

Programación Científica y HPC

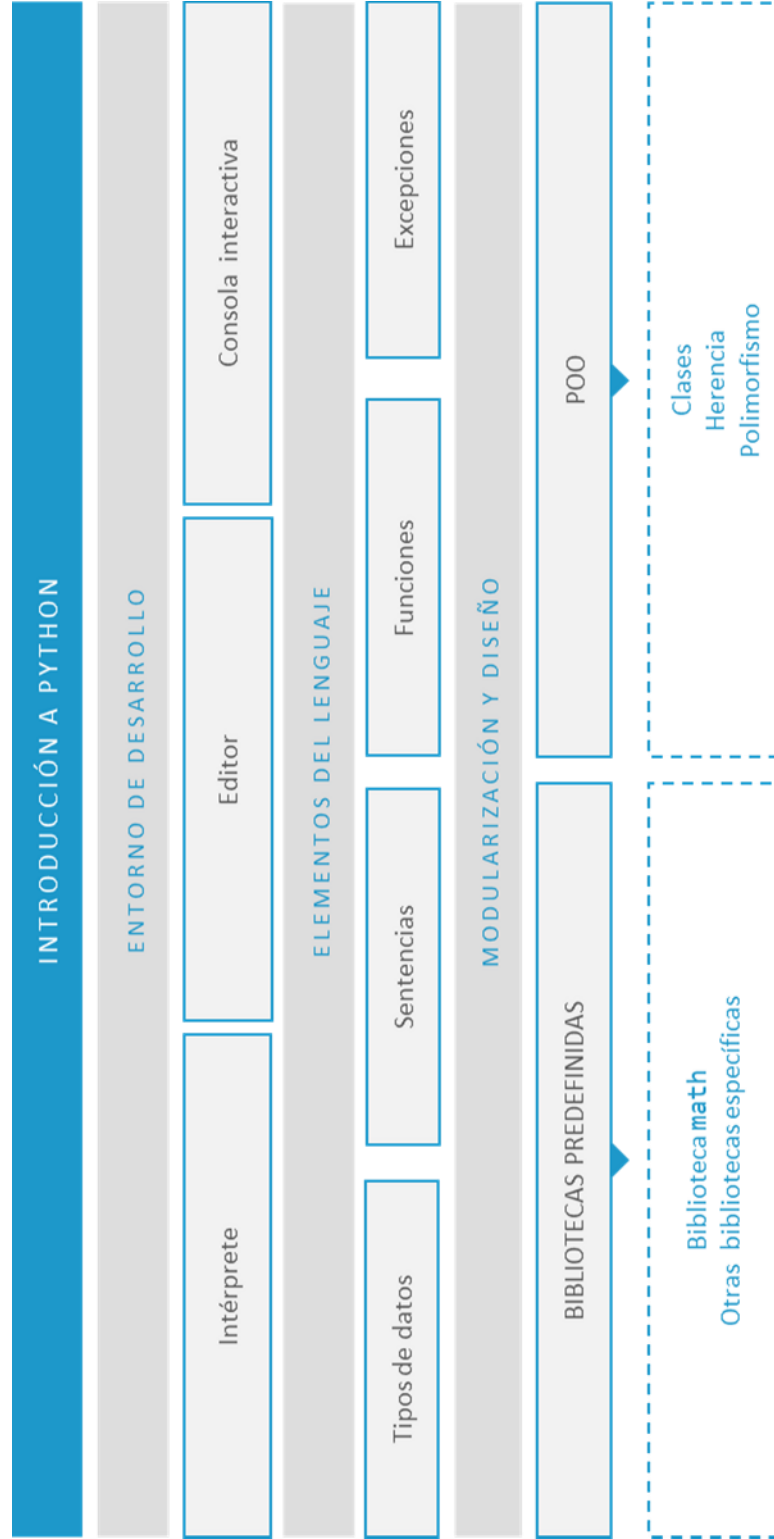
---

# Introducción a Python

# Índice

Esquema	3
Ideas clave	4
2.1. Introducción y objetivos	4
2.2. Entorno de desarrollo	6
2.3. Elementos básicos del lenguaje	11
2.4. Funciones	24
2.5. Manejo de archivos	26
2.6. Excepciones	26
2.7. Módulos	28
2.8. Programación orientada a objetos (POO)	31
2.9. Referencias bibliográficas	39
2.10. Cuaderno de ejercicios	40
A fondo	51
Test	54

# Esquema



## 2.1. Introducción y objetivos

**Python** es un lenguaje de programación que **goza, hoy en día, de una increíble aceptación en la comunidad global del desarrollo de software**. Es difícil encontrar una especialidad en la que Python no sea uno de los lenguajes más usados y demandados laboralmente. Según la encuesta anual de *StackOverflow* de 2019, acerca del estado del desarrollo de software, entre 55 000 participantes consultados, [Python ocupa el cuarto lugar](#) en la categoría de lenguajes de programación más utilizados y el [segundo](#) en la lista de lenguajes de programación más deseados, es decir, son respuestas de programadores que aún no lo usan pero a los que les gustaría aprenderlo. El índice TIOBE, que mide la popularidad de los lenguajes de programación, lo sitúa, en el último trimestre de 2020, en el [tercer lugar](#) por detrás de C y Java, a una corta distancia de este último. Según GitHub, [Python ocupa el segundo lugar](#) en número de contribuyentes únicos, tan solo por detrás de JavaScript (véase Figura 1):

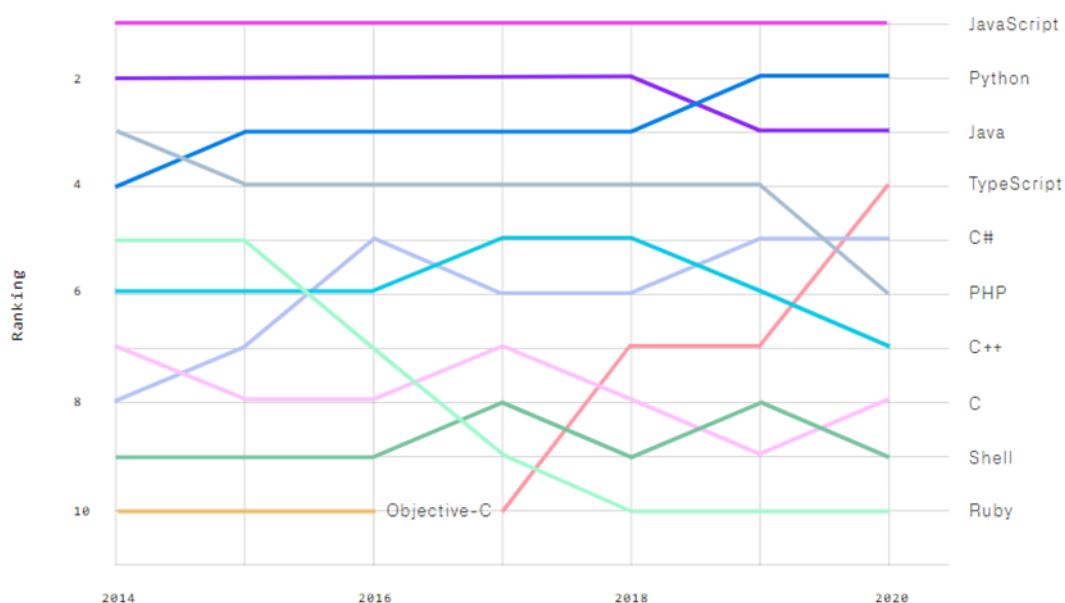


Figura 1. Evolución de popularidad en lenguajes de programación. Fuente: GitHub Octoverse.

Por tanto, se puede concluir que Python **no es un lenguaje en decadencia** (como sí puede estarlo, por ejemplo, FORTRAN) ni que su popularidad se limita a nichos muy concretos (como sí puede ocurrirle, por ejemplo, a R en el campo de la programación estadística). Python **es un lenguaje moderno**, profundamente arraigado en muchas áreas de la computación, dispone de una gran cantidad de módulos y bibliotecas accesibles libremente, y con una comunidad en continua expansión.

Además, Python es un lenguaje de **alto nivel, interpretado, de tipificado dinámico** y que soporta **múltiples paradigmas**, entre ellos programación imperativa, procedural, funcional y orientada a objetos, entre otras.

Los orígenes de Python se remontan a 1991; su creador original, Guido van Rossum, publicó entonces la primera versión del intérprete. Desde sus comienzos, **fue ideado con una filosofía explícita que enfatizaba la simplicidad y la legibilidad del código**. A menudo, otros lenguajes ofrecían múltiples maneras para resolver un determinado problema, así como construcciones sintácticas que reducían la cantidad de código a escribir; sin embargo, esto tiene un coste cognitivo alto para las demás personas que trabajen con ese código, y dificulta el aprendizaje de aquellas que estén comenzando a trabajar con el lenguaje. Python **trata activamente** de evitar estas situaciones y **de resultar lo más familiar posible**, basándose en la premisa de que un código es «escrito una vez, leído mil veces».

Por último, es importante mencionar la existencia de **PEPs** (*Python Enhancement Proposals*). Un PEP **es un documento que detalla las especificaciones de una característica determinada del lenguaje o de su funcionamiento**. Cualquier modificación del lenguaje comienza con una propuesta de PEP detallando los casos de uso y los pormenores de implementación. También existen meta-PEPs, tales como el [PEP-8](#), que detalla la guía de estilo que se debe seguir para programar en Python de manera consistente con el resto de la comunidad, o el [PEP-20](#), que describe el «zen» de Python. El conjunto de todos los PEPs existentes (del orden de 1000) se puede considerar como **el estándar** del lenguaje.

