto:
    """Datos de un producto de almacén."""
    nombre: str
    cantidad: int = 0 # al final

# creación de objetos del nuevo tipo
p1 = Producto("martillo", 20)
p2 = Producto(nombre="martillo", cantidad=20)
p3 = Producto("taladro") # cantidad=0

# se pueden acceder y modificar los campos
p1.cantidad += 5
p2.cantidad += 5
p3.cantidad += 10
```

```python
# se puede acceder a los elementos por nombre
print(f"({p1.nombre}, {p1.cantidad})") # (martillo, 25)

# se puede mostrar el contenido completo
print(p1) # Producto(nombre='martillo', cantidad=25)
print(p2) # Producto(nombre='martillo', cantidad=25)
print(p3) # Producto(nombre='taladro', cantidad=10)

# representación textual del contenido completo
x = repr(p1) # Producto(nombre='martillo', cantidad=25)
x = str(p2)  # Producto(nombre='martillo', cantidad=25)

# comparación del contenido completo == != < <= > >=
ok = (p1 == p2)  # True
ok = (p1 < p3)   # True
```

@dataclass. Ejemplo

Programa de notas de alumnos

- Lee el **nombre y la nota** de cada alumno y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual a 5.

```python
from typing import Final, Optional
from dataclasses import dataclass

_UMBRAL_APROBADO: Final = 5

@dataclass(order=True)
class Alumno:
    nombre: str
    nota: float
    calificacion: Optional[str] = None

def leer_alumno() -> Optional[Alumno]:
    """Lee los datos de un alumno."""
    res: Optional[Alumno] = None
    nombre = input("Introduce Nombre: ")
    if len(nombre) > 0:
        nota = float(input("Introduce nota: "))
        res = Alumno(nombre, nota)
    return res

def calcular_calificacion(alumno: Alumno) -> None:
    """Calcula la calificación de un alumno."""
    if alumno.nota >= _UMBRAL_APROBADO:
        alumno.calificacion = "Aprobado"
    else:
        alumno.calificacion = "Suspenso"

def mostrar_calificacion(alumno: Alumno) -> None:
    """Muestra la calificación de un alumno."""
    print(f"Alumno: {alumno.nombre} {alumno.calificacion}")

def main() -> None:
    alumno = leer_alumno()
    while alumno is not None:
        calcular_calificacion(alumno)
        mostrar_calificacion(alumno)
        alumno = leer_alumno()

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

Agrupación de datos estructurados inmutables vs. mutables

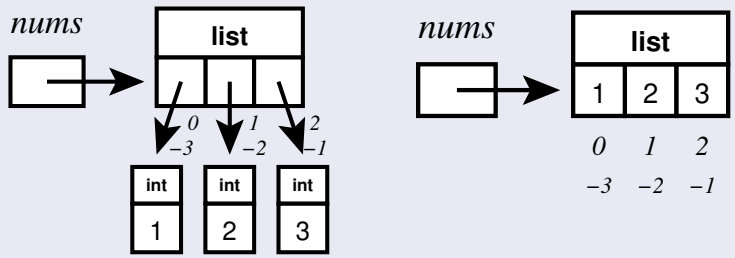
- Tanto `NamedTuple` como `@dataclass` sirven para representar una **agrupación de datos** estructurados, cuyos componentes pueden ser accedidos por sus nombres.
- Sin embargo `NamedTuple` representa datos estructurados **inmutables**, mientras que `@dataclass` representa datos estructurados **mutables**. Esta diferencia es importante, ya que determina el contexto de utilización de ambos.
  - Cuando se desea representar información estructurada *definitiva*, que no va a ser modificada, entonces se recomienda utilizar `NamedTuple`.
  - Cuando se desea representar información estructurada que podría ser modificada, entonces se recomienda utilizar `@dataclass`.
- Una característica importante de `NamedTuple` es que también es una **tupla**, por lo que puede ser utilizado en cualquier sitio donde pudiera ser utilizada una tupla, y proporciona las operaciones de tuplas.

- **Atención**, lo anterior se refiere a representación de datos estructurados, sin considerar los aspectos de **abstracción de datos** que serán explicados en el tema de *programación orientada a objetos*.

Listas

- Una lista es una secuencia iterable y **modificable** de elementos (valores), que se utiliza como **lista** de valores. Su tipo es **list**.
  - Cada **elemento** de la lista está asociado a una **posición** dentro de la lista.

```
lista_vacia = list()
lista_vacia = []
nums = [1, 2, 3]
examenes = [(3, "Marzo", 2023), (12, "Abril", 2023)]
alumno = ("Jose Luis", [6.5, 7.4])

print(nums)      # [1, 2, 3]
```



- Se puede crear una lista con los valores de un **iterable**:

```
lista = list(iterable)

lista = list(range(3))
lista = list((2, 3, 4))
lista = list([2, 3, 4])
```

Listas

Listas por comprensión (list comprehensions)

- Las **listas por comprensión** permiten crear listas de forma concisa, iterando sobre los elementos de un *iterable*, seleccionando elementos según alguna condición, y transformando los elementos seleccionados:

```
lista = [x for x in range(10)]                  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
lista = [x**2 for x in (1, 2, 3)]               # [1, 4, 9]
lista = [x+5 for x in [1, 2, 3]]                # [6, 7, 8]

lista = [x for x in range(10) if x % 2 == 0] # [0, 2, 4, 6, 8]
lista = [(i, x) for (i, x) in enumerate([1, 2, 2]) if x == 2] # [(1, 2), (2, 2)]

lista = [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
      # [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Equivalencia

```
lista = [(i, x) for (i, x) in enumerate([1, 2, 2]) if x == 2]

lista = []
for (i, x) in enumerate([1, 2, 2]):
    if x == 2:
        lista.append( (i, x) )
```

Listas

Secuencias iterables

- Se puede utilizar el bucle **for** para iterar sobre todos los elementos de una secuencia iterable:

```
nums = [1, 2, 3, 4]
for elemento in nums:
    print(elemento, end=' ')
print()
```

```
examenes = [(3, "Marzo", 2023), (12, "Abril", 2023)]
for (idx, elemento) in enumerate(examenes):
    print(idx, elemento, end=' ')
print()
```

Pertenencia de miembro

- `elemento in lista`: devuelve `True` si *elemento* es miembro de *lista*.
- `elemento not in lista`: devuelve `True` si *elemento* no es miembro de *lista*.

```
nums = [1, 2, 3, 4]
ok = 1 in nums       # True
ok = 2 in nums       # True
ok = 3 in nums       # True
ok = 4 in nums       # True
ok = 5 not in nums   # True
```

Listas

Acceso a los elementos de la lista (secuencia)

- `len(lista)`: devuelve el número de elementos de la *lista*.
- `lista[idx]`: devuelve el elemento de la posición *idx* de la *lista*.
  - El primer elemento tiene índice **0**. El último elemento tiene índice **len(lista)-1**.
  - El indice negativo es equivalente a indexar desde el final (**len(lista)-idx**). Por ejemplo **lista[-3]** es equivalente a **lista[len(lista)-3]**.
- `lista[inicio:fin]`: devuelve una **nueva sub-lista** (de tipo list) desde la posición *inicio* hasta la posición *fin*, sin incluirla.
- `lista[inicio:fin:paso]`: devuelve una **nueva sub-lista** (de tipo list) desde la posición *inicio* hasta la posición *fin*, sin incluirla, con el incremento especificado.

