UNIVERSIDAD DE MURCIA

Tecnología de la Programación

Sesión 1 de Prácticas

Introducción a Python

Dpto. de Ingeniería de la Información y las Comunicaciones

Curso 2025-2026

Última modificación:
5 de septiembre de 2025

Sesión 1: Introducción a Python

Índice

1.1 Introducción

Python es un lenguaje de programación de propósito general escrito sobre el lenguaje C. Se usa en una amplia variedad de disciplinas como biología, finanzas, química, análisis numérico, inteligencia artificial, etc. También se usa como lenguaje para crear scripts por parte de los administradores de sistemas informáticos.

En esta primera sesión práctica se ofrece un repaso general de cómo realizar programación imperativa en Python, mediante el uso de variables, colecciones, bloques condicionales, bucles y funciones.

1.2 Programación imperativa en Python

Esta sección ofrece un resumen de los aspectos más relevantes del lenguaje de programación Python, siguiendo un enfoque de programación imperativa. Algunos aspectos más avanzados se pueden consultar en los anexos de esta sesión de prácticas.

1.2.1. Variables, expresiones y conversión de tipos

Tipos de datos. En programación imperativa se distinguen dos tipos de datos: elementales y compuestos.

- **Tipos de datos primitivos (o elementales)**
  - Formado por un único elemento.
  - Tipos: carácter, numérico, booleano, enumerado.
- **Tipos de datos compuestos**
  - Formado por una agrupación de elementos.
  - Tipos: string, array, registro, conjunto, lista, diccionario.

Variable y expresiones. Es importante enfatizar que declararemos las variables indicando de forma explícita su tipo de datos con el operador : y, tras ello, las asignaciones de valores se realizarán con =. Además, las variables usan la convención de nombres snake_case y cuando se quiere considerar que una variable es una constante su identificador tiene todos sus caracteres en mayúsculas. A continuación se muestra un ejemplo de representación para algunos de los tipos de datos más relevantes:

```
numero_entero: int = 2
numero_real: float = 10.01
caracter: str = '2'             # Se interpreta como un string
booleano: bool = True
string: str = " una cadena "    # Se interpreta como un string

# Asignación múltiple
numero_entero, numero_real, booleano= 2, 10.01, False

# Destrucción de variables
del numero_entero   # No es usual esta instrucción

# Las "constantes" se escriben en mayúsculas
PI: float = 3.1415      # La "constante" PI en Python. Ojo, puede cambiar su valor.
```

1.2.1.1. Tipos de datos primitivos

Se distinguen dos tipos de expresiones principales para trabajar con tipos de datos primitivos:

- Expresiones numéricas
  - Aritméticas: suma (+), resta y negación (-), multiplicación (*), división (/), división entera (//). módulo (%), exponente (**).
- Expresiones booleanas
  - Comparaciones: x == y, x != y, x > y, x >= y, x < y, x <= y
  - Operaciones: e1 and e2, e1 or e2, not e

Los resultados de las expresiones se guardan mediante la asignación. Las asignaciones no se consideran formalmente como expresiones, pero sí las construyen. P.e. x = x/10 ≡ x /= 10

Las asignaciones que usa Python son: =, +=, -=, *=, /=, %=, //=, **= (algunas son solo para expresiones numéricas).

Precedencia. El orden de precedencia de los operadores en Python se pueden consultar en la Figura 1.1.

Mutabilidad y casting. Una variable es mutable si se puede cambiar el valor de la variable sin cambiar su referencia en memoria.

La instrucción `id`(var) muestra la referencia de la variable `var`. Esta instrucción nos permite comprobar la mutabilidad de una variable. Si se hacen dos asignaciones a una variable e `id`() cambia, la variable es inmutable. Números, booleanos y strings son inmutables.

El casting es el proceso por el que el valor de una variable se interpreta como otro tipo de dato. Por ejemplo, tenemos las funciones `int`(), `float`(), `str`() para realizar un casting explícito a entero, real y string, respectivamente.

Un caso particular de tipo primitivo son los enumerados, que corresponden a una forma de representar un conjunto de valores constantes y simbólicos agrupados bajo un nombre común. Es decir, imagina un enumerado como un conjunto de opciones predefinidas que podemos usar en nuestro programa, donde cada opción tiene un nombre claro y un valor asociado. Por ejemplo, un enumerado podría representar los días de la semana, una lista de colores a usar en nuestra paleta de colores, etc. A continuación se muestra cómo definir y usar un enumerado en Python. En concreto, dia_elegido corresponderá a LUNES (que internamente se codifica con un 1). Podemos acceder a sus propiedades con .name y .value.

```
from enum import Enum

# Definición de un enumerado para los días de la semana
class DiaSemana(Enum):
    LUNES: int = 1
    MARTES: int = 2
    MIERCOLES: int = 3
    JUEVES: int = 4
    VIERNES: int = 5
    SABADO: int = 6
    DOMINGO: int = 7

# Ejemplo de asignación de un valor
dia_elegido: DiaSemana = DiaSemana.LUNES

# Mostrar el valor del día almacenado
print(dia_elegido.name)  # Salida: LUNES
```

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	**not**	Unario	—	6
Conjunción	**and**	Binario	Por la izquierda	7
Disyunción	**or**	Binario	Por la izquierda	8

Figura 1.1: Orden de Precedencia Fuente: `Introducción a la Programación con Python` (página 37)

```
print(dia_elegido.value) # Salida: 1
```

Además, también podemos comparar el valor de los enumerados usando los operadores de igualdad y desigualdad booleanos:

```
if DiaSemana.LUNES == DiaSemana.MARTES:
    print("Son iguales")
else:
    print("Son diferentes")  # Salida: Son diferentes
```

Es interesante destacar que podemos definir los enumerados de una forma un poco más específica. Si los valores del enumerado van a ser enteros (como es el caso del ejemplo anterior para los días de la semana), podemos usar IntEnum en lugar de Enum. Si por el contrario van a ser cadenas de texto, entonces podemos usar StrEnum. En cualquier caso, podemos usar Enum directamente, al ser una forma más general de representación.

1.2.1.2. Tipos de datos compuestos

En relación a los tipos (o estructuras) de datos compuestos, podemos clasificarlos en mutables e inmutables:

- Las **estructuras mutables** son aquellas cuyo estado puede cambiar una vez creadas.
- Las **estructuras inmutables** son aquellas cuyo estado no puede cambiar una vez creadas.

Python integra las siguientes estructuras de datos compuestas:

- Estructuras mutables. Se pueden cambiar sus términos una vez creada.
  - Listas: **secuencia** de elementos arbitrarios. Se escriben entre corchetes y se separan con comas.
    Ej: [1, "hola", 3].
  - Conjuntos: colección de elementos arbitrarios **únicos**. Se escriben entre llaves y se separan con comas.
    Ej: {1, "hola", 3}.
  - Diccionarios: conjuntos con objetos indexados. Cada elemento consta de un par clave: valor. La clave puede ser cualquier valor inmutable.
    Ej: {"a": 1, "b": "hola", "c": 3}.
- Estructuras inmutables. No puede cambiar sus términos una vez creada.