Es por ello, que el objetivo de este tema es **conocer la sintaxis y la estructura de este lenguaje**, partiendo de la base de que ya se conocen los fundamentos básicos de programación, así como entender **qué papel representa Python en el panorama actual de la computación científica y su evolución**.

## 2.2. Entorno de desarrollo

Una de las primeras cuestiones que se debe plantear cuando se empieza a utilizar un determinado lenguaje de programación es conocer las herramientas que se necesitarán. **Para programar con Python se necesita** como mínimo:

### Intérprete

Un intérprete de Python **es lo más esencial, ya que permite el análisis y la ejecución de los programas**. Este es un lenguaje multiplataforma, por lo que existen intérpretes nativos para diversos sistemas operativos (Windows, macOS, Linux, BSD) y arquitecturas (i386, x86\_64, ARM, PowerPC). Es importante saber qué versión del intérprete se instala, ya que **existen diferencias sustanciales entre las distintas versiones** de Python 2.x y 3.x.

La versión 2.x ya no está soportada oficialmente, pero sigue siendo común encontrarla como la versión por defecto, por ejemplo, en Ubuntu 20.04 y macOS 11 Big Sur. Dentro de la actual versión 3.x existen varias versiones menores, como la 3.5, 3.6 y 3.7. En el momento de la confección de este tema, Python 3.9 es la versión más actual. **Las diferencias entre versiones menores son, como cabría esperar, sutiles y se espera que sean, en su mayoría, compatibles entre sí**. Por ejemplo, un código escrito originalmente con Python 3.4 podrá ser interpretado sin cambios por una versión superior como Python 3.8, pero lo contrario no tiene que cumplirse necesariamente. En menor medida, es importante también conocer que existen diversas **implementaciones** de Python: el intérprete oficial está escrito en lenguaje C

(conocido como CPython, véase Figura 2), pero existen implementaciones alternativas, tales como [PyPy](#) (Python en Python), [Jython](#) (Python en Java), [MicroPython](#) (Python destinado a microcontroladores), etc.

```
$ python
Python 3.8.6 (default, Sep 30 2020, 04:00:38)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hola Mundo')
Hola Mundo
>>> █
```

Figura 2. Intérprete de CPython 3.8 mostrando la interfaz REPL (*read-eval-print loop*). Fuente: elaboración propia.

---

Puede ver el código del intérprete oficial de Python en el correspondiente proyecto de [GitHub](#). Su desarrollo ocurre en abierto y cualquier individuo puede aportar parches o reportar problemas.

---

## Editor de textos

Es evidente que se necesitará alguna herramienta que permita escribir código Python con comodidad y eficiencia. Cualquier editor de textos, por simple que sea, es suficiente para meramente escribir código, pero los programadores esperan al menos ciertas mejoras sobre esa experiencia básica, como, por ejemplo, el **resaltado de sintaxis** o el **autocompletado**.

**Existe toda una gama de editores**, desde los muy **sencillos y ligeros**, que ofrecen poco más que las características mencionadas, hasta los **entornos de desarrollo integrados** que aúnan casi todo lo que un desarrollador puede necesitar durante el ciclo de vida del software: depuración, *profiling*, integración con sistemas de control de versiones, consola de comandos integrada, herramientas de refactorización, entre otras.

## Editores ligeros

Estos son algunos ejemplos de editores ligeros que pueden utilizar son [vim](#), [emacs](#), [Sublime Text](#) (véase Figura 3), [Visual Studio Code](#), [Atom](#).

## Entornos de desarrollo integrado

Por otro lado, algunos ejemplo de entornos de desarrollo integrado o integrated development environment (IDE) son [Eclipse](#), [PyCharm](#), [Visual Studio](#), [KDevelop](#), entre otros.

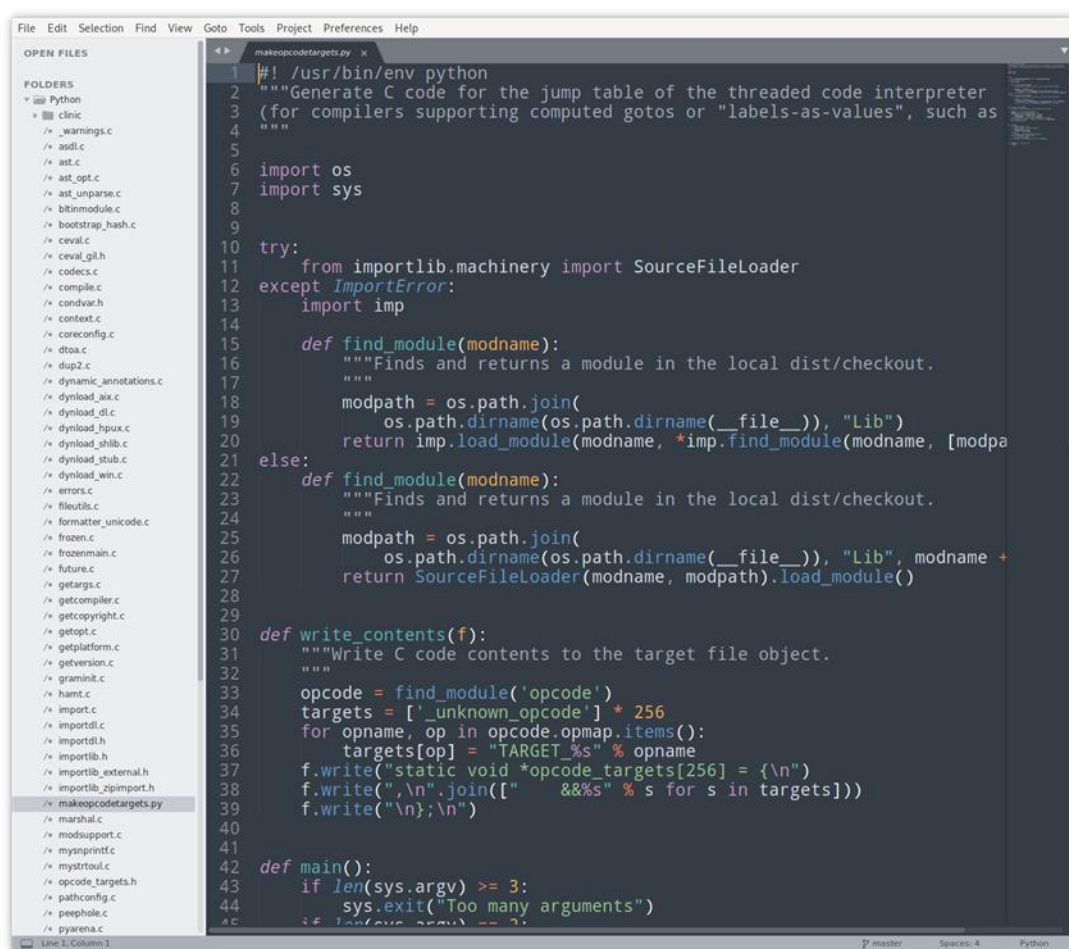


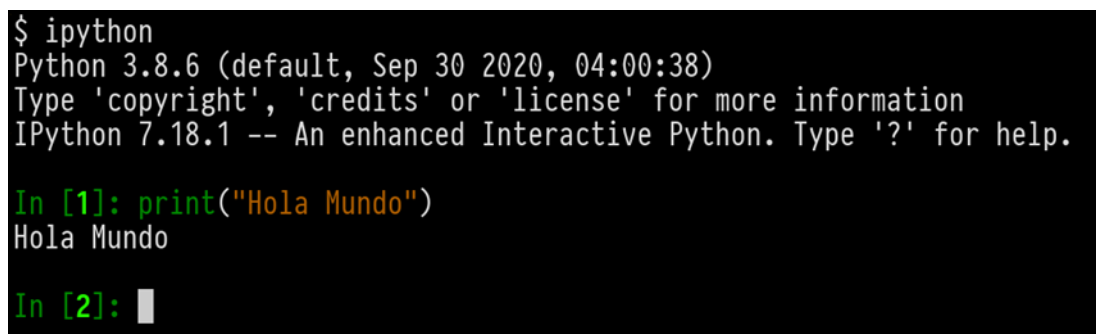
Figura 3. Sublime Text con un archivo de código Python abierto. Fuente: elaboración propia.



Estos son los elementos básicos para trabajar con Python; sin embargo, **se puede hacer uso de una gran variedad de herramientas adicionales** que, aun siendo opcionales, pueden ser de gran ayuda a la hora de programar:

## Consola interactiva mejorada

La experiencia de usuario utilizando el intérprete estándar de Python en modo interactivo (también conocido como REPL, como se ha mencionado previamente en la figura 3) **es bastante básica**, ya que no existe resaltado de sintaxis y las capacidades de autocompletado e introspección son muy limitadas. Existen otros intérpretes totalmente compatibles cuya misión es la de mejorar esta experiencia. El más conocido y utilizado en la comunidad es [IPython](#) (véase Figura 4).



```
$ ipython
Python 3.8.6 (default, Sep 30 2020, 04:00:38)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.18.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print("Hola Mundo")
Hola Mundo

In [2]:
```

Figura 4. Consola interactiva de IPython en acción. Fuente: elaboración propia.

## Herramientas de gestión de dependencias

Casi cualquier proyecto no trivial requiere, en algún momento, importar bibliotecas o librerías de terceros para reutilizar código o incorporar funcionalidades complejas que supondrían otro proyecto en sí mismo. Python **dispone de su propio [ecosistema de paquetes](#)** que pueden ser fácilmente incorporados y gestionados como parte del **proyecto** con el uso de diversas herramientas auxiliares; la más conocida en este ámbito es [pip](#), aunque existen otras utilidades similares como [pipx](#) o [poetry](#).