```
nums = [1, 2, 3, 4]
x = len(nums)              # 4
x = nums[0] # nums[-4]   # 1
x = nums[1] # nums[-3]   # 2
x = nums[2] # nums[-2]   # 3
x = nums[3] # nums[-1]   # 4

nums[2] = 6    # [1, 2, 6, 4]

copia = nums[:]    # copia
```

```
nums = [1, 2, 3, 4]
x = nums[1:-1]     # [2, 3]
x = nums[0:-1:2]   # [1, 3]
x = nums[2:]       # [3, 4]
x = nums[:2]       # [1, 2]
x = nums[::2]      # [1, 3]

nums[1:-1] = [7, 8, 9]   # [1, 7, 8, 9, 4] reemplaza elementos
nums[2:4] = [10]         # [1, 7, 10, 4]   reemplaza elementos
nums[::2] = [20, 21]     # [20, 7, 21, 4]  misma longitud
nums[:] = [1, 2, 3]      # [1, 2, 3] reemplaza todos elementos
nums[:] = []             # []        reemplaza todos elementos
```

Listas

Manipulación de elementos de listas

- `lista.extend(iterable)`: añade todos los elementos del *iterable* al final de *lista*.
- `lista.append(elemento)`: añade un elemento al final de *lista*.
- `lista.insert(pos, elemento)`: añade a *lista* un elemento en la *posición* indicada.
- `lista.pop(pos=-1)`: devuelve y elimina de *lista* el elemento de la *posición* especificada. Si *pos* no se especifica, elimina el último elemento.
- `lista.clear()`: elimina todos los elementos de *lista* (queda vacía).

- `del lista[pos]`: elimina de *lista* el elemento de la *posición* especificada.
- `del lista[inicio:fin:paso]`: elimina de *lista* los elementos especificados por los índices de la *sub-lista*.

- Mientras se está **recorriendo** una lista, **no** se pueden **añadir** ni **eliminar** elementos.

```
x = [1, 2, 3]           # [1, 2, 3]
x.extend([4, 5])        # [1, 2, 3, 4, 5]
x.append(6)             # [1, 2, 3, 4, 5, 6]
x.insert(1, 9)          # [1, 9, 2, 3, 4, 5, 6]
z = x.pop(1)            # [1, 2, 3, 4, 5, 6] -> 9
z = x.pop()             # [1, 2, 3, 4, 5]    -> 6
x.clear()               # []
```

```
x = [1, 2, 3, 4, 5, 6]
del x[2]     # [1, 2, 4, 5, 6]
del x[1:3]   # [1, 5, 6]

x = [1, 2, 3, 4, 5, 6]
del x[::2]   # [2, 4, 6]
```

Listas

Comparaciones

- `==` `!=` `>` `>=` `<` `<=`: devuelven el resultado de las comparaciones lexicográficas.

Concatenación

- El operador `+` crea una nueva lista resultado de concatenar otras dos listas.
- El operador `+=` añade los elementos de una lista al final de otra lista ya existente (equivalente a `lista.extend(iterable)`).

```
x = [1, 2, 3] + [4, 5] # [1, 2, 3, 4, 5]
x += [6]               # [1, 2, 3, 4, 5, 6]

alumno = ("Jose Luis", [6.5, 7.4])
alumno[1][0] = 9.5
alumno[1].append(8.5)
# Aunque la tupla es inmutable, el elemento
# lista dentro de la tupla sí se puede modificar
# ("Jose Luis", [9.5, 7.4, 8.5])
```

```
[1, 2, 3] == [1, 2]           # False
[1, 2, 3] != [1, 2]           # True
[1, 2, 3] >= [1, 2]           # True
[1, 2, 3] > [1, 2]            # True
[1, 2, 3] <= [1, 2]           # False
[1, 2, 3] < [1, 2]            # False
[1, 2, 3, 4] == [1, 2, 3, 4]  # True
[1, 2, 3, 4] == [1, 2, 5, 4]  # False
[1, 2, 3, 4] != [1, 2, 5, 4]  # True
[1, 2, 3, 4] >= [1, 2, 5, 4]  # False
[1, 2, 3, 4] > [1, 2, 5, 4]   # False
[1, 2, 3, 4] <= [1, 2, 5, 4]  # True
[1, 2, 3, 4] < [1, 2, 5, 4]   # True
```

Listas

Otras operaciones

- `lista.count(elemento)`: devuelve la cantidad de veces que el *elemento* se repite en *lista*.
- `lista.reverse()`: invierte los elementos de la lista.
- `lista.sort(*, key=None, reverse=False)`: ordena los elementos de *lista*.
- `lista.remove(elemento)`: buscar el elemento en *lista*, si lo encuentra, elimina el primero. Si no lo encuentra, lanza excepción **ValueError**
- `lista.index(elemento [, start [, end ]])`: buscar el elemento en *lista*, si lo encuentra, devuelve su posición. Si no lo encuentra, lanza excepción **ValueError**

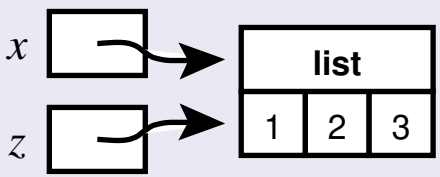
```
x = [7, 3, 5, 1, 5, 3, 5, 8]
z = x.count(5)          # 3
z = x.index(5)          # 2
x.remove(5)             # [7, 3, 1, 5, 3, 5, 8]

x.reverse()             # [8, 5, 3, 5, 1, 3, 7]

x.sort()                # [1, 3, 3, 5, 5, 7, 8]
x.sort(reverse=True)    # [8, 7, 5, 5, 3, 3, 1]
```

```
def voltear(elem: tuple[int,int]) -> tuple[int,int
    return (elem[1], elem[0])

w = [(3, 8), (1, 9), (2, 7)]
w.sort()    # [(1, 9), (2, 7), (3, 8)]

w.sort(key=voltear)
            # [(2, 7), (3, 8), (1, 9)]
```

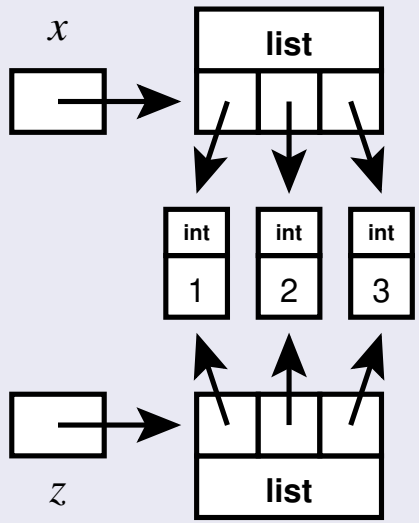
Duplicación y copia de listas

- Las listas son objetos **mutables** (modificables).
- La asignación de variables asigna la referencia, pero el objeto es compartido.
- Si queremos realizar una modificación en un determinado objeto:
  - A veces queremos que la modificación se realice en el objeto compartido.
  - A veces queremos que la modificación se realice en otro objeto independiente, **copia** (duplicación) del original. `copy()` realiza una **copia superficial** de la lista.

```
x = [1, 2, 3] # [1, 2, 3]
z = x  # Objeto compartido

z.append(4)
z[1] = 6
print(x)     # [1, 6, 3, 4]
print(z)     # [1, 6, 3, 4]
```




```
x = [1, 2, 3] # [1, 2, 3]
z = x.copy()  # copia superficial
z = list(x)   # copia superficial
z = x[:]      # copia superficial

z.append(4)
z[1] = 6
print(x)     # [1, 2, 3]
print(z)     # [1, 6, 3, 4]
```



Búsquedas

- Buscar la posición que ocupa, en una lista, el primer elemento que cumpla una determinada condición. Devuelve -1 si no encontrado.