- Strings: **secuencia** de valores que representan códigos Unicode. Se escriben entre comillas.
  Ej: “hola”, ’adiós’
- Tuplas: **secuencia** de elementos arbitrarios. Se escriben entre paréntesis y se separan con comas.
  Ej: (1, "hola", 3).
- Rangos: **secuencias** que se construyen con range([start,] stop [[, step]]).
  Ej: range(1, 10, 2).
- Conjuntos congelados (Frozensets). La versión inmutable de los conjuntos.
  frozenset({1, "hola", 3})

Las secuencias o tipos de datos secuenciales (o en secuencia) representan a colecciones finitas de datos referenciados por un índice. Alternativamente, una secuencia esta formada por una colección de objetos, o términos, indexados con índices 0, 1, 2 ... Es importante destacar que todos aquellos tipos de datos que se basan en secuencias (tuplas, rangos, listas) tienen una serie de propiedades comunes que nos permiten interactuar con sus elementos:

Se puede acceder a sus elementos de distintas formas:

- s[i], el elemento i-esimo de s,
  Cuenta +1 desde el 0 si $i \geq 0$, o cuenta -1 desde la len(s)-i si $i < 0$.
- s[i:j], la rebanada (slice) de s desde i hasta j. Selecciona todos los elementos con índice t tal que $i \leq t < j$.
- Algunas secuencias también admiten la **“división ampliada”** con un tercer parámetro de “paso”: [i:j:k] selecciona todos los elementos con índice t donde $t = i + n \times k$, $0 \leq n$ e $i \leq t < j$.

Funciones comunes

- len(s), que devuelve el número de elementos de una secuencia.
  Si len(s)==n, el conjunto de índices es $\{0, 1, \ldots, n-1\}$.
- min(s) / max(s): el item más pequeño/grande de s.
- sum(s): la suma de los elementos de s.
- sorted(s): una lista con los elementos ordenados de s.
- enumerate(s): un enumerado de la lista s.
- any(s): devuelve True si bool(x) es True para cualquiera de los elementos de la secuencia.
- all(s): devuelve True si bool(x) es True para todos los elementos de la secuencia.

Métodos comunes

- s.count(x): el número total de ocurrencias de x en s.
- s.index(x[, i[, j]]): el índice de x en s [después de i [y antes de j]]

Operaciones comunes en estos elementos son:

- `x in s`: Es True si `x` es un item de `s`.
- `x not in s`: Es True si `x` no es un item de `s`.
- `s + t`: concatena `s` y `t`.
- `n * s`: añade `s` un total de `n`-veces (en algunas secuencias).

A continuación se muestra en detalle cada uno de los tipos compuestos enumerados previamente.

Strings. Construcción: "", ”, "cadena", ’cadena’, str(), str("cad").

Un **string** es una cadena o secuencia de valores que representan códigos Unicode y se escriben entre comillas dobles. Python no tiene un tipo char y en su lugar cada carácter se representa como un objeto string con longitud 1.

Funciones asociadas a los caracteres son:

- ord(char): retorna el código decimal de un char.
- chr(codigo): retorna el char dado el código (número natural) con la función.

Son métodos de las cadenas:

- .lower()/.upper(): Retorna una copia de la cadena de caracteres con todas las letras en **minúsculas / mayúsculas**.
- .capitalize(): Retorna una copia de la cadena con el **primer carácter** en mayúsculas y el resto en minúsculas.
- .title(): Igual pero el primer carácter **de cada palabra** de la cadena.
- .casefold(): Retorna el texto de la cadena, normalizado a minúsculas. Los textos **normalizados** pueden usarse para realizar búsquedas textuales independientes de mayúsculas, minúsculas y caracteres idiomáticos.
- .count(sub[, start[, end]]): Retorna el número de **ocurrencias** no solapadas de la cadena sub en el rango [start, end].
- .find(sub[, start[, end]]): Retorna el menor índice de la cadena s donde se puede **encontrar** la cadena sub, considerando solo el intervalo s[start:end]

- .split(sep=None, maxsplit=-1): Retorna los distintos substring comprendidos entre dos separadores. El separador es el valor de sep. El parámetro maxsplit indica el máximo número de divisiones. Si no se indica ningún separador elimina todos los espacios y devuelve sólo las palabras que conforman la cadena.
- .join(lista): retorna un string formado por los elementos de la lista.

Hay muchas más funciones en https://docs.python.org/es/3/library/stdtypes.html#text-sequence-type-str.

```
# Definir la cadena de texto con espacios
cadena: str = ' tengo espacios '

# Mostrar la cadena y su longitud
print(cadena, "len=", len(cadena)) # Salida: " tengo espacios  len= 16"

# Dividir la cadena en una lista
cadena_split: list[str] = cadena.split()

# Unir una lista de palabras con la cadena original
cadena_join: str = cadena.join(['ahora', 'extras'])

# Definir las variables enteras
i: int = 2
j: int = 11
k: int = 4

# Mostrar diferentes slices de la cadena:

# Mostrar el elemento en la posición i=2.
print(cadena[i]) # Salida: e
# Mostrar los elementos entre i=2 hasta 10<j=11
print(cadena[i:j]) # Salida: engo espa
# Mostrar los elementos en saltos de k=4
print(cadena[i:j:k]) # Salida: e a
# Mostrar el elemento en la posición i=-2 (equivalente a posición 15)
print(cadena[-i]) # Salida: s
# Mostrar los elementos entre j=-11 (posición 1) hasta i=-2 (sin incluir)
print(cadena[-j:-i]) # Salida: o espacio
# Mostrar los elementos de forma inversa desde j=11 hasta i=2 con saltos de k=4
print(cadena[j:i:-k]) # Salida: cen
```

Listas. Construcción: [], [x, y, ...], list() o list(iterable). *Por comprehensión en Anexo III*

Una lista es una secuencia ordenada de elementos arbitrarios que es mutable.

El constructor list() construye una lista vacía. También se construye con un par de corchetes. Los demás constructores crean listas con elementos.

```
# Definir la lista de cadenas
lista: list[str] = ['uno', 'dos', 'tres']

# Mostrar la lista y su longitud
print(lista, "len=", len(lista)) # Salida: ['uno', 'dos', 'tres'] len= 3
```

En el caso de tener una lista donde todos los elementos son del mismo tipo, lo indicaremos en la declaración explícita de la lista. Si contiene diferentes elementos, la declaración explícita indicará únicamente "list". También se puede usar el operador * si deseas repetir una lista varias veces.

```
# Definir las listas de cadenas
l1: list[str] = ['uno', 'dos']
l2: list[str] = ['tres', 'cuatro']

# Concatenar las dos listas: ['uno', 'dos', 'tres', 'cuatro']
resultado_concatenacion: list[str] = l1 + l2

# Repetir la lista l1 tres veces: ['uno', 'dos', 'uno', 'dos', 'uno', 'dos']
resultado_repeticion: list[str] = 3 * l1
```

Dispone de las operaciones/funciones:

- s[i] = x. El elemento i de s es reemplazado por x.
- s[i:j] = t. La rebanada de valores de s que van de i a j es reemplazada por el contenido del iterador t.
- del s[i:j]. Equivalente a s[i:j] = [].
- s[i:j:k] = t. Los elementos de s[i:j:k] son reemplazados por los elementos de t.
- del s[i:j:k]. Borra los elementos de s[i:j:k] de la lista.

Y los siguientes métodos:

- .append(x) **Añade** un valor. Alternativametne lista = l1 + l2 concatena listas.