## Entornos virtuales

El concepto de entorno virtual (usualmente denotado como *virtualenv* en inglés) nace de la necesidad de **gestionar las dependencias de múltiples proyectos en un mismo sistema**. Es posible que distintos proyectos necesiten de versiones diferentes de una misma biblioteca por motivos de compatibilidad interna. Para que estas versiones diferentes puedan coexistir sin causar problemas en el sistema, es posible crear un entorno virtual en un nivel de directorios en el que cada proyecto cuenta con sus bibliotecas en la versión específica que necesitan. Herramientas como [virtualenv](#) o el módulo [venv](#) de la librería estándar de Python, sirven para crear y gestionar estos entornos.

## Formateadores de código

Es importante **mantener una consistencia en cuestiones de estilo a la hora de programar**, como reglas para situar espacios en blanco, el tamaño del sangrado o «indentación», los patrones para nombrar objetos, comentarios y otros. A pesar de no tener ningún efecto en la ejecución propiamente dicha del código, es más sencillo leer, entender y mantener un código que sigue reglas normalizadas y comunes a todos los programadores de Python. Estas reglas están definidas en el [PEP-8](#) y existen utilidades que ajustan automáticamente el formato de los programas para que las sigan, las más notables son [yapf](#), [autopep8](#) y [black](#).

## Linters

La labor de un *linter* es la de **encontrar indicios de usos incorrectos del lenguaje que pueden introducir errores lógicos sutiles y difíciles de depurar**. Algunos ejemplos de estos usos incorrectos pueden ser variables sin utilizar, redefinición de funciones integradas, importación de módulos agresiva utilizando asterisco (\*), código muerto o inalcanzable, entre otros. Hoy en día, son muy utilizados en esta categoría [pylint](#), [pyflakes](#) y [mypy](#).

Antes de comenzar vea el video **Introducción al uso de Python**, en el que se muestra cómo instalar Python, cómo usar la terminal y la ejecución de funciones.



Accede al vídeo

## 2.3. Elementos básicos del lenguaje

Como resumen de todo lo expuesto, se puede decir que Python **es un lenguaje de alto nivel, de propósito general, que soporta distintos paradigmas y que es interpretado**. Los ámbitos en los que se suele usar son diversos, desde la programación científica, programación de sistemas, programación web y aplicaciones de escritorio, hasta para el desarrollo de juegos (Fernández-Montoro, 2013).

Por tanto, en este tema se va a acometer el estudio de un lenguaje extenso con muchas características. Por ello se tratarán los conceptos y ejemplos básicos para familiarizarse con el lenguaje y se proporcionarán enlaces a material adicional para que se pueda profundizar sobre un aspecto en concreto.

Concretamente, en este apartado se van a exponer los elementos básicos del lenguaje, con el objeto de mostrar su sintaxis y los usos más habituales de cada uno. Nótese que el contenido de los cuadros de código siguientes está pensado para ser ejecutado dentro de un intérprete de Python:

- **«Indentación»:** En Python los **niveles de sangrado o «indentación» son relevantes**, sintácticamente hablando. Esto es una decisión de diseño para mejorar la legibilidad, ya que reduce la necesidad de introducir elementos para delimitar bloques como pueden ser las llaves (“{”}) en lenguajes como C++ o Java. Se recomienda sangrar con cuatro espacios y no utilizar tabuladores.

- **Comentarios:** Existen dos tipos de comentarios en Python, sencillos (hasta fin de línea) y multilínea como se muestra en el siguiente código:

```
#Comentario hasta fin de línea
"""Comentario multilínea
    Estan delimitados por caracteres de principio
    y de fin
"""
```

Figura 5. Comentarios en Python. Fuente: elaboración propia.

## Tipos de datos

Para entender el mantenimiento de los tipos de datos en Python, **hay que tener en cuenta que en este todo son objetos, es decir, ejemplares o instancias de una clase.** Una clase es la descripción de un conjunto de objetos similares que cuenta con una serie de operaciones o métodos y unos atributos. Es un concepto propio del paradigma de programación orientada a objetos que, conceptualmente, se usa como base para la implementación de las características de este lenguaje.

Según esto, los tipos predefinidos o estándar en Python son clases para las que se definen las operaciones propias de los tipos. De esta forma, cada valor será un objeto del tipo correspondiente. Las variables serán también objetos. Los tipos son necesarios para la manipulación de los datos de forma consistente y, por ello, las variables que se usen en Python tendrán un tipo asociado.

**No es un lenguaje que se considere fuertemente tipificado**, es decir, los tipos no se asignan a las variables en tiempo de compilación, por lo que no se pueden hacer comprobaciones semánticas sobre ellos durante este proceso.

Ante la duda, siempre será posible consultar el tipo de la variable mediante el uso de la función `type(variable)`. Más adelante se presenta un ejemplo de esto. Pero antes se debe tener en cuenta las siguientes ideas:

Hay dos diferencias importantes en Python con respecto a otros lenguajes considerados tipificados. No se necesita declarar las variables con un tipo. El tipo lo adquieren del valor que se les asigna. Una variable puede cambiar de tipo si se le asigna un valor de tipo distinto al que tenía.

Python cuenta con varios tipos de datos predefinidos. Unos se consideran simples o básicos, porque almacenan un valor atómico, es decir, que no se pueden subdividir. Otros, compuestos, debido a que su valor está formado por varios.

## Los tipos de datos simples en Python

### ► Tipos de datos numéricos

En la Tabla 1 se muestran los tipos con su representación y un código de ejemplo.

Tipo numérico	Tipo Python	Representación	Código ejemplo
Enteros	int	decimal	<pre>&gt;&gt;&gt;a=3 #a es de tipo int &gt;&gt;&gt; type(a) #clase de a &lt;class 'int'&gt;</pre>
Coma flotante	float	binario	<pre>&gt;&gt;&gt; a=0b1010 #a es de tipo int en binario &gt;&gt;&gt; a 10</pre>
		octal	<pre>&gt;&gt;&gt; a=0o67 #a es de tipo int en octal &gt;&gt;&gt; a 55</pre>
		hexadecimal	<pre>&gt;&gt;&gt; a=0xFA #a es de tipo int en hexadecimal &gt;&gt;&gt; a 250</pre>
		punto flotante	<pre>&gt;&gt;&gt;b=1.1 #b es de tipo float &gt;&gt;&gt; type(b) #clase de b &lt;class 'float'&gt;</pre>
		notación científica	<pre>&gt;&gt;&gt; r=2.3e-5 #r es de tipo float &gt;&gt;&gt; r 2.3e-05 &gt;&gt;&gt; type(r) &lt;class 'float'&gt;</pre>
Complejos	complex		<pre>&gt;&gt;&gt;c=1+1j #c es de tipo complex &gt;&gt;&gt; type(c) #clase de c &lt;class 'complex'&gt; &gt;&gt;&gt;c.real 1.0 &gt;&gt;&gt;c.imag 1.0</pre>

Tabla 1. Tipos numéricos en Python. Fuente: elaboración propia.

## ► Operaciones sobre tipos numéricos, su compatibilidad y conversión

Los tipos numéricos en Python tienen definidas las operaciones aritméticas de suma (+), resta (-), producto (\*), división(/) y resto de la división (%). Esta última no está definida para los complejos.

Los tipos numéricos son compatibles entre sí, de forma que cuando se opera con variables de distinto tipo el resultado se convierte al tipo más amplio. El orden es int, float, complex. En la Figura 6 se muestra un ejemplo de uso de operadores y la compatibilidad y conversión de tipos.

```
>>> flotante=1.0                # tipo float
>>> entero=3                    # tipo int
>>> complejo=3-2j               #tipo complex
>>> entero+flotante              #el tipo resultante será float
4.0
>>> entero+complejo              #el tipo resultante será complejo
(6-2j)
>>> flotante/complejo            #el tipo resultante será complejo
(0.23076923076923078+0.15384615384615385j)
>>> flotante%entero              #el tipo resultante será float
1.0
>>> entero%flotante              #el tipo resultante será float
0.0
>>> flotante%complejo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't mod complex numbers.
```

Figura 6. Operadores aritméticos, compatibilidad de tipos y conversión de tipos. Fuente: elaboración propia.

## ► Tipo de datos cadena de caracteres

En Python las cadenas son objetos de la clase str. El valor de una cadena está formado por una serie de caracteres entre comillas simples o dobles.

- **Operaciones soportadas por las cadenas**

Las operaciones que se pueden realizar con las cadenas en Python son, entre otras, operaciones para obtener subcadenas o elementos de la cadena y la operación de concatenación. En el siguiente código se puede ver un ejemplo.

```

>>> s="Hola mundo" # s es de tipo str
>>> type(s)         # tipo de s
<class 'str'>
>>> len(s)           # longitud de la cadena
10
>>> s[0]             # elemento (primer elemento) de la cadena
'H'
>>>
>>> s[1:3]           # dos elementos de la cadena (segundo (1) y tercero(2))
'ol'
>>> s[2:5]           # tres elementos de la cadena
'la '
>>> s[1:]            # todos los caracteres desde el segundo (1) elemento
'ola mundo'
>>> s+" espero que te encuentres bien" # concatenación de cadenas
'Hola mundo espero que te encuentres bien'

```

Figura 7. Cadenas y sus operaciones. Fuente: elaboración propia.