```
def buscar(lista: list, elem) -> int:
    if elem in lista:
        pos = lista.index(elem)
    else:
        pos = -1
    return pos

# Se puede hacer mejor capturando
# excepciones. Lo veremos en
# el tema 2
```

```
def buscar(lista: list, elem) -> int:
    pos = 0
    while ((pos < len(lista))
           and (elem != lista[pos])):
        pos += 1
    return pos if pos < len(lista) else -1
```

```
def buscar(lista: list, elem) -> int:
    pos = -1
    idx = 0
    while ((idx < len(lista)) and (pos < 0)):
        if (elem == lista[idx]):
            pos = idx
        idx += 1
    return pos
```

Listas. Ejemplo-1 (I)

Leer lista de números

- Lee de teclado una lista de números naturales (cero y positivos), separados por espacios, hasta introducir una *línea vacía*, y finalmente la muestra en pantalla.

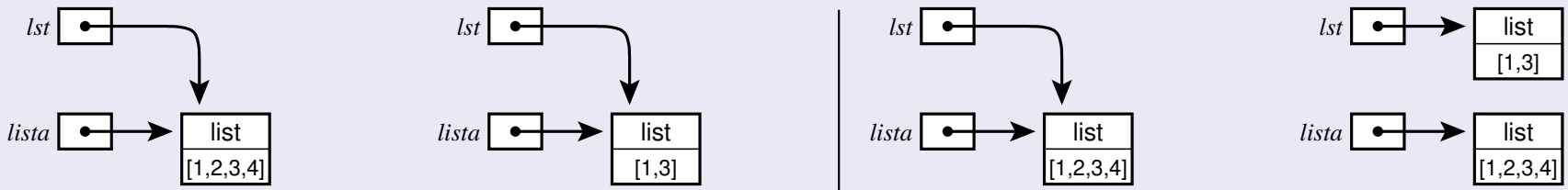
```
def leer_lista(mensaje: str) -> list[int]:
    """devuelve una lista de números naturales leídos de teclado (hasta línea vacía)"""
    lista: list[int] = list()
    datos = input(mensaje)
    while len(datos) > 0:
        lista.extend(int(x) for x in datos.split() if x.isdecimal())
        datos = input()
    return lista

def main() -> None:
    print("Lista:", leer_lista("Introduce lista de números: "))

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

Listas. Ejemplo-1 (II)

Eliminar elementos de una lista de números

- Eliminar todos los números pares de una lista de números enteros.

```
def eliminar_pares(lst: list[int]) -> None:
    """elimina los números pares de una lista de números enteros"""
    lst[:] = [num for num in lst if num % 2 != 0]

    # ATENCIÓN, lo siguiente no es válido
    # véase pg 43, sección "Paso de parámetros por valor de la referencia al objeto"
    # lst = [num for num in lst if num % 2 != 0]

def main() -> None:
    lista = leer_lista("Introduce lista de números: ")
    eliminar_pares(lista)
    print("Lista:", lista)

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



Listas. Ejemplo-2 (I)

Programa de notas de alumnos

- Lee el **nombre y la nota** de cada alumno y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota es mayor o igual **a la nota media de todos los alumnos**.

```
from typing import Optional, NamedTuple
class Alumno(NamedTuple):
    nombre: str
    nota: float

def leer_alumno() -> Optional[Alumno]:
    """Lee los datos de un alumno."""
    res: Optional[Alumno] = None
    nombre = input("Introduce Nombre: ")
    if len(nombre) > 0:
        nota = float(input("Introduce nota: "))
        res = Alumno(nombre, nota)
    return res
def leer_alumnos() -> list[Alumno]:
    """Lee los datos de varios alumnos."""
    lista_alumnos: list[Alumno] = list()
    alumno = leer_alumno()
    while alumno is not None:
        lista_alumnos.append(alumno)
        alumno = leer_alumno()
    return lista_alumnos
```

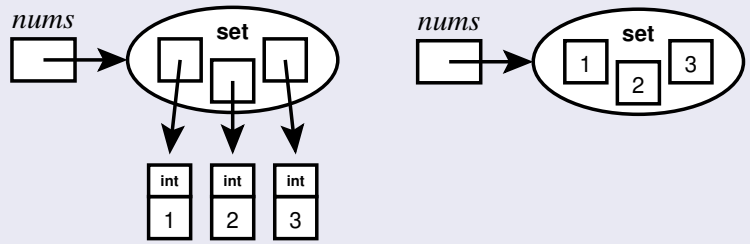
Listas. Ejemplo-2 (II)

```python
def calc_media(lista_alumnos: list[Alumno]) -> float:
    """Calcula la nota media."""
    media = 0.0
    if len(lista_alumnos) > 0:
        suma = 0.0
        for alumno in lista_alumnos:
            suma += alumno.nota
        media = suma / len(lista_alumnos)
    return media

def mostrar_alumno(alumno: Alumno, umbral: float) -> None:
    """Muestra la nota final de un alumno."""
    if alumno.nota >= umbral:
        print(f"Alumno: {alumno.nombre} Aprobado")
    else:
        print(f"Alumno: {alumno.nombre} Suspenso")

def mostrar_alumnos(lista_alumnos: list[Alumno], umbral: float) -> None:
    """Muestra la nota de los alumnos."""
    for alumno in lista_alumnos:
        mostrar_alumno(alumno, umbral)

def main() -> None:
    """Lee las notas de alumnos y muestra su nota final."""
    alumnos = leer_alumnos()
    media = calc_media(alumnos)
    mostrar_alumnos(alumnos, media)

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

Conjuntos

- Un conjunto es una colección iterable y **modificable** de elementos (valores), que se utiliza como un **conjunto** de valores. Su tipo es **set**.
  - El conjunto está organizado para realizar **consultas eficientes** (muy rápidas), para saber si un elemento se sencuentra en el conjunto (operador `in`).
  - No hay elementos **repetidos**. Los elementos repetidos no se añaden al conjunto.
  - Los elementos del conjunto deben ser **inmutables** (componentes utilizados en el hash).
  - Cada **elemento** del conjunto **no** está asociado con ninguna posición determinada dentro del conjunto. Un conjunto **no** es una secuencia.

```
conjunto_vacio = set() # no es válido {}
nums = {1, 2, 3}
examenes = {(3, "Marzo", 2023), (12, "Abril", 2023)}

print(nums)      # {1, 2, 3}
```



- Se puede crear un conjunto con los valores de un **iterable**:

```
conjunto = set(iterable)
conjunto = set([2, 3, 4])
conjunto = set((2, 3, 4))
conjunto = set(range(3))
```

Conjuntos

Conjuntos por comprensión (set comprehensions)

- Los **conjuntos por comprensión** permiten crear conjuntos de forma concisa, iterando sobre los elementos de un *iterable*, seleccionando elementos según alguna condición, y transformando los elementos seleccionados:

Conjuntos iterables

- Se puede utilizar el bucle **for** para iterar sobre todos los elementos de un conjunto iterable.
  - Se desconoce el orden de iteración de los elementos.

```
conjunto = {x**2 for x in range(5) if x%2==0}

for elemento in conjunto:
    print(elemento, end=' ')      # 0 16 4
print()
```

Conjuntos

Pertenencia de miembro

- `elemento in conjunto`: devuelve `True` si *elemento* es miembro de *conjunto*.
- `elemento not in conjunto`: devuelve `True` si *elemento* no es miembro de *conjunto*.