- .insert(i, x), **inserta** un ítem en una posición dada. Ej: [0, 1, 2].insert(1, 10) == [0, 10, 1, 2]
- .extend(iterable), **añade** cualquier objeto iterable a la lista.
- .pop([x]), **extrae y quita** el último elemento o el elemento indicado.
- .remove(x), **elimina** el primer elemento que sea igual a x.
- .index(x[,start[,end]]), **indica** el índice del elemento x en el rango dado.
- .sort(*, reverse=False), **ordena** los elementos de la lista in situ.

```
# Crea la lista de cadenas y números
lista: list = ['uno', 'dos', 'tres']
print(lista) # Salida: ['uno', 'dos', 'tres']

# Añade elementos
lista = lista + [4]
print(lista) # Salida: ['uno', 'dos', 'tres', 4]

lista.append(5)
print(lista) # Salida: ['uno', 'dos', 'tres', 4, 5]

lista.insert(0, 'cero')
print(lista) # Salida: ['cero', 'uno', 'dos', 'tres', 4, 5]

# Elimina elementos
lista.remove('dos')  # Por valor
print(lista) # Salida: ['cero', 'uno', 'tres', 4, 5]

del lista[0]  # Por índice
print(lista) # Salida: ['uno', 'tres', 4, 5]

lista.pop()  # Extrae y quita el último elemento
print(lista) # Salida: ['uno', 'tres', 4]

lista.pop(1)  # Extrae y quita el elemento en la posición 1
print(lista) # Salida: ['uno', 4]

# Encontrar elementos
indice_uno: int = lista.index('uno')  # Buscar el índice de 'uno'
print("Indice de 'uno':", indice_uno) # Salida: Indice de 'uno': 0

# Verificar si un elemento está en la lista
esta_tres: bool = 'tres' in lista
esta_4: bool = 4 in lista

print("'tres' está en la lista:", esta_tres) # Salida: 'tres' está en la lista: False
print("4 está en la lista:", esta_4) # Salida: 4 está en la lista: True
```

Ten en cuenta que los elementos de una lista son arbitrarios, por lo que puede contener elementos de distintos tipos:

```
# Crea una lista que contiene cadenas, enteros, y una lista anidada
lista: list = ['uno', 2, ['tres', 4]]

# Mostrar la lista completa
print(lista) # Salida: ['uno', 2, ['tres', 4]]

# Acceder al segundo elemento de la lista (un entero)
print(lista[1]) # Salida: 2

# Acceder al tercer elemento de la lista (una lista anidada)
print(lista[2]) # Salida: ['tres', 4]

# Acceder al primer elemento de la lista anidada
print(lista[2][0]) # Salida: tres
```

Por simplicidad, cuando queramos trabajar con una lista con varios tipos de elementos, la definiremos de tipo `list`. Si quisiésemos ser puristas, debería tener tipo `list[object]`, pues reflejará cualquier tipo de datos. No obstante, nos abstraeremos en la asignatura por simplicidad.

Tuplas. Construcción: (x, y, ...) o tuple(iterable). *Por comprehensión en Anexo III.*

La versión inmutable de las listas son las tuplas. Una tupla es una secuencia ordenada de elementos arbitrarios. Se declara con apertura de paréntesis, seguido de los elementos separados por comas y cierre de paréntesis. Una tupla vacía está formada por un par de paréntesis. Una vez creada no se puede modificar pero sí se puede consultar su contenido usando índices o el operador in.

Cuando realicemos una declaración explícita de una tupla, por simplicidad indicaremos únicamente "tuple", ya que si no tendríamos que especificar el tipo de cada uno de los elementos que componen la tupla.

Las tuplas son más rápidas que las listas y como solo se puede iterar entre ellos, es mejor usar una tupla que una lista si se va a trabajar con un conjunto estático de valores. Se pueden convertir tuplas en listas y viceversa.

- tuple() recibe como parámetro una lista y devuelve una tupla.
- list() recibe como parámetro una tupla y devuelve una lista.

```
# Definir una tupla de cadenas
tupla: tuple = ('uno', 'dos', 'tres')
print(type(tupla)) # Salida: <class 'tuple'>

# Definir una lista de cadenas
lista: list = ['uno', 'dos', 'tres']
print(type(lista)) # Salida: <class 'list'>

# Convertir la lista en una tupla
t: tuple = tuple(lista)
print(type(t)) # Salida: <class 'tuple'>

# Convertir la tupla en una lista
l: list = list(tupla)
print(type(l)) # Salida: <class 'list'>

# Mostrar la tupla y su longitud
print(tupla, "len=", len(tupla)) # Salida: ('uno', 'dos', 'tres') len= 3

# Verificar si el valor 2 está en la tupla
print(2 in tupla) # Salida: False

# Acceder al segundo elemento de la tupla
print(tupla[1]) # Salida: dos
```

Rangos. Construcción: range([start,] stop[, step])

El tipo range representa una secuencia inmutable de números y se usa usualmente para bucles que se deben de ejecutar un número de veces (bucles for). Un rango se construye con uno, dos o tres naturales:

- range(stop) construye un rango que empieza en 0 y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- range(start, stop) construye un rango que empieza en `start` y finaliza en `stop`. Cada elemento de la secuencia se diferencia del anterior en una unidad.
- range(start, stop, step) construye un rango que empieza en `start`, finaliza en `stop` y cada elemento de la secuencia se diferencia del anterior en una `step`-unidades.

```
tuple(range(10))
tuple(range(10, 100, 20))
tuple(range(100, 10, -20))
```

Nota *"Los rangos implementan todas las operaciones comunes de las secuencias, excepto la concatenación y la repetición. La ventaja de usar un objeto de tipo range en vez de uno de tipo list o tuple es que con range siempre se usa una cantidad fija (y pequeña) de memoria, independientemente del rango que represente (Ya que solamente necesita almacenar los valores para start, stop y step, y calcula los valores intermedios a medida que los va necesitando)".*

Conjuntos. Construcción: {}, {x, y, ...}, set() o set(iterable). *Por comprehensión en Anexo III.*

Estos tipos de datos representan a conjuntos no ordenado de elementos (objetos únicos). Sus términos no pueden ser indexados por ningún subíndice. La función incorporada len() devuelve el número de elementos en un conjunto. Se utilizan para pruebas rápidas de pertenencia, la eliminación de elementos duplicados de una secuencia y el cálculo de operaciones matemáticas como la intersección, unión, diferencia y diferencia simétrica. Se tienen dos tipos de conjuntos:

- Conjuntos (Set)
  Representan un conjunto mutable. Se declara con apertura de llaves, los elementos separados por comas y cierre de llaves. Usar solo la apertura y cierre de llaves sin elementos define un diccionario, no un conjunto. Para definir un conjunto vacío usa el constructor set().
- Conjuntos congelados (Frozensets)
  Representan un conjunto inmutable. Se crean con el constructor frozenset().

```
# Definir un conjunto mutable de cadenas
conjunto_mutable: set = {"Perl", "Python", "Java"}
print(conjunto_mutable)

# Crear un conjunto mutable a partir de una tupla
```

```
conjunto_mutable_desde_tupla: set = set(("Perl", "Python", "Java"))
print(conjunto_mutable_desde_tupla)

# Convertir el conjunto mutable en un conjunto inmutable (frozenset)
conjunto_inmutable: frozenset = frozenset(conjunto_mutable)
print(conjunto_inmutable)
```