### ► Tipo de datos lógico

En Python el tipo de datos para almacenar valores lógicos es el tipo `bool`. Los valores que toma son `True` y `False`.

- Operadores lógicos y operadores relacionales

Los operadores que se utilizan con las expresiones lógicas o *booleana* son `not` (negación), `and` (conjunción) y `or` (disyunción). Además, los operadores relacionales como `<`, `<=`, `>`, `>=`, `==` y `!=` devuelven como resultados valores *booleanos*.

```

>>> b=True           # b es de tipo bool
>>> b
True
>>> type(b)          # el tipo de b
<class 'bool'>
>>> not b             # negación
False
>>> c=False
>>> b and c           # conjunción
False
>>> b or c            # disyunción
True
>>> c=3<5            # operador relacional menor que
>>> c
True

```

Figura 8. El tipo `bool` y sus usos. Fuente: elaboración propia.

---

Pueden ampliar información con el [tutorial](#) introductorio que ofrece Python sobre el uso de este lenguaje.

---

## Los tipos de datos compuestos en Python

### ► Tipo de datos lista

Este tipo representa secuencias ordenadas de objetos que pueden ser de distintos tipos.

- **Definición.** Las secuencias se construyen separadas por comas y encapsuladas entre corchetes. Pueden crear listas anidadas. Las variables con valores de esta forma son del tipo `list`.
- **Operaciones.** Para el acceso a los elementos se utiliza el operador `[]`, mediante el que se puede obtener un elemento o un rango de elementos. Para insertar y eliminar valores existen las funciones `insert`, `append` y `pop`.

```
>>> lista1=[2, 0.5, "hola"]      # lista de valores distintos con un nivel
>>> type(lista1)                # tipo de lista1
<class 'list'>
>>> lista2=[1, [1,3.4], [2.5]] # lista de valores distintos con varios niveles
>>> type(lista2)                # tipo de lista2
<class 'list'>
>>> len(lista1)                 # longitud de lista 1
3
>>> lista1[2]                  # elemento de índice 2(tercero)de lista1
'hola'
>>> lista2[2]                  # elemento de índice 2(tercero)de lista2 (lista anidada)
[2.5]
>>> lista2[1]                  # elemento de índice 1(segundo)de lista2 (lista anidada)
[1, 3.4]
>>> lista2[1:]                 # elementos de lista2 a partir del segundo(índice 1)
[[1, 3.4], [2.5]]
>>> lista1+lista2              # concatenación de listas
[2, 0.5, 'hola', 1, [1, 3.4], [2.5]]
>>> lista1.append("perro")     # se añade un elemento al final de la lista
>>> lista1
[2, 0.5, 'hola', 'perro']
>>> lista1.insert(1,"gato")    # se añade un elemento en la posición 1
>>> lista1
[2, 'gato', 0.5, 'hola', 'perro']
>>> lista1.pop()               # se quita el ultimo elemento de la lista
'perro'
>>> lista1
[2, 'gato', 0.5, 'hola']
>>> lista1.pop(2)              # se quita un elemento de la posición 2
0.5
>>> lista1
[2, 'gato', 'hola']
```

Figura 8. Uso de listas en Python. Fuente: elaboración propia.



## ► Tipo de datos tupla

Este tipo representa secuencias ordenadas de objetos que pueden ser de distintos tipos, pero que son inmutables, es decir, su valor no puede cambiar.

Se suelen usar para datos que se quieren proteger de la escritura.

- **Definición.** Las secuencias se construyen separadas por comas y encapsuladas entre paréntesis. Se puede tener anidamiento de tuplas. Las variables con valores de esta forma son del tipo `tuple`.
- **Operaciones.** Para el acceso a los elementos se utiliza el operador `[]`, mediante el que se puede obtener un elemento o un rango de elementos.

```
>>> tupla=(1,"hola",3.5)                # tupla de valores distintos con un nivel
>>> type(tupla)                          # tipo de tupla
<class 'tuple'>
>>> tupla.append(3)                       # intento fallido de añadir valores a la tupla
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
>>> tupla[0]                             # primer valor de la tupla
1
>>> tupla[2]=8.5                         # intento fallido cambio del tercer valor
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupla[1:]                             # elementos de la lista a partir del segundo
('hola', 3.5)
>>> tuplaniveles=(1,(3,"hola"),3.5)      # tupla de valores distintos con varios niveles
>>> tuplaniveles
(1, (3, 'hola'), 3.5)
>>> tupla+tuplaniveles                   # concatenación de tuplas
(1, 'hola', 3.5, 1, (3, 'hola'), 3.5)
>>> tupla=tupla+tuplaniveles             # concatenación de tuplas sobre la primera
>>> tupla
(1, 'hola', 3.5, 1, (3, 'hola'), 3.5)
```

Figura 9. Código para el uso de tuplas en Python. Fuente: elaboración propia.

## ► Tipo de datos conjunto

Este tipo representa colecciones no ordenadas de objetos que no pueden ser repetidos.

- **Definición.** Los conjuntos se construyen mediante valores separados por comas y encapsulados entre llaves. Las variables con valores de esta forma son del tipo `set`.
- **Operaciones.** Están definidas las operaciones habituales de conjuntos. En la Figura 10 se muestra un ejemplo.

```
>>> a={1,3,1,2,1}           # conjunto, los elementos repetidos se desechan
>>> a                        # conjunto a resultante
{1, 2, 3}                    # tipo de la variable a
>>> type(a)
<class 'set'>
>>> b={1,2,5,7}
>>> a.union(b)               # unión de conjuntos (operación conmutativa)
{1, 2, 3, 5, 7}
>>> a.intersection(b)       # intersección de conjuntos (operación conmutativa)
{1, 2}
>>> a.difference(b)          # diferencia de conjuntos a-b (operación no conmutativa)
{3}
>>> b.difference(a)          # diferencia de conjuntos b-a (operación no conmutativa)
{5, 7}
```

Figura 10. Conjunto en Python. Fuente: elaboración propia.

#### ► Tipo de datos diccionario

Este tipo representa colecciones no ordenadas de pares clave-valor. Esto permite el acceso indexado a los valores mediante la clave correspondiente.

- **Definición.** El diccionario se construye mediante una lista separada por comas de elementos (clave:valor) encapsulados entre llaves. Las variables con valores de esta forma son del tipo `dict`.
- **Operaciones.** Están definidas operaciones para inserción y extracción por clave, entre otras.

```

>>> diccionario={}                                # diccionario vacío
>>> diccionario
{}
>>> type(diccionario)                             # tipo de la variable diccionario
<class 'dict'>
>>> diccionario["uno"]=3                          #inserción de elemento con clave "uno" y valor 3
>>> diccionario["dos"]=5                          #inserción de elemento con clave "dos" y valor 5
>>> diccionario
{'uno': 3, 'dos': 5}
>>> diccionario[1]=7.5                            #diccionario después de la inserción
>>> diccionario
{'uno': 3, 'dos': 5, 1: 7.5}                       #inserción de elemento con clave numérica
>>> dicc2={"uno":"hola","dos":"adios"}#nuevo diccionario creado con elementos
>>> dicc2
{'uno': 'hola', 'dos': 'adios'} #consulta de elemento con clave "uno"
>>> diccionario['uno']
3
>>> diccionario[1]                                #consulta de elemento con clave 1
7.5>>> diccionario.pop("uno")                      #eliminación del elemento con clave "uno"
3
>>> diccionario
{'dos': 5, 1: 7.5}

```

Figura 11. Diccionarios en Python. Fuente: elaboración propia.

### ► Tipo de datos archivo

Este tipo se define para manejar archivos almacenados en el ordenador o extraídos de Internet.

## Sentencias de control

Ahora se revisarán las sentencias de control disponibles en Python y algunos ejemplos de uso. En estos ejemplos se usarán expresiones y sentencias de entrada/salida con el objeto de mostrar su uso.

### Sentencias condicionales

La sentencia condicional de Python se construye usando sentencias `if` con alternativas `else` o `no`. En las siguientes tablas se muestran las opciones y su funcionamiento.

Si se cumple la condición, se ejecuta el bloque de sentencias que sigue a la condición.

**Sentencia if  
sin else**

```
if numElem < maximoNumElem:  
    ...print("No se ha alcanzado el maximo. El numero de  
    elementos es: ", numElem)  
    ...  
Resultado: No se ha alcanzado el máximo. El numero de  
elementos es: 0
```

Tabla 2. Sentencias condicionales en Python. Fuente: elaboración propia.

Si se cumple la condición, se ejecuta el bloque de sentencias que sigue a la condición, sino se ejecuta el bloque que sigue a else.

**Sentencia if  
con else**

```
>>> numElem=0  
>>> maximoNumElem=10  
>>> if numElem < maximoNumElem:  
...     numElem=numElem+1  
...     print("El numero de elementos es: ",numElem)  
... else:  
...     print("Se ha alcanzado el máximo de elementos.")  
...  
Resultado: Si el máximo es 1. El numero de elementos  
es: 1
```

Tabla 2. Sentencias condicionales en Python. Fuente: elaboración propia.

### Sentencia if, elif y else

El bloque `if` puede ir seguido de una serie de bloques `elif` con condiciones y un bloque `else` (todos opcionales).

Las condiciones se evalúan por orden de escritura y se ejecuta el bloque de la primera que sea verdad (las que van detrás no se evaluarán).