Comparaciones

- `== !=`: devuelven el resultado de las comparaciones de igualdad o desigualdad. Comparan la igualdad de los elementos, el orden de los elementos no influye.
- `> >= < <=`: devuelven el resultado de las comparaciones de superconjuntos o subconjuntos, el orden de los elementos no influye.

```
nums = {1, 2, 3, 4}
ok = 1 in nums       # True
ok = 2 in nums       # True
ok = 3 in nums       # True
ok = 4 in nums       # True
ok = 5 not in nums   # True
```

```
{1, 2, 3} == {1, 2}          # False
{1, 2, 3} != {1, 2}          # True
{1, 2, 3} >= {2, 1}          # True
{1, 2, 3} > {2, 1}           # True
{1, 2, 3} <= {2, 1}          # False
{1, 2, 3} < {2, 1}           # False
{1, 2, 3, 4} == {2, 3, 4, 1} # True
{1, 2, 3, 4} == {1, 2, 5, 4} # False
{1, 2, 3, 4} != {1, 2, 5, 4} # True
{1, 2, 3, 4} >= {2, 3, 4, 1} # True
{1, 2, 3, 4} > {2, 3, 4, 1}  # False
{1, 2, 3, 4} <= {2, 3, 4, 1} # True
{1, 2, 3, 4} < {2, 3, 4, 1}  # False
```

Conjuntos

Manipulación de elementos del conjunto

- `len(conjunto)`: devuelve el número de elementos del *conjunto*.
- `conjunto.add(elemento)`: añade un elemento a un conjunto. Si está repetido, no se añade.
- `conjunto.discard(elemento)`: elimina el elemento de *conjunto*. Si no lo encuentra, no hace nada.
- `conjunto.remove(elemento)`: elimina el elemento de *conjunto*. Si no lo encuentra, lanza excepción **KeyError**.
- `conjunto.pop()`: devuelve un elemento cualquiera del conjunto, y lo elimina del conjunto. Si el conjunto está vacío, lanza excepción **KeyError**.
- `conjunto.clear()`: elimina todos los elementos de *conjunto* (queda vacío).

- Mientras se está **recorriendo** un conjunto, **no** se pueden **añadir** ni **eliminar** elementos.

```
nums = {1, 2, 3}
z = len(nums)    # 3
nums.add(4)      # {1, 2, 3, 4}
nums.discard(2) # {1, 3, 4}
nums.discard(5) # {1, 3, 4}
nums.remove(4)  # {1, 3}
x = nums.pop()  # {3} -> 1
nums.clear()     # {}
```

Conjuntos

Operaciones con conjuntos

- `resultado = conjunto_1 OP conjunto_2`: devuelve el resultado de la realizar la operación *conjunto_1 OP conjunto_2*.
- `conjunto OP= otro_conjunto`: asigna a *conjunto* el resultado de la realizar la operación con *conjunto OP otro_conjunto*.
- Disponemos de los siguientes operadores *(OP)* de conjuntos:
  - `|`: unión de conjuntos.
  - `&`: intersección de conjuntos.
  - `-`: diferencia de conjuntos.
  - `^`: diferencia simétrica de conjuntos.

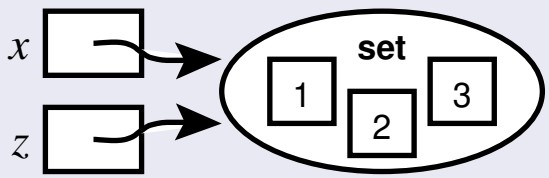
```
x = {1, 2, 3, 4}
y = {3, 4, 5, 6}
z = x | y  # {1, 2, 3, 4, 5, 6}
z = x & y  # {3, 4}
z = x - y  # {1, 2}
z = x ^ y  # {1, 2, 5, 6}
```

```
y = {3, 4, 5, 6}
x = {1, 2, 3, 4}
x |= y          # {1, 2, 3, 4, 5, 6}

x = {1, 2, 3, 4}
x &= y          # {3, 4}

x = {1, 2, 3, 4}
x -= y          # {1, 2}

x = {1, 2, 3, 4}
x ^= y          # {1, 2, 5, 6}
```

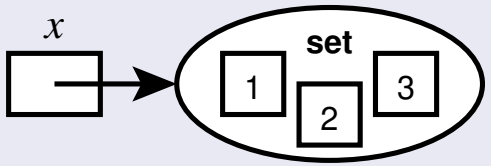
Duplicación y copia de conjuntos

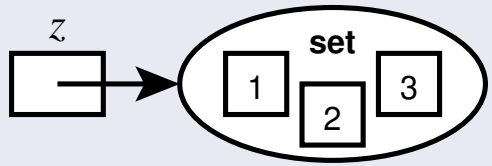
- Los conjuntos son objetos **mutables** (modificables).
- La asignación de variables asigna la referencia, pero el objeto es compartido.
- Si queremos realizar una modificación en un determinado objeto:
  - A veces queremos que la modificación se realice en el objeto compartido.
  - A veces queremos que la modificación se realice en otro objeto independiente, **copia** (duplicación) del original. `copy()` realiza una **copia superficial** del conjunto.

```
x = {1, 2, 3} # {1, 2, 3}
z = x   # Objeto compartido

z.add(4)
print(x)     # {1, 2, 3, 4}
print(z)     # {1, 2, 3, 4}
```




```
x = {1, 2, 3} # {1, 2, 3}
z = x.copy()  # copia superficial
z = set(x)    # copia superficial

z.add(4)
print(x)     # {1, 2, 3}
print(z)     # {1, 2, 3, 4}
```





Conjuntos. Ejemplo

Ejemplo

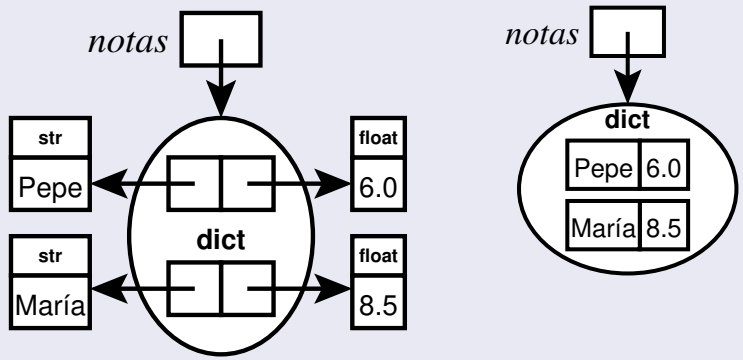
- Desarrolla un programa que lea de teclado una frase, compuesta por palabras separadas por espacios, y posteriormente muestre aquellas palabras que han sido repetidas, y aquellas palabras que no han sido repetidas.

```
def main() -> None:
    """Muestra las palabras repetidas y no repetidas de una frase"""
    frase = input("Introduce una frase: ")
    palabras_no_repetidas: set[str] = set()
    palabras_repetidas: set[str] = set()
    for palabra in frase.split():
        palabra = palabra.lower()
        if palabra in palabras_no_repetidas:
            palabras_no_repetidas.discard(palabra)
            palabras_repetidas.add(palabra)
        elif palabra not in palabras_repetidas:
            palabras_no_repetidas.add(palabra)
    print("Palabras con repeticiones: ", palabras_repetidas)
    print("Palabras sin repeticiones: ", palabras_no_repetidas)

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

Diccionarios

- Un diccionario es una colección iterable y **modificable** de pares de elementos (clave de acceso y valor asociado), que se utiliza como una **correspondencia** o **asociación** de claves con valores asociados. Su tipo es **dict**.
  - El diccionario está organizado para realizar **consultas eficientes** (muy rápidas), para saber si una clave se encuentra en el diccionario (operador `in`), y para poder acceder y/o modificar el valor asociado a una clave.
  - Las **claves** son **únicas**, no hay claves **repetidas**. Cuando se asigna un valor asociado a una clave, el valor anterior se desecha.
  - Las **claves** del diccionario deben ser **inmutables** (componentes utilizados en el hash).
  - Cada **valor** del diccionario está asociado con una clave de acceso.
  - Los **elementos** del diccionario **no** están asociados con ninguna posición determinada dentro del diccionario. Un diccionario **no** es una secuencia.