Nota *Los conjuntos no pueden estar formados por elementos mutables.*

```
set( {1: 2, 3: 4} )
set( [1   , "h", (0, (3, 4)) ] ); set((0, 1)) # ok
set(([0], [1])) # no ok. Los elementos son listas y éstas son mutables.
```

Algunos operadores que se tienen con conjuntos son:

- conjA - conjB realiza la diferencia del conjA con el conjunto conjB.
- conjA -= conjB calcula la diferencia del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA | conjB calcula la unión del conjA con el conjunto conjB.
- conjA |= conjB calcula la unión del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA & conjB calcula la intersección de los conjA y conjB.
- conjA &= conjB calcula la intersección del conjA con el conjunto conjB y el resultado se almacena en conjA.
- conjA <= conjB indica si el conjA es un subconjunto del conjunto conjB.
- conjA < conjB indica si el conjA es un subconjunto propio del conjunto conjB.
- conjA >= conjB indica si el conjA es un superconjunto del conjunto conjB.
- conjA > conjB indica si el conjA es un superconjunto propio del conjunto conjB.

Los métodos equivalentes a estos operadores son:

- .difference(): retorna la diferencia de dos conjuntos. conjA.difference(conjB) Alternativamente se puede usar el operador -.
- .difference_update(): retorna la diferencia de dos conjuntos, almacenando en el resultado en el objeto que invoca al método. En concreto, conjA.difference_update(conjB) actualizaría el contenido de conjA. Alternativamente se puede usar el operador -=.
- .union(): la unión de conjuntos. conjA.union(conjB). Alternativamente se puede usar el operador |.
- .update(): modifica el conjunto con la unión de conjuntos y el resultado lo almacena en el primer conjunto. conjA.update(conjB). Alternativamente se puede usar el operador |=.
- .intersection(): la intersección de conjuntos. conjA.intersection(conjB). Alternativamente se puede usar el operador &.
- .intersection_update(): modifica el conjunto con la intersección de conjuntos. Alternativamente se puede usar el operador &=.
- .issubset(): indica si un conjunto es subconjunto de otro. conjA.issubset(conjB). Alternativamente se puede usar el operador <=.
- .issuperset(): indica si un conjunto es un superconjunto de otro. conjA.issuperset(conjB). Alternativamente se puede usar el operador >=.

Otros métodos que se pueden realizar con los conjuntos son:

- .add(x): **añadir** un elemento x.
- .discard(x): **elimina** el elemento x.
- .remove(x): igual que .discard() **pero** si el elemento no existe aparecerá un mensaje de error.
- .clear(): **borrar** todos los elementos.
- .copy(): realizar una **copia** del conjunto.
- .isdisjoint(conj): indica si conj **es disjunto** con el conjunto actual.
- .pop(): **borra y retorna** un elemento arbitrario del conjunto.

Diccionarios Construcción: {}, {k1: v1, k2: v2, ...}, dict(), o dict(iterable). *Por comprehensión en Anexo III*.

Los mapeos representan a conjuntos finitos de objetos pero indexados. La notación de subíndice m[k] selecciona el elemento indexado por k en el mapeo m. Los índices no tienen que ser necesariamente números naturales. Para ello se utilizan tablas hash y políticas de resolución de colisiones.

De forma nativa Python implementa los **diccionarios**, que son su forma de llamar a los mapeos. Representan a una colección mutable y no ordenada de pares clave-valor. La clave puede ser cualquier valor inmutable (ese es el valor que

indexa en el mapeo). La razón es que la implementación eficiente de los diccionario requiere una clave que permanezca constante.

Se dispone de la función len() que devuelve el número de elementos en un mapeo. Se pueden construir diccionarios vacíos con {} y dict(). Mediante asignación se define un diccionario usando llaves e indicando los pares (clave, valor):

```
dic = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Ejemplo:

```
dic = {
    'Madrid': 'España',
    'París': 'Francia',
    'Londres': 'Inglaterra'
}
```

Mediante el constructor se pasa como argumento una lista con tuplas (clave, valor):

```
dic = dict([
    (<key>, <value>),
    (<key>, <value>),
      .
      .
    (<key>, <value>)
])
```

Ejemplo:

```
dic = dict([
    ('Madrid', 'España'),
    ('París', 'Francia'),
    ('Londres', 'Inglaterra')
])
```

En el caso de que las claves sean string se puede simplificar esta segunda opción quitando la notación de tuplas y realizando la asignación `clave = valor`.

```
dic = dict(
    <key>=<value>,
    <key>=<value>,
      .
      .
    <key>=<value>
)
```

Ejemplo:

```
dic = dict(
    Madrid='España',
    París='Francia',
    Londres='Inglaterra'
)
```

Operadores con diccionarios son:

- d[key] = value: **Asigna** el valor value a d[key].
- len(d): retorna el número de entradas almacenadas; es decir, el número de pares (clave, valor).
- del d[k]: borra la clave k junto con su valor.
- key [`not`] `in` d: indica si la clave [no] está en d.

Métodos con diccionarios son:

- .setdefault(key,[v]): **Retorna** el valor de key, pero si key no existe le asigna el valor v. **Muy útil** para inicializaciones.
- copy(): realiza una copia del diccionario.
- .keys() y .values() retornan las **claves** y **valores**, respectivamente. Para acceder a ellos en forma de lista, se debe hacer casting con list().
- .get(k): retorna **el valor de la clave** dada.
- .items(): retorna una **lista** cuyos elementos son tuplas **(clave, valor)**.
- .popitem(): **Borra el último** par clave-valor añadido al diccionario y lo retorna como tupla.
- .pop(k): **Retorna** el valor de la clave dada **y elimina** del diccionario el par (key,pop(key)).
- .update(otroDict): **fusiona** las claves y valores de los diccionarios. Sobreescribe los valores que tengan la misma clave.
- .clear(): **Limpia** el diccionario de pares clave-valor.

```python
# Definir un diccionario con claves de tipos mixtos
d: dict = {1: 'value', 'key': 2}
print(d) # Salida: {1: 'value', 'key': 2}

# Mostrar las claves del diccionario
print(d.keys()) # Salida: dict_keys([1, 'key'])

# Mostrar los pares clave-valor del diccionario
print(d.items()) # Salida: dict_items([(1, 'value'), ('key', 2)])

# Acceder a los valores usando las claves
print("d[1] =", d[1]) # Salida: d[1] = value
print("d['key'] =", d['key']) # Salida: d['key'] = 2