Si no se cumple ninguna se ejecuta el bloque del `else`.

```
>>> numElem=5
>>> maximoNumElem=10
>>> if numElem == 0:
...     print("Condición de comienzo. El numero de
...     elementos es: ",numElem)
... elif numElem < maximoNumElem :
...     print("Se podrían añadir elementos porque no se
...     ha alcanzado el máximo. El numero de elementos es:
...     ",numElem)
... else :
...     print("Solo se pueden eliminar elementos
...     porque se ha alcanzado el máximo. El numero de
...     elementos es: ",numElem)
...
Resultado: Se podrían añadir elementos porque no se ha
alcanzado el máximo. El numero de elementos es: 5
```

Tabla 2. Sentencias condicionales en Python. Fuente: elaboración propia.

Hay que señalar que, en Python, no existe la sentencia de alternativa múltiple, *switch-case* de otros lenguajes. Algunas formas son más eficientes que otras.

Si quiere profundizar sobre las operaciones de Python puede consultar los ejemplos de operadores propuestos por [Pythones](#).

## Bucles

Los bucles de Python son de dos tipos: bucles condicionales y bucles iterativos. En la Tabla 5, se enumeran sus características y se ponen ejemplos de implementación.

Ejecuta repetidamente un bloque de instrucciones mientras se cumple la condición.

#### Bucle while

```
#programa que imprime los enteros de uno a 10

entero=1

while entero<10:
    print(entero," ",end="")    #end="" evita que print
    salte de línea
    entero=entero+1
    print(entero)

Resultado: 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10
```

Tabla 3. Bucles en Python y sus usos. Fuente: elaboración propia.

	<p>Ejecuta de forma iterativa un bloque de sentencias tantas veces como determine el recorrido de una secuencia.</p>
	<pre>for i in range(1,4): #itera sobre un rango     print(i) Resultado: 1           2           3</pre>
	<pre>for i in range(3): #itera sobre un rango     print(i) Resultado: 0           1           2</pre>
	<pre>lista=[1,3,7] for i in lista: #itera sobre una lista     print(i) Resultado: 1           3           7</pre>
	<pre>tupla=(1,6,3) #itera sobre una tupla for i in tupla:     print(i) Resultado: 1           6           3</pre>
Sentencia for	<pre>diccionario={1:"rojo",2:"azul",3:"verde"} for i in diccionario: #itera sobre claves del     print(i) Resultado: 1           2           3</pre>
	<pre>for i in diccionario.keys(): #itera sobre claves del     print(i) Resultado: 1           2           3</pre>
	<pre>for i in diccionario.values():#itera sobre valores     print(i) Resultado: rojo           azul           verde</pre>
	<pre>for i in diccionario.items():#itera sobre elementos     print(i) Resultado: (1, 'rojo')           (2, 'azul')           (3, 'verde')</pre>
	<pre>for c,v in diccionario.items():#itera sobre elementos,     selecciona     print(c,"-&gt;",v) Resultado: 1 -&gt; rojo           2 -&gt; azul           3 -&gt; verde</pre>

Tabla 3. Bucles en Python y sus usos. Fuente: elaboración propia.

Los bucles pueden ser interrumpidos mediante la sentencia `break()` o se puede forzar a que termine una iteración antes de tiempo y salte a la siguiente con `continue()`.

## 2.4. Funciones

Una función es un **bloque de código genérico al que se pone nombre**. Las funciones no presentan siempre la misma ejecución, dado que en ocasiones dependen de unos parámetros cuyos valores cambian el resultado de la ejecución.

Para una función se deben definir, por tanto, los argumentos o parámetros de esta, de los que se debe especificar su nombre y su tipo y el valor de retorno de la función, si es el caso.

Las funciones en Python cuentan con:

- ▶ **Cabecera de la función**

```
def nombre(lista_argumentos):
```

- ▶ **Cuerpo de la función**

```
    Bloque de sentencias indentado  
    Return
```

Un ejemplo de código se muestra a continuación, donde se implementa una función que convierte kilómetros en millas:

```
>>> def convertirKmaMillas(kilometros):  
...     factorConversion=0.621371  
...     return kilometros*factorConversion  
...  
>>> convertirKmaMillas(20)  
12.42742
```

Figura 12. Ejemplo de función en Python. Fuente: elaboración propia.

En la Tabla 4 se muestran los tipos de argumentos de una función y la forma de pasar valores u objetos a estos.



Es muy importante resaltar que los valores o parámetros actuales se asocian, por referencia, a los argumentos de la función en la llamada. Esto quiere decir que, en la función, lo que se realice sobre el argumento tienen efecto sobre los objetos pasados en la llamada, a no ser que sean inmutables.

Argumentos o parámetros formales	Parámetros actuales	
Número determinado sin valores por defecto (param1, param2)	<b>Por posición</b> Los valores u objetos se ponen en el mismo orden que los argumentos.	nombreFuncion(valor1, valor2)
	<b>Por nombre</b> No importa el orden, se indica el nombre al que se va a asociar el valor.	nombreFuncion (param2=valor2,param1=valor1)
Número determinado con valores por defecto que deben irse asignando del último al primer parámetro (param1,param2=valor)	nombreFuncion(valor1,valor2) nombreFuncion(valor1) # se omite el parámetro que tiene valor	
Número indeterminado de parámetros (*nombre_param)	nombreFuncion(valor1,valor2,valor3) nombreFuncion(valor1,valor2,valor3, valor4) Se llama con el número de valores que se quiera. La función debe estar implementada de forma no determinada.	

Tabla 4. Tipos de argumento de una función. Fuente: elaboración propia.

Los argumentos formales son los que se especifican en la definición de la función. Los parámetros actuales son los valores que se pasan cuando se llama a la función

## Retorno de la función

Las funciones puede que devuelvan un resultado de su ejecución o no.

- ▶ Si no devuelven nada terminan con return.
- ▶ Si devuelven algo, return debe ir seguido del objeto o valor que se quiera retornar.

## 2.5. Manejo de archivos

Para usar un archivo en un programa de Python se cuenta, entre otras, con las **funciones elementales** que permiten crear un archivo, abrir archivo, escribir en archivo, leer de archivo y borrar archivo. En la Figura 13 se muestra un ejemplo del uso de los archivos y las funciones que están implementadas en Python.

```
>>> f=open("/Documents/prueba.txt",'w')      #crea un archivo nuevo y lo abre
>>> f.write("Esta es la primera linea")      #escribe en el archivo
>>> f.close()                                #cierra el archivo(importante para cambiar de modo)

>>> f=open("/Users/mluisadiez/Documents/prueba.txt",'a')  #abre el archivo existente para escribir
>>> f.write("\nEsta es la segunda linea")      #escribe en el archivo línea nueva
>>> f.close()                                #cierra el archivo

>>> f=open("/Documents/prueba.txt",'r')      #abre el archivo existente para leer
>>> print(f.read())                          #imprime el contenido del archivo
Esta es la primera linea
Esta es la segunda linea
>>> f.close()                                #cierra el archivo para poder volver a leer

>>> f=open("/Documents/prueba.txt",'r')      #abre el archivo existente para leer
>>> lineas=f.readlines()                     #guarda en una lista las lineas
>>> print(lineas)                           #imprime la lista
['Esta es la primera linea\n', 'Esta es la segunda linea']
>>> for i in lineas:                          #recorre la lista
...     print(i)                            #imprime línea por línea
...
Esta es la primera linea
Esta es la segunda linea
```

Figura 13. Uso de archivos en Python. Fuente: elaboración propia.

---

Para profundizar en la manipulación de archivos y directorios puedes consultar el [tutorial](#) de entrada y salida de Python y el [blog](#) de UniPython sobre operaciones con archivos y carpetas.

---

## 2.6. Excepciones

Para la gestión de errores en tiempo de ejecución en Python se usan objetos de tipo *exception*.

Si las excepciones no se controlan en tiempo de ejecución se aborta la ejecución y se imprime una traza (*tracebak*) informando del error.

En este lenguaje existe una serie de excepciones predefinidas que pueden ser capturas.

El control de excepciones en Python, pensado para evitar que el programa termine de manera abrupta por un error de ejecución, se implementa mediante una estructura try-except-else-finally.

- ▶ **try** va seguido del bloque de código que se quiere ejecutar, pero en el que se podría elevar una excepción.
- ▶ **except** puede ir seguido del nombre de una excepción que será la que capturaré. Si no va seguido de ningún nombre, captura cualquier excepción.
- ▶ **else** va seguido de un bloque de código que se ejecutará si no se produce la excepción.
- ▶ **finally** se ejecuta siempre, se produzca o no la excepción.

```
while True:    #bucle infinito que acabará cuando se consiga dividir

    try:

        x=float(input("Introduzca el numero que quiere dividir: "))
        y=float(input("Introduzca el divisor: "))
        resultado = x / y    #si se divide por 0 se eleva la excepción ZeroDivisionError

    except ZeroDivisionError:
        print("Se ha intentado dividir por cero. Introduzca los datos nuevamente")
    else:
        # se ejecuta si no se produce la excepción
        print("El resultado es", resultado)
        break    #sale del bucle porque se ha hecho la división correctamente
    finally:
        print("Siempre se ejecuta")

Resultados posibles:
Si y=0, se indica el error y se vuelve a pedir datos
Si y no es 0, se ejecuta el else que muestra el resultado de la división y sale del bucle
En cualquier caso se ejecuta finally.
```

Figura 14. Uso de excepciones en Python. Fuente: elaboración propia-

Las excepciones predefinidas son, entre otras:

- ▶ **TypeError**: cuando se usa un tipo de dato que no es compatible con una operación.
- ▶ **ZeroDivisionError**: cuando se divide por cero.
- ▶ **FileNotFoundError**: intento de accesos a un archivo que no existe.