```
diccionario_vacio = dict()
diccionario_vacio = {}
notas = {"Pepe": (5.0, 7.0), "María": (8.0, 9.0)}
notas = {"Pepe": [5.0, 7.0], "María": [8.0, 9.0]}
notas = {"Pepe": 6.0, "María": 8.5}

print(notas)     # {'Pepe': 6.0, 'María': 8.5}
```



Diccionarios

- Se puede crear un diccionario con los valores de otro diccionario, o de un **iterable** de pares:

```
diccionario = dict(diccionario)
diccionario = dict(iterable_de_pares)
diccionario = dict([("Pepe", 6.0), ("María", 8.5)])
```

Diccionarios por comprensión (dictionary comprehensions)

- Los **diccionarios por comprensión** permiten crear diccionarios de forma concisa, iterando sobre los elementos de un *iterable*, seleccionando elementos según alguna condición, y transformando los elementos seleccionados:

```
diccionario = {x: x**2 for x in range(5) if x%2==0}
# {0: 0, 2: 4, 4: 16}

diccionario = {k:v for (k,v) in zip(["a", "b"], [1, 2])}
# {'a': 1, 'b': 2}
```

Diccionarios

Diccionarios iterables

- Se puede utilizar el bucle **for** para iterar sobre todos los elementos de un diccionario iterable.
  - El orden de iteración de los elementos coincide con el orden de inserción.

```
diccionario = {"pepe": 0, "maria": 4, "juan": 16}

for clave in diccionario:              # iteración sobre las claves del diccionario
    print(clave, end=' ')              # pepe maria juan
print()

for clave in diccionario.keys():       # iteración sobre las claves del diccionario
    print(clave, end=' ')              # pepe maria juan
print()

for (clave, valor) in diccionario.items(): # iteración sobre los elementos del diccionari
    print(f"{clave}: {valor}", end="; ")    # pepe: 0; maria: 4; juan: 16;
print()

for valor in diccionario.values():     # iteración sobre los valores del diccionario
    print(valor, end=' ')              # 0 4 16
print()
```

Diccionarios

Pertenencia de clave

- `clave in diccionario`: devuelve `True` si la *clave* es miembro de las claves del *diccionario*.
- `clave not in diccionario`: devuelve `True` si la *clave* no es miembro de las claves del *diccionario*.

Comparaciones

- `== !=`: devuelven el resultado de las comparaciones de igualdad o desigualdad. Comparan la igualdad de los elementos (*clave*: *valor*), el orden de los elementos no influye.

```
diccionario = {"pepe": 0, "maria": 4, "juan": 16}
ok = "pepe" in diccionario      # True
ok = "luis" in diccionario      # False
ok = "pepe" not in diccionario  # False
ok = "luis" not in diccionario  # True

dict2 = {"maria": 4, "pepe": 0, "juan": 16}
ok = diccionario == dict2       # True
ok = diccionario != dict2       # False
```

Diccionarios

Manipulación de elementos del diccionario

- `len(diccionario)`: devuelve el número de elementos (*clave*: *valor*) del *diccionario*.
- `diccionario[clave] = valor`: asigna el *valor* asociado a la *clave* en el *diccionario*, el valor anterior se desecha.
- `diccionario[clave]`: devuelve el valor asociado a la *clave* del *diccionario*. Lanza excepción `KeyError` si la clave no está.
- `diccionario.get(clave, default=None)`: devuelve el valor asociado a la *clave* del *diccionario*. Si la clave no está, devuelve el valor por *defecto* especificado.
- `diccionario.pop(clave [, default])`: devuelve y elimina el valor asociado a la *clave* del *diccionario*. Si la clave no está, devuelve el valor por *defecto* o lanza `KeyError`.
- `del diccionario[clave]`: elimina de *diccionario* el elemento (par *clave*: *valor*) correspondiente a la *clave* especificada. Lanza excepción `KeyError` si la clave no está.

```
diccionario = {"pepe": 0, "maria": 4}  # {"pepe": 0, "maria": 4}
diccionario["juan"] = 16               # {"pepe": 0, "maria": 4, "juan": 16}
diccionario["pepe"] = 2                # {"pepe": 2, "maria": 4, "juan": 16}
x = diccionario["maria"]               # 4
x = diccionario["luis"]                # Lanza KeyError
x = diccionario.get("maria")           # 4
x = diccionario.get("luis")            # None
x = diccionario.get("luis", 0)         # 0
x = diccionario.pop("pepe")            # {"maria": 4, "juan": 16} -> 2
del diccionario["juan"]                # {"maria": 4}
```

Diccionarios

Operaciones con diccionarios

- `diccionario.clear()`: elimina todos los elementos (*clave*: *valor*) del *diccionario*.
- `diccionario.update(diccionario_o_iterable_de_pares)`: añade/actualiza al diccionario todos los elementos del *diccionario* o *iterable* de pares.

- Mientras se está **recorriendo** un diccionario, **no** se pueden **añadir** ni **eliminar** claves.

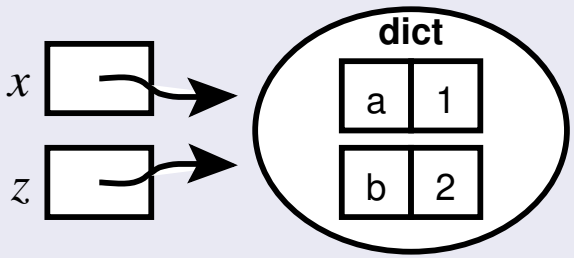
```
diccionario = {"pepe": 0, "maria": 4, "juan": 16}
dict2 = {"maria": 7, "luis": 9}
diccionario.update(dict2)       # {'pepe': 0, 'maria': 7, 'juan': 16, 'luis': 9}
diccionario.clear()             # {} diccionario vacío
```

Diccionarios

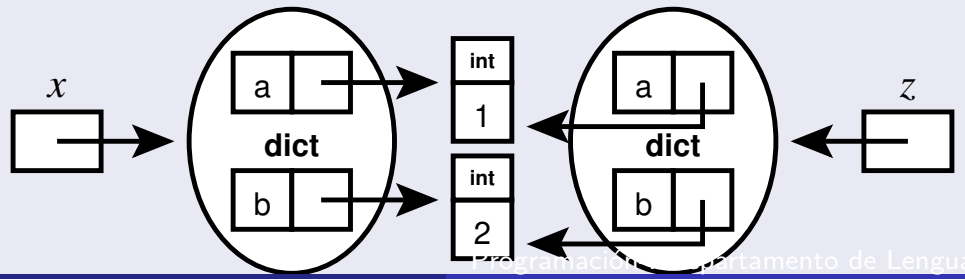
Duplicación y copia de diccionarios

- Los diccionarios son objetos **mutables** (modificables).
- La asignación de variables asigna la referencia, pero el objeto es compartido.
- Si queremos realizar una modificación en un determinado objeto:
  - A veces queremos que la modificación se realice en el objeto compartido.
  - A veces queremos que la modificación se realice en otro objeto independiente, **copia** (duplicación) del original. `copy()` realiza una **copia superficial** del diccionario.