# Crear diccionarios usando dict() a partir de listas de listas/tuplas
d1: dict = dict([[1, 2], [3, 4]])
print(d1) # Salida: {1: 2, 3: 4}

d2: dict = dict([(3, 26), (4, 44)])
print(d2) # Salida: {3: 26, 4: 44}

# Actualizar d1 con los pares clave-valor de d2
d1.update(d2)
print(d1) # Salida: {1: 2, 3: 26, 4: 44}
```

Puede interesar defaultdict de collections. Ej: d = defaultdict(lambda: "No existe"). Si d[clave] no existe invocará a la función dada pero no lanzará un error.

1.2.2. Programación estructurada secuencial

Consta de una secuencia de órdenes directas. El conjunto de instrucciones **vienen dadas por el lenguaje** de programación.

- Sentencias de Asignación, consistentes en el paso de valores de una expresión o literal a una zona de la memoria.
- Lectura, `input()`, consistente en recibir desde un dispositivo de entrada algún dato (usualmente, el teclado).
- Escritura, `print()`, consiste en mandar a un dispositivo de salida algún valor (normalmente la pantalla). La función print() permite escribir el texto de diferentes formas, indicamos solo algunas:
  - **Print básico con varios argumentos**. Puede recibir múltiples argumentos y los separa automáticamente por un espacio.

    ```python
    print("Hola", "mundo", 123) # Salida: Hola mundo 123
    ```

  - **Uso de concatenación con el operador +**. Se pueden concatenar strings usando el operador +, pero debes asegurarte de que todos los elementos sean cadenas de texto.

    ```python
    nombre = "Sergio"
    print("Hola, " + nombre) # Salida: Hola, Sergio
    ```

  - **Uso de comas para separar elementos**. Al usar comas, la función automáticamente convierte los elementos en strings y los separa con un espacio.

    ```python
    edad = 25
    print("Tengo", edad, "años") # Salida: Tengo 25 años
    ```

  - **Uso de f-strings (cadenas formateadas)**. Introducido en Python 3.6, f-strings permiten insertar variables directamente en las cadenas utilizando llaves . Es la forma más moderna y recomendada para componer cadenas con texto y variables intercaladas.

    ```python
    nombre = "Alicia"
    edad = 25
    print(f"Hola, soy {nombre} y tengo {edad} años") # Salida: Hola, soy Alicia y tengo 25 años
    ```

- Generación de números aleatorios. Muy útil en programación. Debemos importar la librería random para que funcione. Destacamos algunas de las formas más comunes:
  - **Generar un número entero aleatorio dentro de un rango**. Usa random.randint(a, b) para generar un número entero aleatorio entre a y b, ambos incluidos.

    ```python
    import random
    numero_aleatorio: int = random.randint(1, 10)
    print(numero_aleatorio) # Salida: Un número entre 1 y 10, por ejemplo 7
    ```

- **Generar un número real aleatorio entre 0 y 1**. Usa random.random() para obtener un número real entre 0.0 y 1.0.

```
import random
numero_aleatorio: float = random.random()
print(numero_aleatorio) # Salida: Un número real entre 0.0 y 1.0, por ejemplo 0.5678
```

- **Generar un número real en un rango específico**. Usa random.uniform(a, b) para generar un número real entre a y b.

```
import random
numero_aleatorio: float = random.uniform(1.5, 5.5)
print(numero_aleatorio) # Salida: Un número real entre 1.5 y 5.5, por ejemplo 3.725
```

- Tamaño, `len`(), calcula el número de datos que tiene una secuencia (p.e. un string, una lista).
- Identificación, `id`(), retorna la referencia de una variable.
- Tipo, `type`(), indica el tipo de dato de un literal o de una variable.
- ...

En Python tenemos:

- **Sentencias de asignación:** `https://docs.python.org/3/reference/simple_stmts.html`
- **Built-in Functions:** `https://docs.python.org/3/library/functions.html`

1.2.3. Programación estructurada condicional

Es aquella que ejecuta ciertas órdenes si se cumple una condición booleana.
En Python se usa la sentencia compuesta con cláusulas if, elif, else.

- Condicional simple.

```
if condicion:
    estructura
```

```
if condicion: sentencia  # Inline
```

- Condicional doble.

```
if condicion:
    estructura_if
else:
    estructura_else
```

```
var = exp_si_true  if  condicion else exp_si_false # Inline
```

- Condicional anidado.

```
if condicion1 [op condicion2 [op condicion3] ... ]:
    estructura_if
elif condicion:
    estructura_else_if
else:  # Casi obligado si se usa elif.
    estructura_else
```

1.2.4. Programación estructurada iterativa

Bucle while. La versión imperativa de esta estructura consta de los siguientes pasos:

1. Se parte de una variable, que se llamará variable de control y que se inicializará a cierto valor.
2. Entonces se comprueba una condición booleana donde interviene la variable de control.
3. Si la condición es cierta, entonces se ejecutarán nuevas estructuras.
4. Entres las estructuras habrá alguna secuencial que modifique la variable de control.
5. Se vuelve al paso 2.

El proceso se repite hasta que la variable de control tome un valor que hace que la condición booleana sea falsa.
En Python se usa la sentencia while:

```
control = valor_inicial
while expresion_booleana_con_la_var_de_control:
    estructuras
    modificar la variable de control
else: # opcional
   estructuras
```

El bloque else no se realizará si se ejecutara la sentencia break en el bloque while.

Bucle for. La Abstracción de Iteración permite recorrer elementos de un contenedor **sin tener en cuenta** su representación interna. De forma transparente al programador se recurre a un iterador que sabe cómo recorrer al contenedor. La abstracción tiene la forma:

```
Para cada elemento P de Contenedor
    acción sobre P
```

En Python algunos contenedores se llaman iterables.

- Todas las secuencias estudiadas previamente son objetos iterables.
- Los iterables responden a la abstracción de iteración.
- La abstracción se particulariza como:

```
for x in contenedor:
    acción con x
[else : acciones]
```

Si bien para el caso de diccionarios, se tienen estas opciones:

```
for k[,v] in diccionario:
    acción con k [,v]
[else : acciones]
```

```
for x in diccionario.values():
    acción con x
[else : acciones]
```

Instrucciones para controlar la iteración. Existen dos sentencias, break y continue, que pueden ser útiles a la hora de programar. Sin embargo, pueden llevar a malas prácticas en programación, alterando el flujo de ejecución del programa. En la asignatura no se permite su uso, salvo en situaciones muy concretas.

- break
  - Solo puede ocurrir sintácticamente en un bucle for o while.
  - Terminará el bucle adjunto más cercano y omitirá la cláusula opcional else.
- continue
  - Solo puede ocurrir sintácticamente en un bucle for o while.
  - Continúa con la siguiente iteración del bucle más cercano.
  - No ejecutará lo que aparezca después de continue.

Programación procedimental

Una función es una **secuencia** de instrucciones identificada con un **nombre** que retorna un valor. En Python una función puede retornar varios valores y, en este caso, "empaqueta" todos los datos de retorno en una tupla.

```
def nombre_funcion ( lista de parámetros ) -> tipo de dato de retorno:
   estructuras de la función
   return valores
```

Un procedimiento es una función que no tiene la instrucción de retorno.

```
def nombre_procedimiento ( lista de parámetros ) [-> None]:
   estructuras del procedimiento
```

Si una función/método tiene n-parámetros podemos invocar a la función con n-argumentos de tal forma que el 1^{er} argumento se sustituya por el 1^{er} parámetro, el 2^{o} argumento por el 2^{o} parámetro, etc ... Son parámetros posicionales.

```
# Definir una función con 4 parámetros
def fun(a: int, b: int, c: int, d: int) -> None:
    print(a, b, c, d)

# Invocar la función con 4 parámetros posicionales
fun(1, 2, 3, 4)
```

Es importante destacar que podríamos eliminar el tipo de retorno None (nada) y funcionaría perfectamente.

Los k-últimos parámetros de una función pueden ser opcionales.

- Los opcionales determinan un valor literal por defecto.
- **Primero** los obligatorios **y después** los opcionales (o por defecto).