- ▶ **NameError**: se usa un identificador que no ha sido inicializado.

Se pueden definir excepciones propias mediante la declaración de clases derivadas de `Exception`.

---

Para profundizar sobre el manejo de excepciones en pueden consultar los [documentos](#) de la librería en la página oficial de Python.

---

## 2.7. Módulos

Los programas o aplicaciones cuanto más complejos son, más necesitan de su **organización**. Para organizar esta complejidad se debe realizar la **subdivisión de una aplicación en piezas más pequeñas con un grado de independencia grande**, de forma que, tras un proceso de ensamblaje, darán lugar al programa completo. Estas piezas son denotadas como módulos.

**Los módulos se diseñan según los principios de acoplamiento y cohesión.**

- ▶ **Acoplamiento**: medida del grado de interdependencia entre módulos. Es deseable que sea lo menor posible.
- ▶ **Cohesión**: según este principio los elementos del módulo deben estar fuertemente relacionados, de manera que todos traten aspectos de la tarea común que el módulo pone a disposición. Un módulo cohesivo ideal sólo realiza una tarea. Por tanto, la cohesión debe ser altas.

Luego, **un lenguaje de alto nivel debe contar con las herramientas que soporten la «modularización»**, es decir, la **subdivisión de una aplicación en módulos**, cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de los restantes módulos.

Según Liskov (1972), la «modularización» consiste en **dividir en un programa en módulos que pueden ser compilados de forma separada, pero que tienen conexiones con otros módulos.**

Los **módulos** en Python son **archivos en los que se hacen definiciones de funciones, clases, constantes y variables.** Esto permite encapsular recursos que estén fuertemente relacionados, aportando un aspecto de la tarea para la que el módulo ha sido diseñado.

De hecho, las conocidas bibliotecas de Python son en sí mismas módulos. Por ejemplo, la biblioteca `math`, es el módulo que proporciona funciones y constantes matemáticas.

Para usar los módulos en Python deben ser importados en el archivo de código en el que se van a usar mediante el uso de `import nombre_modulo`, es decir, en el caso de la biblioteca matemática mencionada `import math`.

Para usar las funciones `degrees` y `radians` y la constante `pi`, es necesario cualificar su nombre con el del módulo, por ejemplo, `math.degrees`, `math.radians` y `math.pi`.

Si se van a usar pocos recursos, se puede realizar una **importación selectiva, en la que se indica las funciones o recursos que se van a importar.** Esto elimina la necesidad de la cualificación. Para llevar a cabo esta importación se usa `from nombre_modulo import nombre_recurso`.

```

from math import radians,degrees,pi
def conversionMedidasAngulo(angulo,magnitud="radianes"):

    if magnitud=="radianes":
        return radians(angulo) #uso de función de math
    else:
        return degrees(angulo)    #uso de función de math

def tablaEquivalenciasAngulos():

    angulosRadianes=[0,pi/6,pi/4,pi/3,pi/2,pi,3*pi/2,2*pi]

    #uso de constante pi de math

```

Figura 15. Uso de la importación selectiva. Fuente: elaboración propia.

Aunque la escritura del código con esta forma de importación parece más rápida y clara, el uso de la cualificación mejora la compresión, ya que se sabe de donde proviene el recurso, y evita conflictos en el caso de usar identificadores con el mismo nombre. Todo módulo cuenta con lo que se conoce con un espacio de nombre, que se denota mediante el nombre del módulo.

En Python se pueden definir módulos propios mediante la creación de archivos que contengan la definición de las funciones y constantes que se quieran incorporar, y que estén relacionadas con la tarea para la que se diseña el módulo. El nombre del módulo debe ser significativo.

Los **módulos creados por el usuario se importan de la misma forma que los predefinidos del lenguaje**. El mismo se debe almacenar en un directorio, teniendo en cuenta los que se indica a continuación.

Tengan en cuenta que el intérprete de Python busca los módulos en el directorio actual, en los directorios definidos en la variable de entorno PYTHONPATH y en la ruta predeterminada.

Como ejemplo de uso de un módulo propio, se van a encapsular en el mismo archivo las funciones que se han definido en el Ejercicio 3 del cuaderno de ejercicios. De esta forma, se va a crear un módulo al que se pondrá por nombre `calculoNumerosAuxiliares.py`. Tengan en cuenta que esto solo un ejemplo de definición de módulos en Python y que se ha creado con solo dos funciones.

Si se quiere usar el módulo, se tendrá que importar en el archivo. La importación quedaría de esta forma, si el módulo definido se encuentra almacenado en el directorio actual:

```
import calculoNumerosAuxiliares  
calculoNumerosAuxiliares.mcm(2,3)
```

Figura 16. Importación módulo propio. Fuente: elaboración propia.

**El nombre y el diseño del módulo debe hacerse de forma cuidada.** No se trata de juntar muchas funciones en un mismo archivo, aunque no tengan una semántica común subyacente.

Un módulo debe contener todos los recursos necesarios para servir a la tarea para la que ha sido concebido, pero no más de los necesarios.

---

Para profundizar sobre este tema pueden recurrir al [tutorial](#) de disponible en la página de Python.

---

## 2.8. Programación orientada a objetos (POO)

“La programación orientada a objetos es un método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa un ejemplar o instancia de una

clase y cuyas clases son miembros de una jerarquía de clases unidas mediante relaciones de herencia” (Booch, 1994).

Por tanto, y de acuerdo con la definición anterior, para crear una aplicación orientada a objetos, se debe:

- ▶ **Definir clases** (tipos de objetos) que cuenten con una serie de atributos y funciones, que en POO se conocen como métodos que representan el comportamiento de la clase.
- ▶ **Establecer las relaciones entre las clases.** Estas relaciones pueden ser de herencia, de agregación, de asociación y de uso.
- ▶ **Crear objetos de las clases definidas e invocar a sus métodos.** De esta forma es como progresará la ejecución.

Es habitual describir un programa orientado a objetos como una serie de eventos asociados con mensajes. Estos eventos son la creación de objetos (mensaje de construcción), intercambio de mensajes entre objetos (se invoca a los métodos de otros objetos) y eliminación de objetos (mensaje de destrucción que, en la mayoría de los lenguajes modernos, son destruidos por el sistema)

## Principales propiedades de la POO

- ▶ **Abstracción.** Extrae las características esenciales del objeto y separa su comportamiento esencial de su implementación.
- ▶ **Encapsulamiento y ocultación de información.** Los datos y las funciones se encapsulan en el objeto y los detalles de implementación quedan inaccesibles, proporcionando la interfaz necesaria para que sea usado evitando manipulaciones no deseables.
- ▶ **Modularidad.** Los programas se dividen en módulos de acuerdo con los principios vistos en el apartado anterior.



- ▶ **Jerarquía.** Las abstracciones u objetos son ordenadas de acuerdo con relaciones de herencia y agregación.
- ▶ **Polimorfismo y ligadura dinámica.** El polimorfismo tiene que ver con la capacidad de que una entidad tome muchas formas. En concreto, en POO, se trata de que una misma invocación al método de un objeto se ejecute de formas distintas. Para ello, la ligadura entre el objeto y el método debe ser dinámica, es decir, se establece en el momento de la ejecución. (Joyanes-Aguilar, 1998)

Para implementar el polimorfismo en POO debe existir una jerarquía de clases, métodos polimórficos redefinidos en cada clase de la jerarquía y ligadura dinámica entre el objeto invocado y el método polimórfico.

## Términos usados en POO

En la Tabla 5, se exponen y explican los términos que representan a las entidades que se usan en un programa orientado a objetos

Término	Descripción
Clase	Tipo de datos que cuenta con una serie de atributos (miembros de dato) y un comportamiento asociado mediante la definición de métodos o miembros función. Los miembros son accesibles mediante la notación punto. <code>class nombre:</code>
Objetos	Ejemplar o instancia de una clase. Los valores de cada atributo son distintos para cada objeto y soporta los métodos definidos. <code>nombreObjeto=nombreClase(parámetrosConstructor)</code>
Atributos de clase	Atributos cuyo valor es compartido por todos los objetos de la clase. Se declara en el bloque tras las cabecera de la clase. <code>nombre=valorInicial</code>
Atributos de objetos	Se crean al crear el objeto. En Python todos los atributos son públicos. Para conseguir la ocultación se puede nombrar al atributo con dos guiones bajos delante ( <code>__nombre</code> ) y solo se podrá acceder a él a través de un método de la clase. Se crea en el constructor de la clase. <code>nombreAtributo=valorInicial</code>
Métodos	Funciones que soporta el objeto. Los tipos son: <ul style="list-style-type: none"> <li>► <b>Constructor</b> (<code>__init__</code>). Crea el objeto con sus atributos. Tiene al menos un argumento <code>self</code> que referencia el objeto que se crea.</li> <li>► <b>Métodos</b>. Todos tienen al menos un argumento <code>self</code> que referencia al objeto que se enlaza.</li> <li>► <b>Método</b> <code>__str__</code>. Este método permite definir la manera en la que se quiere visualizar los atributos del objeto y se solicitará cuando se invoque <code>print</code> para el objeto.</li> </ul>

Tabla 5. Términos de POO. Fuente: elaboración propia.

Cabe aclarar, que para cumplir con la propiedad de ocultación se debe tener en cuenta que atributos deben ser privados. Por otro lado, en POO se suelen implementar métodos *getters* (para consultar atributo) y *setters* (para modificarlos). Aunque todo es público, en Python se recomienda seguir con esta filosofía e implementar al menos los *setters* para indicar al usuario que se quiere preservar la manipulación.