```
x = {"a": 1, "b": 2} # {"a": 1, "b": 2}
z = x   # Objeto compartido

z["b"] = 7
print(x)     # {"a": 1, "b": 7}
print(z)     # {"a": 1, "b": 7}
```




```
x = {"a": 1, "b": 2} # {"a": 1, "b": 2}
z = x.copy()    # copia superficial
z = dict(x)     # copia superficial

z["b"] = 7
print(x)     # {"a": 1, "b": 2}
print(z)     # {"a": 1, "b": 7}
```



Diccionarios. Ejemplo-1

Ejemplo

- Desarrolla un programa que lea de teclado una frase, compuesta por palabras separadas por espacios, y posteriormente muestre cada palabra, y la cantidad de veces que ha sido repetida.

```
def main() -> None:
    """Muestra la cantidad de repeticiones de cada palabra en una frase"""
    frase = input("Introduce una frase: ")
    cuenta_palabras: dict[str,int] = dict()  # clave: palabra, valor: cuenta_repeticiones
    for palabra in frase.split():
        palabra = palabra.lower()
        if palabra not in cuenta_palabras:
            cuenta_palabras[palabra] = 0
        cuenta_palabras[palabra] += 1
    print("Cuenta palabras: ", cuenta_palabras)

    for (palabra, cuenta) in cuenta_palabras.items():
        print(f"{palabra}: {cuenta}")


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

Diccionarios. Ejemplo-2

Ejemplo

- Desarrolla un programa que lea de teclado una frase, compuesta por palabras separadas por espacios, y posteriormente muestre cada palabra, y las posiciones dentro de la frase donde ha aparecido.

```
def main() -> None:
    """Muestra las posiciones de cada palabra en una frase"""
    frase = input("Introduce una frase: ")
    posiciones_palabras: dict[str,list[int]] = dict()  # clave: palabra, valor: lista_posicio
    for (pos, palabra) in enumerate(frase.split(), start=1):
        palabra = palabra.lower()
        if palabra not in posiciones_palabras:
            posiciones_palabras[palabra] = list()
        posiciones_palabras[palabra].append(pos)
    print("Posiciones de palabras: ", posiciones_palabras)

    for (palabra, lista_pos) in posiciones_palabras.items():
        print(f"{palabra}: {lista_pos}")


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

Diccionarios. Ejemplo-3 (I)

Programa de notas de alumnos

- Lee el **nombre y la nota** de cada alumno, considerando que un alumno puede tener **múltiples** notas, y muestra el nombre y si está aprobado o suspenso, considerando que el alumno está aprobado si su nota media es mayor o igual **a la nota media de todos los alumnos**.
  - Se leerá el nombre un alumno junto con una determinada nota. Si el alumno tiene varias notas, cada nota, junto con su nombre, se leerá de forma independiente de las anteriores.
  - Se almacenará el nombre de cada alumno asociado a una lista con todas sus notas.

Diccionarios. Ejemplo-3 (II)

```python
from typing import Optional, NamedTuple
class Alumno(NamedTuple):
    nombre: str
    nota: float

def leer_alumno() -> Optional[Alumno]:
    """Lee los datos de un alumno."""
    res: Optional[Alumno] = None
    nombre = input("Introduce Nombre: ")
    if len(nombre) > 0:
        nota = float(input("Introduce nota: "))
        res = Alumno(nombre, nota)
    return res

def leer_alumnos() -> dict[str, list[float]]:
    """Lee los datos de varios alumnos."""
    dic_alumnos: dict[str,list[float]] = dict()  # clave: nombre_alumno, valor: lista_notas
    alumno = leer_alumno()
    while alumno is not None:
        if alumno.nombre not in dic_alumnos:
            dic_alumnos[alumno.nombre] = list()
        dic_alumnos[alumno.nombre].append(alumno.nota)
        alumno = leer_alumno()
    return dic_alumnos
```

Diccionarios. Ejemplo-3 (III)

```python
def calc_media_notas(lista_notas: list[float]) -> float:
    """Calcula la nota media de una lista de notas."""
    media = 0.0
    if len(lista_notas) > 0:
        suma = 0.0
        for nota in lista_notas:
            suma += nota
        media = suma / len(lista_notas)
    return media

def calc_media(dic_alumnos: dict[str, list[float]]) -> float:
    """Calcula la nota media del total de alumnos."""
    media = 0.0
    if len(dic_alumnos) > 0:
        suma = 0.0
        for lista_notas in dic_alumnos.values():
            suma += calc_media_notas(lista_notas)
        media = suma / len(dic_alumnos)
    return media
```

Diccionarios. Ejemplo-3 (IV)

```python
def mostrar_alumno(nombre: str, lista_notas: list[float], umbral: float) -> None:
    """Muestra la nota final de un alumno."""
    media_notas = calc_media_notas(lista_notas)
    if media_notas >= umbral:
        print(f"Alumno: {nombre} Nota Media: {media_notas} Aprobado")
    else:
        print(f"Alumno: {nombre} Nota Media: {media_notas} Suspenso")

def mostrar_alumnos(dic_alumnos: dict[str, list[float]], umbral: float) -> None:
    """Muestra la nota de los alumnos."""
    for (nombre, lista_notas) in dic_alumnos.items():
        mostrar_alumno(nombre, lista_notas, umbral)

def main() -> None:
    """Lee las notas de alumnos y muestra su nota final."""
    alumnos = leer_alumnos()
    media = calc_media(alumnos)
    mostrar_alumnos(alumnos, media)

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

Tabla resumen de estructuras de datos

Estructuras	tuple *(inmutable)*	list	set	dict
Consultar	**l = len(tupla)** **(e in tupla)** (e not in tupla)	**l = len(lista)** **(e in lista)** (e not in lista) **i = lista.index(e, f, t)** VE *(f t opcional)*	**l = len(conj)** **(e in conj)** (e not in conj)	**l = len(dicc)** **(k in dicc)** (k not in dicc)
Acceder	**tupla[i]** tupla[f:t:s] *(f t s opcional)*	**lista[i]** lista[f:t:s] *(f t s opcional)*		**dicc[k]** KE **v = dicc.get(k, v)** *(v opcional)*
Añadir	tupla += t2 *(nueva)* tupla = t1 + t2 *(nueva)*	**lista.append(e)** **lista.extend(itble)** lista.insert(i, e) lista += it lista = l1 + l2 *(nueva)*	**conj.add(e)**	**dicc[k] = v**
Eliminar		**lista.clear()** **lista.remove(e)** VE **del lista[i]** del lista[f:t:s] *(f t s opcional)* e = lista.pop(i)	**conj.clear()** **conj.remove(e)** KE **conj.discard(e)** e = conj.pop() KE	**dicc.clear()** **del dicc[k]** KE v = dicc.pop(k, v) KE *(v opcional)*
Modificar		lista.sort(key) *(key opcional)* lista.reverse() **lista[i] = e** lista[f:t:s] = itble	**conj = c1 | c2** *(nuevo)* conj |= c2 **conj = c1 & c2** *(nuevo)* conj &= c2 **conj = c1 – c2** *(nuevo)* conj –= c2 **conj = c1 ^ c2** *(nuevo)* conj ^= c2	dicc.update(otro) **dicc[k] = v**
Duplicar		lista = otra.copy()	conj = otro.copy()	dicc = otro.copy()
Iterar	**for e in tupla:** it = iter(tupla) e = next(it)	**for e in lista:** it = iter(lista) e = next(it)	**for e in conj:** it = iter(conj) e = next(it)	**for k in dicc:** **for k in dicc.keys():** **for (k, v) in dicc.items():** **for v in dicc.values():**