```python
def fun(a: int, b: int, c: int = 3, d: int = 4):
    pass # Pass indica que no hace nada. Si lo dejásemos vacío daría error
```

Python permite invocar por palabras claves (keywords).

- Usar **keyword =** especifica el nombre del parámetro en la invocación.
- El orden de los parámetros pueden cambiarse.
- En la declaración de la función, los keywords siempre se pondrán al final.

```python
# Para la función anterior
fun(b=2, d=4, a=1, c=3) # Invocamos con 4 keywords.
fun(1, 2, d=4, c=3) # Los keywords al final
fun(d=4, 1, 2, c=3) # Incorrecto
```

Cuando no se conoce el número de argumentos que se usarán, se usa el Packing Arguments (empaquetamiento de argumentos) con el operador *.

```python
def fun(*args):   # Empaquetará todos los argumentos
    print(args)

fun(1, 2, 3, 4)   # Invocación normal
```

En el ejemplo, todos los argumentos se agrupan en una tupla.

También existe el Unpacking Arguments. Dada una función/método con varios parámetros podemos empaquetar los argumentos con *.

```python
# Definir una función con 4 parámetros de tipo int
def fun(a: int, b: int, c: int, d: int) -> None:
    print(a, b, c, d)

# Crear una lista con 4 enteros
lista: list[int] = [1, 2, 3, 4]

# Invocar la función utilizando el desempaquetado de la lista
fun(*lista)  # Invocación empaquetada
```

Para acceder a uno de los argumentos empaquetados se usan índices:

```python
def fun(*args):
    print(len(args), args[1]) # Muestra el cardinal y el 2o argumento

fun(1, 2, 3, 4)
```

Estaría mejor acceder a ellos con un nombre (keyword). De hecho, podemos empaquetar y usar keywords usando **diccionarios**, con **.

```python
def fun(**kwargs):
    print(f"{len(kwargs)} elementos", kwargs) # Muestra el diccionario

fun(a=1, b=2, c=3, d=4)     # Invocación con keywords
diccionario = {'p1': 1, 'p2': 2, 'p3':3}  # Los veremos
fun(**diccionario)          # Invocación con empaquetado
```

Si se quieren usar los tres modos de pasar argumentos, el orden debe ser:

1. posicionales
2. empaquetados sin keyword
3. empaquetados con keyword

```python
def fun(a: int, b: int, *args: int, **kwargs: int):
    print(a, b)  # Muestra los posicionales
    print(args)  # Muestra los empaquetados sin keyword
    print(kwargs)# Muestra los empaquetados con keyword

fun(1, 2, 3, 4, 5, p1=6, p2=7, p3=8)
```

1.3 Relación de ejercicios

1. **Tipos de datos primitivos y operadores**. Escribe un programa que pida al usuario dos números enteros. El programa debe realizar las siguientes operaciones: suma, resta, multiplicación, división y módulo entre ambos números. Luego realiza la siguiente operación $3 + 5 * 2^2/(4 - 1)$ y muestra el resultado por pantalla.
   Después de esto, compara ambos números utilizando las siguientes expresiones booleanas y muestra los resultados:
   - ¿El primer número es mayor que el segundo?
   - ¿El primer número es igual al segundo?
   - ¿El primer número es distinto al segundo?

   Asegúrate de que tu código contiene:
   - Al menos un comentario de una línea explicando brevemente lo que hace el programa.
   - Un comentario de varias líneas.

   Ejemplo de entrada del usuario:

   ```
   Ingresa el primer número: 4
   Ingresa el segundo número: 2
   ```

   Salida esperada:

   ```
   Suma: 6
   Resta: 2
   Multiplicación: 8
   División: 2.0
   Módulo: 0
   Resultado de 3 + 5 * 2 ** 2 / (4 - 1): 9.333333333333334
   ¿El primer número es mayor que el segundo? True
   ¿El primer número es igual al segundo? False
   ¿El primer número es distinto al segundo? True
   ```

2. **Casting entre tipos primitivos**. Crea un programa que reciba un valor numérico en formato string desde input(), lo convierta a entero y luego a real, y realice una operación matemática con él (por ejemplo, multiplicarlo por 1.5). Muestra el resultado de cada conversión y operación.
   Ejemplo de entrada del usuario:

   ```
   Ingresa un valor numérico: 2
   ```

   Salida esperada:

   ```
   Valor convertido a entero: 2
   Valor convertido a real: 2.0
   Resultado de multiplicar el valor por 1.5: 3.0
   ```

3. **Trabajando con strings (inmutables)**. Escribe un programa que reciba una frase del usuario y realice las siguientes operaciones:
   - Convertir toda la frase a mayúsculas usando upper().
   - Convertir toda la frase a minúsculas usando lower().
   - Aplicar capitalize() a la frase y mostrar el resultado.
   - Aplicar title() a la frase para capitalizar cada palabra.
   - Usar casefold() para comparar la frase original con una versión en minúsculas de la misma, verificando si son iguales.
   - Solicitar al usuario una letra y mostrar el código ASCII de esa letra utilizando ord(), y luego mostrar el carácter correspondiente a un código ASCII ingresado por el usuario utilizando chr().
   - Contar cuántas veces aparece una letra específica en la frase (la letra debe ser introducida por el usuario) utilizando count().

- Usar find() para buscar la posición de una palabra dentro de la frase.
- Dividir la frase en palabras usando split(), mostrar la palabra más larga, y luego unir las palabras de nuevo usando join() para reconstruir la frase.

Ejemplo de entrada del usuario:

```
Ingresa una frase: El sol brilla sobre el mar
Ingresa una letra para buscar su código ASCII: E
Ingresa un código ASCII para convertir a carácter: 97
Ingresa una letra para contar en la frase: l
Ingresa una palabra para buscar su posición en la frase: mar
```

Salida esperada:

```
Frase en mayúsculas: EL SOL BRILLA SOBRE EL MAR
Frase en minúsculas: el sol brilla sobre el mar
Frase con la primera letra en mayúscula: El sol brilla sobre el mar
Frase con cada palabra en mayúscula: El Sol Brilla Sobre El Mar
Comparación de la frase original con su versión en minúsculas: True
Código ASCII de la letra 'E': 69
Carácter correspondiente al código ASCII 97: a
La letra 'l' aparece 3 veces en la frase.
La palabra 'mar' se encuentra en la posición: 24
Palabra más larga en la frase: brilla
Frase reconstruida: El sol brilla sobre el mar
```

4. **Listas (mutables)**. Define una lista con cinco números de forma manual. Por ejemplo, con los valores [10, 20, 30, 40, 50]. Luego, realiza las siguientes operaciones en el siguiente orden:
   - Consultar el valor del tercer elemento de la lista.
   - Obtener una porción (slice) de la lista que contenga los elementos desde la posición 1 a la 3 (sin incluir la posición 3).
   - Agregar un nuevo número al final de la lista (por ejemplo, el 60).
   - Eliminar el segundo número de la lista.
   - Generar un número aleatorio y añadirlo como primera posición de la lista.
   - Ordenar la lista de mayor a menor.

   Ejemplo de salida:

```
Lista original: [10, 20, 30, 40, 50]
Valor del tercer elemento: 30
Slice de la lista (elementos 1 a 3): [20, 30]
Lista después de agregar un número: [10, 20, 30, 40, 50, 60]
Lista después de eliminar el segundo elemento: [10, 30, 40, 50, 60]
Lista ordenada de mayor a menor: [60, 50, 40, 30, 10]
Lista después de añadir un número aleatorio en la primera posición:
[3, 60, 50, 40, 30, 10]
Lista final después de todas las operaciones: [3, 60, 50, 40, 30, 10]
```

5. **Tuplas (inmutables)**. Define manualmente una tupla que contenga los nombres de cuatro ciudades. Luego, realiza las siguientes operaciones:
   - Accede al segundo valor de la tupla y muéstralo por pantalla.
   - Convierte la tupla en una lista utilizando `list`() para poder ser modificada.
   - Cambia el valor de la tercera ciudad por otro nombre de ciudad de tu elección. Tras ello, utilizando `print`(), `type`() y `len`(), muestra el contenido actualizado de la lista, el tipo de la estructura y su longitud después de la modificación, en una única línea.

- Convierte la lista modificada de nuevo en una tupla utilizando `tuple()`. Nota importante: especifica la nueva variable de tipo tupla únicamente con el tipo explícito `tuple`, pues la lista que convirtamos podría tener datos de diferentes tipos (enteros, string, etc).
- Muestra por pantalla el contenido de la tupla, su tipo y longitud.

Ejemplo de salida:

```
Tupla original: ('Madrid', 'Barcelona', 'Valencia', 'Sevilla')
La segunda ciudad es: Barcelona
Lista convertida desde la tupla:
['Madrid', 'Barcelona', 'Valencia', 'Sevilla']
Lista actualizada:
['Madrid', 'Barcelona', 'Malaga', 'Sevilla'], Tipo: <class 'list'>, Longitud: 4
Tupla después de la reconversión:
('Madrid', 'Barcelona', 'Malaga', 'Sevilla'), Tipo: <class 'tuple'>, Longitud: 4
```

6. **Conjuntos (mutables)**. Escribe un programa que defina dos conjuntos de números enteros de forma manual y los muestre por pantalla. Por ejemplo, el primer conjunto podría ser {1, 2, 3, 4, 5} y el segundo {4, 5, 6, 7, 8}. Luego, realiza las siguientes operaciones, mostrando el contenido de ambos conjuntos tras cada operación:
   - Muestra la unión de ambos conjuntos.
   - Muestra la intersección de ambos conjuntos.
   - Muestra los elementos que están en el primer conjunto pero no en el segundo.
   - Añade un nuevo número al primer conjunto.
   - Elimina un número del segundo conjunto.

   Ejemplo de salida:

```
Conjunto 1: {1, 2, 3, 4, 5}
Conjunto 2: {4, 5, 6, 7, 8}
Unión de los conjuntos: {1, 2, 3, 4, 5, 6, 7, 8}
Intersección de los conjuntos: {4, 5}
Elementos en el conjunto 1 pero no en el conjunto 2: {1, 2, 3}
Conjunto 1 después de añadir el número 9: {1, 2, 3, 4, 5, 9}
Conjunto 2 después de eliminar el número 6: {4, 5, 7, 8}
```

7. **Crea un diccionario donde las claves sean los nombres de cuatro estudiantes y los valores sean sus respectivas notas en un examen. Luego, realiza las siguientes operaciones, mostrando el diccionario actualizado tras cada operación:**
   - Muestra el diccionario original.
   - Muestra el listado de estudiantes y el número total de estudiantes en el diccionario.
   - Muestra el listado de las notas almacenadas, sin mostrar los nombres de los estudiantes.
   - Añade un nuevo estudiante y su nota al diccionario.
   - Modifica la nota de uno de los estudiantes.
   - Elimina a un estudiante del diccionario.
   - Muestra el promedio de las notas de los estudiantes restantes.

   Ejemplo de salida:

```
Diccionario original: {'Ana': 8.5, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8}
Listado de estudiantes: ['Ana', 'Luis', 'María', 'Pedro']
Número total de estudiantes: 4
Listado de notas: [8.5, 7.3, 9.1, 6.8]
Diccionario después de añadir a Carlos:
{'Ana': 8.5, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8, 'Carlos': 8.0}
Diccionario después de modificar la nota de Ana:
{'Ana': 9.0, 'Luis': 7.3, 'María': 9.1, 'Pedro': 6.8, 'Carlos': 8.0}
Diccionario después de eliminar a Pedro:
```

```
{'Ana': 9.0, 'Luis': 7.3, 'María': 9.1, 'Carlos': 8.0}
Promedio de las notas: 8.35
```

8. **Condicionales**. Escribe un programa que determine si un cliente de una tienda es elegible para un descuento especial. El programa debe solicitar al cliente dos datos: 1) La cantidad de dinero que ha gastado en la tienda durante el último mes (en euros), y 2) Si tiene una membresía premium (si/no). Nota: representar internamente la membresía como un booleano. El programa debe aplicar las siguientes reglas para determinar si el cliente recibe un descuento:
   - Si el cliente ha gastado más de 100€ y tiene una membresía premium, se le otorga un descuento del 20 %.
   - Si el cliente ha gastado más de 100€ o tiene una membresía premium, se le otorga un descuento del 10 %.
   - Si no cumple ninguna de las condiciones anteriores, no se le otorga ningún descuento.

   El programa debe mostrar el porcentaje de descuento que recibe el cliente.

   Ejemplo de salida esperada:

   ```
   Introduce el gasto del último mes: 120
   ¿Tienes membresía premium (si/no)? no
   Descuento aplicado: 20%
   ```

9. **Bucle while**. Escribe un programa que solicite al usuario un número entero positivo. El programa debe sumar los números desde 1 hasta ese número, pero se detendrá si la suma acumulada supera un valor límite, que también debe ser proporcionado por el usuario. Utiliza un bucle `while` con una expresión booleana en la condición para controlar el proceso.
   Reglas: el bucle debe continuar mientras el número actual sea menor o igual al número proporcionado y la suma acumulada sea menor o igual al límite establecido.
   Ejemplo de salida esperada:

   ```
   Introduce un número entero positivo: 5
   Introduce un límite para la suma: 10
   Suma acumulada: 1
   Suma acumulada: 3
   Suma acumulada: 6
   Suma acumulada: 10
   La suma se detiene porque alcanzó o superó el límite.
   ```

10. **Bucle for con range()**. Escribe un programa que solicite al usuario un número entero positivo que representará el tamaño de una palabra. El programa debe componer un string de dicho tamaño utilizando la función `range`(). En cada iteración del bucle `for`, se generará un número aleatorio del 0 al 9, se mostrará la posición actual, y se concatenará el número al string resultante. Finalmente, el programa deberá mostrar el string generado.
    Ejemplo de salida:

    ```
    Introduce un número entero positivo: 5
    Posición 0: 4
    Posición 1: 9
    Posición 2: 3
    Posición 3: 7
    Posición 4: 2
    String generado: 49372
    ```

11. **Bucle for con listas**. Escribe un programa que genere una lista de tamaño aleatorio entre 3 y 7. Luego, usando un bucle `for`, llena la lista con números aleatorios entre 0 y 9 (ambos inclusive). Finalmente, mostraremos el contenido de la lista de tres formas diferentes:
    - Hacer `print`() directamente sobre la variable de tipo lista creada.
    - Usar un bucle `for` con `range`() para mostrar el contenido indicando el índice de la posición y su valor. Nota: considerar el uso de `len`().

- Usar un bucle `for` haciendo uso del operador `in` de la siguiente forma: `for` elemento `in` lista .

Ejemplo de salida:

```
Tamaño de la lista: 5
Lista generada: [2, 8, 5, 3, 7]
Elemento en la posición 0: 2
Elemento en la posición 1: 8
Elemento en la posición 2: 5
Elemento en la posición 3: 3
Elemento en la posición 4: 7
```

12. **Funciones con parámetros obligatorios**. Escribe una función que reciba dos números enteros como parámetros y retorne el mayor de ellos. El programa debe generar aleatoriamente cinco pares de números, y para cada par, utilizar la función para encontrar y mostrar el mayor número. ***¿Sabrías documentar la función con Docstring?***
Ejemplo de salida:

```
El mayor número entre 82 y 93 es: 93
El mayor número entre 28 y 62 es: 62
El mayor número entre 78 y 30 es: 78
El mayor número entre 23 y 69 es: 69
El mayor número entre 25 y 32 es: 32
```

13. **Funciones con parámetros opcionales**. Crea una función que reciba el nombre de una persona y su edad, siendo la edad un parámetro opcional (por defecto 18). La función debe imprimir un mensaje del tipo: "Hola [nombre], tienes [edad] años". Si no se proporciona la edad, debe imprimir el valor por defecto.
Ejemplos de salidas posibles:

```
Hola Ana, tienes 18 años.
Hola Carlos, tienes 25 años.
```

14. **Funciones con argumentos empaquetados**. Escribe una función que reciba una cantidad variable de números (*args) y devuelva el promedio de todos ellos. Además, la función debe aceptar argumentos con nombre (**kwargs) para mostrar mensajes personalizados al usuario (por ejemplo, un mensaje de bienvenida y otro de despedida).
Ejemplo de salida:

```
¡Hola! Bienvenido a la calculadora de promedios.
Gracias por usar el programa.
El promedio es: 30.0
```

15. **Funciones con argumentos de todo tipo**. Define una función que reciba como argumento obligatorio el nombre de una persona, un parámetro opcional que represente el saldo (por defecto 0), y una serie de transacciones utilizando *args para sumar al saldo. Por ejemplo, las transacciones a realizar se pueden expresar como [100, -50, 200, -30], donde números positivos indicarían depósitos a cuenta (aumentar el valor de la cuenta), y números negativos retiradas de efectivo (valores negativos). Además, recibe **kwargs con información adicional sobre la persona (como dirección o número de cuenta). La función debe devolver dos valores: el saldo final después de aplicar las transacciones y un string con un mensaje ilustrativo detallando los aspectos adicionales de la persona.
Ejemplo de salida

```
Saldo final: 320
Cliente: Juan Pérez
Direccion: Calle Falsa 123
Numero_cuenta: 123456789
```

16. **Enumerados**. Define un enumerado Color para gestionar colores. El programa tendrá disponibles los siguientes colores:

- ROJO: Representado por el número 1.
- VERDE: Representado por el número 2.
- AZUL: Representado por el número 3.
- AMARILLO: Representado por el número 4.

Tras ello, programa la siguiente funcionalidad:

- Define una variable de tipo Color que represente tu color favorito de entre los disponibles, y muestra su valor por pantalla.
- Crea una función `mostrar_color` que reciba un Color y muestre por pantalla su nombre y su valor.
- Crea una función capaz de mostrar todos los colores disponibles en el enumerado utilizando un bucle for (podemos hacer uso de `mostrar_color`). Nota: podemos usar un bucle for para iterar las opciones del enumerado, con `for color in Color`.
- Crea una función capaz de mostrar todos los colores disponibles en el enumerado utilizando un bucle for (podemos hacer uso de `mostrar_color`). Nota: podemos usar `for color in Color`.

1.4 Anexos

1.4.1. Anexo I: Aspectos avanzados de Python

Este anexo sirve como una ampliación de lo tratado en la sección 1.2 y servirá como documentación de consulta del lenguaje en casos puntuales en los que el estudiante lo requiera.

1.4.1.1. Construcción de Contenedores por comprehensión

Un contenedor por comprensión es el que se construye basado en la notación matemática de creación de conjuntos (builder notation[1]). Por ejemplo, podemos construir: $S = \{ \underbrace{2 \cdot x}_{\text{Expresión salida}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{Conj. entrada}}, \underbrace{x^2 > 3}_{\text{Predicado}} \}$

Definir conjuntos basándonos en propiedades también se conoce como comprensión de conjuntos, abstracción de conjuntos o definición por intención de un conjunto.
En Python la construcción por comprehensión[2] se expresa como

```
salida = <expresion(x) for x in iterable [if condicion]>
```

donde:

- <> representa a `[ ]` si se quiere construir **listas**, representa a `( )` si queremos construir **tuplas** y representa a `{ }` para construir **conjuntos**.
  En el siguiente ejemplo se construye un conjunto porque se usan llaves:

```
{2*x for x in range(5) if x*x > 3}
```

- `for x in secuencia` se puede sustituir por cualquier conjunto de sentencias que definan el valor de `x`. Por ejemplo, un bucle anidado donde aparezca `x`.

```
[2*x for v in range(2) for x in range(5) if x*x > 3]
```

  Notar que el condicional sirve para filtrar los valores de x. Solo a aquellos valores de la secuencia que cumplan la condición se les aplicará la expresión.
- expresion(x) es cualquier expresión sobre x. Algunos ejemplos son:
  - No alterar el valor de x; es decir, expresion(x)=x.
  - Aplicar alguna operación o función. P.e. expresion(x)=x**2.
  - Usar algún método. P.e. expresion(x)=x.upper().
  - Usar expresiones ternarias. P.e. expresion(x)=x `if` x %2==0 `else` 1000.

1.4.1.2. Funciones Lambda o Anónimas.

El λ-cálculo es un sistema formal matemático desarrollado en los años 1930s diseñado para trabajar con la noción de función, aplicación de funciones y recursión. Proporciona una semántica simple para la computación y la primera simplificación es que el λ-cálculo **trata las funciones de forma anónima**. La escritura anónima de $\text{square\_sum}(x, y) = x^2 + y^2$ es $(x, y) \mapsto x^2 + y^2$.
En Python las expresiones anónimas se llaman Funciones Lambda y tiene la siguiente expresión:

```
lambda argumentos: expresión
```

En la definición deberás tener en cuenta:

- Puedes usar cualquier número de argumentos.
- Solo habrá una única expresión.

```
el_doble = lambda x: x * 2
print(el_doble(10))
suma = lambda x, y: x + y
print(suma(4, 5))
acotado = lambda x: 4 <= x <= 8
print(acotado(5))
```

Son útiles junto con funciones clausura como `map()`, `filter()`, reduce(), ... para trabajar con colecciones. Se verán más adelante.

[1] https://en.wikipedia.org/wiki/Set-builder_notation
[2] https://en.wikipedia.org/wiki/List_comprehension