Un ejemplo se muestra en la Figura 17, en la que se implementa una clase cliente. Se han definido los atributos mínimos para simplificar el ejemplo. Se implementa el

constructor, un método para encriptar la clave, un método *getter* para obtener la clave del cliente, al cual es el único atributo oculto, y un método *\_str\_* para generar una vista del cliente y un método que compara si dos objetos de tipo cliente son iguales (comparación en profundidad y no por referencias).

```
class Cliente:
    numClientes=0 #atributo de clase, compartido por todos los objetos
    #se usa para generar los identificadores de los clientes

    def __init__(self,nombre, direccion, email, clave):
        Cliente.numClientes=Cliente.numClientes+1 #atributo de clase
        self.id="CL"+str(Cliente.numClientes) #atributo id
        self.nombre=nombre #atributo nombre
        self.direccion=direccion #atributo direccion
        self.email=email #atributo email
        self.__clave=self.encriptarClave(email,clave)#atributo clave
        #este atributo esta oculto

    def encriptarClave(self,email,clave):
        import hashlib #biblioteca metodos encriptacion
        aux=email+clave
        claveEncriptada=hashlib.sha3_512(aux.encode('utf-8')).hexdigest()
        return claveEncriptada

    def getClave(self): #metodo para acceder al atributo oculto
        return self.__clave

    def __str__(self): #metodo para genera la vista del cliente
        datosCliente="El cliente es: "
        datosCliente+="\nId de cliente: "+self.id
        datosCliente+="\nNombre: "+self.nombre
        datosCliente+="\nDireccion: "+self.direccion
        datosCliente+="\nEmail: "+self.email
        return datosCliente

    def __eq__(self,objeto):
        return(nombre==objeto.nombre and direccion==objeto.direccion and id==objeto.id and email==objeto.email and getClave()==objeto.getClave())
```

Figura 17. Ejemplo de clase con atributo de clase y atributo oculto. Fuente: elaboración propia.

A continuación, se muestra el uso de la clase y la salida obtenida.

```
cliente=Cliente("Maria López García","Carretas, 7. 28015 Madrid","maria@de.es","MD123$")
# se crea el cliente
print(cliente) #se imprimen los datos del cliente, invoca __str__ del objeto

print(cliente.nombre) #imprime un atributo directamente porque es publico
print("La clave es "+cliente.getClave()) #imprime el atributo invocando al método getter

print(cliente.__clave) #da error porque el atributo está oculto
```

## Salida

```
El cliente es:
Id de cliente: CL3
Nombre: Maria López García
Direccion: Carretas, 7. 28015 Madrid
Email: maria@de.es
Maria López García
La clave es 2891660d8d33622bfda18c2b3cad6865decd1c0d6c16d760f3c24c09d5e6ffdb744b586582124d2ab0c68f7afb3999baffe9758e2cbfdc316121f83cbea6426

AttributeError                                Traceback (most recent call last)
<ipython-input-191-080c30125072> in <module>
      6 print("La clave es "+cliente.getClave()) #imprime el atributo invocando al método getter
      7
----> 8 print(cliente.__clave) #da error porque el atributo está oculto

AttributeError: 'Cliente' object has no attribute '__clave'
```

Figura 18. Construcción de objeto y uso de los métodos. Fuente: elaboración propia.

Como se puede ver, el resultado del intento de acceso al atributo *\_clave* da error porque es privado.

## Herencia

En Python, se tiene la posibilidad de **declarar jerarquías de clases**. Las clases de **los niveles más bajos de la jerarquía heredan los atributos y métodos de su clase padre, denotada también como clase base**. También se ha implementado la herencia múltiple, por lo que una clase puede heredar de varias.

El proceso de definición de clases derivadas se conoce como especialización, dado que se va definiendo un tipo cada vez más concreto.

Si se mira la jerarquía de abajo a arriba se dice que existe una generalización, las clases superiores son menos concretas y específicas que las derivadas.

- ▶ Para **definir una clase derivada** se debe usar la siguiente sintaxis: `class nombre(listaNombresClaseBase):`.
- ▶ Es importante tener en cuenta que, al construir un objeto de la clase derivada, **parte de los atributos pertenecen a la clase base**. Luego, en el constructor de la derivada, se debe invocar al constructor de la clase base mediante la siguiente sentencia: `clasebase.__init__(self,argumentos)` o bien, `super().__init__(argumentos)`.
- ▶ Para **acceder a los atributos de la clase base** se pueden usar los métodos de la clase base. Los métodos se invocarán mediante: `super().metodo_clase_base(argumentos)`.

En la Figura 19 se presenta la clase cliente anterior, pero como clase derivada de persona, de forma que los atributos comunes de todas las personas se definen en la base, y en el cliente solo aquellos específicos de la aplicación que maneja clientes. Se redefine el método `__str__` para la clase derivada

```

class Persona:
    def __init__(self,nombre,direccion):
        self.nombre=nombre
        self.direccion=direccion

    def getNombre(self):
        return self.nombre

    def getDireccion(self):
        return self.direccion

    def __str__(self):
        datosCliente="Los datos de la persona son: "
        datosCliente+="\nNombre: "+self.nombre
        datosCliente+="\nDireccion: "+self.direccion

        return datosCliente

class Cliente(Persona):
    numClientes=0

    def __init__(self,nombre, direccion, email, clave):
        Cliente.numClientes=Cliente.numClientes+1
        super().__init__(nombre,direccion) #invoca al constructor del padre
        self.id="CL"+str(Cliente.numClientes)

        self.email=email
        self.__clave=self.encriptarClave(email,clave)

    def encriptarClave(self,email,clave):
        import hashlib
        aux=email+clave
        claveEncriptada=hashlib.sha3_512(aux.encode('utf-8')).hexdigest()
        return claveEncriptada

    def getClave(self):
        return self.__clave

    def __str__(self):
        datosCliente="El cliente es: "
        datosCliente+="\nId de cliente: "+self.id
        datosCliente+="\nNombre: "+super().getNombre()
        datosCliente+="\nDireccion: "+super().getDireccion()
        #en la sentencia anterior se invoca al método del padre
        datosCliente+="\nEmail: "+self.email
        return datosCliente

```

Figura 19. Definición de jerarquía de clases. Fuente elaboración propia.

A continuación, se muestra el uso de estas clases y se presenta la salida obtenida. La comprobación con `isinstance` es importante, porque el método `getClave()` solo está definido en la clase «Cliente» y así se evita que de error. Luego se verá en otros ejemplos como la conversión de tipos o *cast* en la herencia puede evitar esta consulta.

```

persona=Persona("María","Tribulete, 1") # se crea una persona
cliente=Cliente("Pedro","Carretas, 7","mam@de.es","1345") # se crea un cliente

print(persona) # se imprime los datos de la persona
print()
print(cliente) # se imprime los datos del cliente
print()
if isinstance(cliente, Cliente):
    print("La clave: "+cliente.getClave()+" es de ",end="") #este metodo solo está en la derivada
    print(cliente.getNombre()) #usa método de cliente #uso método heredado
    print()

Salida:

Los datos de la persona son:
Nombre: María
Dirección: Tribulete, 1

El cliente es:
Id de cliente: CL1
Nombre: Pedro
Dirección: Carretas, 7
Email: mam@de.es

La clave: 9792f22da7790844eccb80e042aa96d08bd5e70e44014137ff625284d11885fbc5dd25c620148f5818e8eb333476b3ad0e97855b230c8147b77fced3488279 es de Pedro

```

Figura 20. Uso de las clases de una jerarquía. Fuente: elaboración propia.

## Polimorfismo

Para implementar el polimorfismo es necesario definir una jerarquía de clases en la que se implementan métodos polimórficos. Este es un **procedimiento que se define en la clase base de una jerarquía y se redefine en las derivadas**, porque reflejan un comportamiento común de los objetos de la jerarquía, pero que se expresa de forma distinta en cada una de las clases. Todos los métodos deberán tener el mismo nombre y los mismos argumentos. Esto también se conoce como **sobrecarga de métodos**.

Este método **se invoca y se enlaza con un objeto en tiempo de ejecución**, es decir, se establece una ligadura dinámica. Según sea el tipo del objeto con el que se invoca se ejecutará un método u otro. Esto permite manejar colecciones de objetos de clases distintas de una jerarquía.

En el Ejercicio 5 se muestra un ejemplo de uso del polimorfismo, pero antes se van a tratar algunos conceptos presentes también en la implementación proporcionada.

## Clases y métodos abstractos

Dependiendo del proceso de generalización o especialización de la jerarquía, en ocasiones, se definen clases en los niveles superiores de la misma que no presentan la suficiente concreción como para definir las completamente. Sin embargo, estas

clases conceptualmente son importantes porque **permiten la clasificación semántica de las clases agrupándolas por categorías.**

Estas clases se conocen como **clases abstractas**, son clases no «instanciables» porque **algunos de sus métodos no están implementados**, es decir, son métodos abstractos. Muchos de estos representan un comportamiento común de la jerarquía y deberán ser definidos en las clases derivadas más concretas. Serán, por tanto, métodos polimórficos.

Para implementar clases abstractas en Python se debe:

- ▶ Importar la clase ABC y el decorador `abstractmethod`.
- ▶ La clase base se declara como derivada de la clase ABC.
- ▶ La especificación del método abstracto debe ir precedida de `@abstractmethod`.

---

Para profundizar en las clases abstractas pueden consultar la [librería](#) de la página oficial de Python.