Leyenda: [e: Elemento] [i: Indice] [f: Desde] [t: Hasta] [s: Paso] [k: Clave] [v: Valor] [itble: Iterable] [it: Iterador] [VE: ValueError] [KE: KeyError]

Funciones incorporadas (I)

- `abs(x)`: devuelve el valor absoluto de un número (entero o real). Si es un número complejo, devuelve su magnitud.
- `round(numero, ndigitos)`: devuelve el valor de un número real, redondeado al número de dígitos especificado.
- `round(numero)`: devuelve el valor de un número real, redondeado al número entero más cercano.

- `type(valor)`: devuelve el tipo del *valor*.
- `isinstance(valor, tipo)`: devuelve `True` si *valor* es una instancia del *tipo* o *subclase*.
- `issubclass(clase, tipo)`: devuelve `True` si *clase* es *tipo* o una subclase de *tipo*.

```
x = abs(-5)               # 5
x = abs(-5.5)             # 5.5

x = round(5.4)            # 5
x = round(5.5)            # 6
x = round(-5.4)           # -5
x = round(-5.5)           # -6
x = round(3.3333333, 0)   # 3.0
x = round(3.3333333, 2)   # 3.33
x = round(3.5555555, 2)   # 3.56
```

```
x = type(5)               # <class 'int'> (tipo)
x = type(5) is int        # True
x = type(5).__name__      # "int" (string)

x = isinstance(5, int)                  # True
x = isinstance(5, (int, float))         # True
x = isinstance(False, bool)             # True
x = isinstance(False, int)              # True

x = issubclass(bool, (int, float)) # True
```

Funciones incorporadas (II)

- `sum(iterable)`: devuelve el valor de la suma de los elementos (números) del *iterable*.
- `max(..., *, key=None)`: devuelve el mayor elemento de los *valores* o *iterable*.
- `min(..., *, key=None)`: devuelve el menor elemento de los *valores* o *iterable*.

- `all(iterable)`: devuelve `True` si todos elementos del iterable son `True` (o vacío).
- `any(iterable)`: devuelve `True` si hay algún elemento `True` en el iterable.

```python
x = sum([1, 2, 3])      # 6

x = max(1, 2, 3)        # 3
x = max([1, 2, 3])      # 3

x = min(1, 2, 3)        # 1
x = min([1, 2, 3])      # 1
```

```python
def voltear(elem: tuple) -> tuple:
    return (elem[1], elem[0])

x = max([(1, 9), (2, 7)])                # (2, 7)
x = max([(1, 9), (2, 7)], key=voltear) # (1, 9)

x = min([(1, 9), (2, 7)])                # (1, 9)
x = min([(1, 9), (2, 7)], key=voltear) # (2, 7)
```

```python
x = all(x > 42 for x in it)     # True si todos los elementos de it son mayores que 42
x = not all(x > 42 for x in it) # True si algún elemento de it NO es mayor que 42
x = any(x > 42 for x in it)     # True si algún elemento de it SÍ es mayor que 42
x = not any(x > 42 for x in it) # True si ningún elemento de it es mayor que 42
```

Funciones incorporadas (III)

- `sorted(iterable, *, key=None, reverse=False)`: devuelve nueva *lista* ordenada.
- `reversed(secuencia)`: devuelve un *iterador* que permite iterar la *secuencia* al revés.
- `zip(*iterables)`: devuelve un *iterador* con los elementos tuplas con cada elemento de los *iterables*.

```
z = sorted([3, 1, 2])                                  # [1, 2, 3]
z = sorted([3, 1, 2], reverse=True)                    # [3, 2, 1]
z = sorted([(3, 8), (1, 9), (2, 7)])                   # [(1, 9), (2, 7), (3, 8)]

z = sorted([(3, 8), (1, 9), (2, 7)], key=voltear)      # [(2, 7), (3, 8), (1, 9)]

z = [x for x in reversed([4, 2, 3])]                   # [3, 2, 4]
z = [x for x in zip([4, 2, 3], [1, 2, 3])]             # [(4, 1), (2, 2), (3, 3)]
```

Funciones anónimas. Expresiones lambda

- Las expresiones **lambda** son usadas para crear *funciones anónimas*.
  - Las funciones anónimas (*lambda*) pueden ser **usadas** en cualquier lugar donde sea requerido un objeto de tipo función.
  - Están restringidas a una definir **una sola expresión**, no pueden contener sentencias ni anotaciones.
  - Pueden utilizar los **parámetros** y las **variables** del ámbito que la contiene.
  - La siguiente expresión:

    ```
    lambda parametros: expresion
    ```

    produce un objeto de función anónima (sin nombre) que se comporta como una función que hubiese sido definida de la siguiente forma:

    ```
    def <lambda>(parametros):
        return expresion
    ```

- Utilizaremos las expresiones *lambda* en muy pocas ocasiones:

```
lista = [(3, 8), (1, 9), (2, 7)]
lista.sort(key=lambda elem: (elem[1], elem[0]))        # [(2, 7), (3, 8), (1, 9)]
z = sorted(lista, key=lambda elem: (elem[1], elem[0])) # [(2, 7), (3, 8), (1, 9)]
x = min(lista, key=lambda elem: (elem[1], elem[0]))    # (2, 7)
x = max(lista, key=lambda elem: (elem[1], elem[0]))    # (1, 9)

lista = ["PEPE", "ana", "Luis"]
lista.sort(key=lambda elem: elem.upper())        # ["ana", "Luis", "PEPE"]
z = sorted(lista, key=lambda elem: elem.upper()) # ["ana", "Luis", "PEPE"]
x = min(lista, key=lambda elem: elem.upper())    # "ana"
x = max(lista, key=lambda elem: elem.upper())    # "PEPE"
```

Módulo de funciones matemáticas

Algunas funciones del módulo **math**	Significado
`def isclose(x: float, y: float) -> bool:`	True si x e y son **aproximadamente iguales**
`def hypot(x: float, y: float) -> float:`	hipotenusa de x e y ($\equiv \sqrt{x^2 + y^2}$)
`def sqrt(x: float) -> float:`	raiz cuadrada de x, $\sqrt{x}$, $x \geq 0$
`def cbrt(x: float) -> float:`	raiz cúbica de x, $\sqrt[3]{x}$
`def pow(x: float, y: float) -> float:`	x^y
`def exp(x: float) -> float:`	e^x
`def exp2(x: float) -> float:`	2^x
`def log(x: float) -> float:`	logaritmo natural, $ln(x)$, $x > 0$
`def log2(x: float) -> float:`	logaritmo binario, $log_2(x)$, $x > 0$
`def log10(x: float) -> float:`	logaritmo decimal, $log_{10}(x)$, $x > 0$
`def ceil(x: float) -> float:`	menor entero $\geq x$, $\lceil x \rceil$
`def floor(x: float) -> float:`	mayor entero $\leq x$, $\lfloor x \rfloor$
`def trunc(x: float) -> float:`	valor entero de x, sin decimales
`def round(x: float) -> float:`	valor entero más cercano a x
`def fabs(x: float) -> float:`	valor absoluto de x, $\|x\|$
`def fmod(x: float, y: float) -> float:`	resto de x / y
`def sin(r: float) -> float:`	seno, *sin(r)* (en radianes)
`def cos(r: float) -> float:`	coseno, *cos(r)* (en radianes)
`def tan(r: float) -> float:`	tangente, *tan(r)* (en radianes)
`def asin(x: float) -> float:`	arco seno, *arcsin(x)*, $x \in$ *[-1,1]*
`def acos(x: float) -> float:`	arco coseno, *arccos(x)*, $x \in$ *[-1,1]*
`def atan(x: float) -> float:`	arco tangente, *arctan(x)*
`def atan2(y: float, x: float) -> float:`	arco tangente, *arctan(y/x)*
`def degrees(r: float) -> float:`	convierte de radianes *(r)* a grados
`def radians(g: float) -> float:`	convierte de grados *(g)* a radianes
`def factorial(x: int) -> int:`	factorial de x ($\equiv x!$)
`def gcd(x: int, ...) -> int:`	máximo común divisor de los argumentos enteros
`def lcm(x: int, ...) -> int:`	mínimo común múltiplo de los argumentos enteros
`def prod(x: float, ...) -> float:`	producto de los argumentos
`def comb(x: int, y: int) -> int:`	coef. binomial de x sobre y ($\equiv \frac{x!}{y! \cdot (x-y)!}$)
`def perm(x: int, y: int) -> int:`	permutaciones de x sobre y ($\equiv \frac{x!}{(x-y)!}$)
Constantes: `pi`, `e`	$pi = 3.141592653589793$; $e = 2.718281828459045$

Comparación de rendimiento (I)

- A veces disponemos de diferentes códigos alternativos que realizan una misma tarea, y deseamos comparar el **rendimiento** que ofrecen.
  - El módulo **timeit** permite medir el tiempo de ejecución de diferentes tareas, dentro de un entorno controlado para ello (desactiva el recolector de basura).

```python
import timeit
import gc

def crear_datos() -> list[int]:
    return list(range(10000))

def buscar_01(lista: list[int], elem: int) -> int:
    if elem in lista:
        pos = lista.index(elem)
    else:
        pos = -1
    return pos

def buscar_02(lista: list[int], elem: int) -> int:
    pos = 0
    while ((pos < len(lista))
           and (elem != lista[pos])):
        pos += 1
    return pos if pos < len(lista) else -1
```

```python
def buscar_03(lista: list[int], elem: int) -> int:
    pos = -1
    idx = 0
    while ((idx < len(lista)) and (pos < 0)):
        if (elem == lista[idx]):
            pos = idx
        idx += 1
    return pos

def buscar_04(lista: list[int], elem: int) -> int:
    try:
        pos = lista.index(elem)
    except ValueError:
        pos = -1
    return pos

def buscar_05(lista: list[int], elem: int) -> int:
    elementos = [(i, x) for (i, x) in enumerate(lista) if x == elem]
    return elementos[0][0] if len(elementos) > 0 else -1
```

Comparación de rendimiento (II)

```
def main():
    funciones = [buscar_01, buscar_02, buscar_03, buscar_04, buscar_05]

    for funcion in funciones:
        gc.collect()      # recolector de basura

        print(funcion.__name__, "(Enc): ", timeit.timeit(f"{funcion.__name__}(datos, 9999)",
                                                         setup="datos = crear_datos()",
                                                         number=100000,
                                                         globals=globals()))

        gc.collect()      # recolector de basura

        print(funcion.__name__, "(NEnc):", timeit.timeit(f"{funcion.__name__}(datos, 99999)",
                                                         setup="datos = crear_datos()",
                                                         number=100000,
                                                         globals=globals()))


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

```
buscar_01 (Enc):  10.862614524085075
buscar_01 (NEnc):  5.309606962837279
buscar_02 (Enc):  53.17623388301581
buscar_02 (NEnc): 52.303601005114615
buscar_03 (Enc):  59.44733107299544
buscar_03 (NEnc): 59.46225355099887
buscar_04 (Enc):   5.566075945040211
buscar_04 (NEnc):  5.6156813281122595
buscar_05 (Enc):  28.45722083095461
buscar_05 (NEnc): 28.751606370089576
```

Comparación de rendimiento (III)

- A veces disponemos de diferentes códigos alternativos que realizan una misma tarea, y deseamos comparar el **rendimiento** que ofrecen.
  - El módulo **timeit** permite medir el tiempo de ejecución de diferentes tareas, dentro de un entorno controlado para ello (desactiva el recolector de basura).

```
import gc
import timeit

def crear_datos() -> list[int]:
    return list(range(10000))

def eliminar_pares_1(lista: list[int]) -> None:
    idx_dst = 0
    for elemento in lista:
        if elemento % 2 != 0:
            lista[idx_dst] = elemento
            idx_dst += 1
    del lista[idx_dst:]

def eliminar_pares_2(lista: list[int]) -> None:
    lista[:] = [elemento for elemento in lista if elemento % 2 != 0]

def eliminar_pares_3(lista: list[int]) -> None:
    lista[:] = (elemento for elemento in lista if elemento % 2 != 0)
```

Comparación de rendimiento (IV)

```
def main():
    funciones = [eliminar_pares_1, eliminar_pares_2, eliminar_pares_3]

    for funcion in funciones:
        gc.collect()     # recolector de basura

        print(funcion.__name__, ":", timeit.timeit(f"{funcion.__name__}(datos)",
                                                   setup="datos = crear_datos()",
                                                   number=100000,
                                                   globals=globals()))

    gc.collect()     # recolector de basura


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

```
eliminar_pares_1 : 24.284139893949032
eliminar_pares_2 : 18.551311956020072
eliminar_pares_3 : 24.39300862303935
```

Herramientas de desarrollo (I)

Entornos de desarrollo

- **Thonny**: `https://thonny.org/` (lo utilizaremos durante el curso)
- VsCodium: `https://vscodium.com/`

Analizadores de código

- **Pylint**: `https://pylint.readthedocs.io/` (incorporado en Thonny)
- **MyPy**: `https://www.mypy-lang.org/` (incorporado en Thonny)
  - Es necesaria su activación en el entorno de desarrollo Thonny
  - Módulo **typing**: *forma parte de la biblioteca estándar de Python* (tema 1).

Generador de documentación

- Módulo **pydoc**: *forma parte de la biblioteca estándar de Python* (tema 1).

```
>>> import pydoc
>>> import dato
>>> help(dato)              # muestra la documentación en consola
>>> pydoc.writedoc(dato)    # crea documento HTML con la documentación
```

- Generación de diagramas UML
  - **pyreverse**: `https://pylint.readthedocs.io/`
  - **plantuml**: `https://plantuml.com/`

Herramientas de desarrollo (II)

Registro de errores

- Módulo **logging**: *forma parte de la biblioteca estándar de Python* (tema 2).

Pruebas unitarias

- Módulo **unittest**: *forma parte de la biblioteca estándar de Python* (tema 6).
- Módulo coverage: `https://coverage.readthedocs.io/`.
- pytest: `https://docs.pytest.org/`

Distribución e instalación de paquetes

- Gestor de paquetes del entorno de desarrollo Thonny
- pip: `https://pip.pypa.io/`
- anaconda: `https://www.anaconda.com/`

Sistema de control de versiones

- Git: `https://git-scm.com/`