---

En el cuaderno de ejercicios se muestra un ejemplo de implementación de una parte de una jerarquía de figuras.

## 2.9. Referencias bibliográficas

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Benjamin Cummings.

Booch, G., Rumbaugh, J., y Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2ª ed.). Addison-Wesley.

Fernández-Montoro, A. (2013). *Python 3 al descubierto* (2ª ed.). RC Libros.

Joyanes-Aguilar, L. (1998). *Programación orientada a objetos*. McGraw-Hill.

Liskov, B. H. (5-7 de diciembre de 1972) *A design methodology for reliable software systems* [Presentación en papel]. AFIPS '72 (Fall, part I): Proceedings of the December 15-17, 1972, fall joint computer conference, part I, pp. 191-199.  
<https://doi.org/10.1145/1479992.1480018>

Octoverse (2020). *Top languages over the years* [Imagen]. Octoverse.  
<https://octoverse.github.com/>

## 2.10. Cuaderno de ejercicios

### Ejercicio 1. Simular un switch-case en Python

Se quiere implementar un código que lea un valor numérico entre uno y doce e imprima el nombre del mes correspondiente.

La primera opción es la más sencilla, pero tiene los siguientes inconvenientes:

- ▶ Evalúa todas las opciones, aunque se haya cumplido una de ellas.
- ▶ Se debe crear una condición para el caso de error, al tener varios `if`, el `else` solo se ejecutaría para uno de ellos (el último).



```

print ("Introduzca un número entre 1 y 12 para saber el nombre del mes correspondiente: ")
mes=int(input())
"""Se realiza una conversión de la entrada a entero
   porque input convierte la entrada en una cadena
   """

if mes == 1:
    print('El mes es enero')
if mes == 2:
    print('El mes es febrero')
if mes == 3:
    print('El mes es marzo')
if mes == 4:
    print('El mes es abril')
if mes == 5:
    print('El mes es mayo')
if mes == 6:
    print('El mes es junio')
if mes == 7:
    print('El mes es julio')
if mes == 8:
    print('El mes es agosto')
if mes == 9:
    print('El mes es septiembre')
if mes == 10:
    print('El mes es octubre')
if mes == 11:
    print('El mes es noviembre')
if mes == 12:
    print('El mes es diciembre')
if mes<1 or mes>12: #Se tiene que comprobar porque todos son if sin else
    print('El valor del número introducido no está en el rango')

```

Figura 21. Código de implementación del ejercicio 1. Opción 1. Fuente: elaboración propia.

La segunda opción usará la sentencia `if..elif..else`. Es más eficiente que el anterior porque no se evalúan todas las condiciones (en cuanto una se cumple se ejecuta y acaba) y el caso de error no requiere condición, ya que se implementa con un `else`.

Sin embargo, sigue teniendo inconvenientes:

- ▶ El código es repetitivo porque se deben escribir todas las condiciones.
- ▶ Cada opción es excluyente.

```

print ("Introduzca un número entre 1 y 12 para saber el nombre del mes correspondiente: ")
mes=int(input())
"""Se realiza una conversión de la entrada a entero
   porque input convierte la entrada en una cadena
   """

if mes == 1:
    print('El mes es enero')
elif mes == 2:
    print('El mes es febrero')
elif mes == 3:
    print('El mes es marzo')
elif mes == 4:
    print('El mes es abril')
elif mes == 5:
    print('El mes es mayo')
elif mes == 6:
    print('El mes es junio')
elif mes == 7:
    print('El mes es julio')
elif mes == 8:
    print('El mes es agosto')
elif mes == 9:
    print('El mes es septiembre')
elif mes == 10:
    print('El mes es octubre')
elif mes == 11:
    print('El mes es noviembre')
elif mes == 12:
    print('El mes es diciembre')
else:
    print('El valor del número introducido no está en el rango')

```

Figura 22. Código de implementación del ejercicio 1. Opción 2. Fuente: elaboración propia.

La tercera opción es muy eficiente, ya que hace uso de un diccionario. Dado que los switch-case se suelen usar con un conjunto de valores determinados, esos valores pueden ser la clave de un diccionario (en el caso de ejemplo podríamos también usar listas porque el valor del mes es entero, teniendo en cuenta que habría sumar uno al índice).

```

""" se crea el diccionario """
meses={1:"enero",2:"febrero",3:"marzo",4:"abril",5:"mayo",6:"junio",7:"julio",8:"agosto",9:"septiembre",10:"octubre",11:"noviembre",12:"diciembre"}

print ("Introduzca un número entre 1 y 12 para saber el nombre del mes correspondiente: ")
mes=int(input())
"""Se realiza una conversión de la entrada a entero
   porque input convierte la entrada en una cadena
   """

if mes <1 or mes>12:
    print('El valor del número introducido no está en el rango')
else:
    print('El mes es',meses[mes])

```

Figura 23. Código de implementación del ejercicio 1. Opción 3. Fuente: elaboración propia.

## Ejercicio 2. Construir una lista con los valores de un diccionario

En este ejercicio se va a mostrar cómo construir una lista con los nombres de los meses iterando sobre el diccionario, que relaciona cada número del uno al doce con el nombre de mes. Hay dos formas de hacerlo:

- Se itera sobre el diccionario y se añaden los elementos a la lista, que se ha creado vacía al principio, usando la función `add()`.

```
""" se crea el diccionario """
meses={1:"enero",2:"febrero",3:"marzo",4:"abril",5:"mayo",6:"junio",7:"julio",8:"agosto",9:"septiembre",10:"octubre",11:"noviembre",12:"diciembre"}

listaNombresMeses=list()          #se crea la lista vacia sobre la que se van a añadir los valores (es importante la inicialización)
for c,v in meses.items():
    listaNombresMeses.append(v)   #se añade al final de la lista un elemento en cada iteración
```

Figura 24. Código de implementación del ejercicio 2. Opción 1. Fuente: elaboración propia.

La lista creada estará formada por ['enero', 'febrero', 'marzo', 'abril', 'mayo', 'junio', 'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', 'diciembre'].

- El valor extraído en la iteración se inserta automáticamente en la lista. No hace falta inicializar la lista antes ni usar la función `add()`.

```
""" se crea el diccionario """
meses={1:"enero",2:"febrero",3:"marzo",4:"abril",5:"mayo",6:"junio",7:"julio",8:"agosto",9:"septiembre",10:"octubre",11:"noviembre",12:"diciembre"}

listaNombresMeses=[v for c,v in meses.items()]
```

Figura 25. Código de implementación del ejercicio 2. Opción 2. Fuente: elaboración propia.

La lista creada estará formada por ['enero', 'febrero', 'marzo', 'abril', 'mayo', 'junio', 'julio', 'agosto', 'septiembre', 'octubre', 'noviembre', 'diciembre'].

### Ejercicio 3. Implementación de las funciones máximo común divisor y mínimo común múltiplo

En este ejercicio se van a definir las funciones que calculan el máximo común divisor y el mínimo común múltiplo de dos números. Para simplificar la implementación de las funciones más complejas se hará uso de las otras funciones definidas si es el caso.

```
def mcd(num1, num2):  
    """  
  
    Parametros:  
    num1 : numero de tipo entero  
    num2 : numero de tipo entero  
  
    Se calcula usando el algoritmo de Euclides  
  
    Resultado:  
    Máximo común divisor de los dos números  
  
    """  
    aux1=max(num1,num2)  
    aux2=min(num1,num2)  
    while (aux2!=0):  
        mcd=aux2  
        aux2=aux1%aux2  
        aux1=mcd  
    return mcd
```

Figura 26. Implementación del máximo común divisor. Fuente: elaboración propia.

```
def mcm(num1, num2):  
    """  
  
    Parametros:  
    num1 : numero de tipo entero  
    num2 : numero de tipo entero  
  
    Resultado:  
    Mínimo común múltiplo de los dos números  
  
    """  
    aux1=max(num1,num2)  
    aux2=min(num1,num2)  
    return int((aux1/mcd(aux1,aux2))*aux2)
```

Figura 27. Implementación del mínimo común múltiplo. Fuente: elaboración propia.

```

help(mcm)
print(" ")
print(mcm(28,34))

```

Figura 28. Llamada a la función mcm. Fuente: elaboración propia.

```

Help on function mcm in module __main__:
mcm(num1, num2)
  Parametros:
    num1 : numero de tipo entero
    num2 : numero de tipo entero

  Resultado:
    Mínimo común múltiplo de los dos números
476

```

Figura 29. Salida de la ejecución. Fuente: elaboración propia.

Si se observan, todas las funciones llevan unos comentarios justo debajo de la cabecera, estos se conocen como docstring, que informan sobre la función. Para obtener esa información se usar `help(nombre_funcion)`.

#### Ejercicio 4. Implementación de una función que convierta grados en radianes y viceversa

Las funciones que se van a implementar no tienen gran utilidad, pero sirven para mostrar el uso de valores por defecto en los argumentos. Se ha implementado una función para convertir un ángulo en grados a radianes y viceversa. Como lo más habitual es la conversión a radianes, se considera que es el valor por defecto. La otra función imprime una tabla con la conversión de los ángulos principales, de grados a radianes y de radianes a grados.

```

import math

def conversionMedidasAngulo(angulo,magnitud="radianes"):
    """
    Parametros
    angulo : tipo float (grados o radianes)
              el ángulo cuyo valor se quiere convertir.
    magnitud : str, optional
              la magnitud de salida. The default is "radianes".

    Resultado:
    Valor del angulo en la magnitud indicada.
    """
    if magnitud=="radianes":
        return math.radians(angulo)
    else:
        return math.degrees(angulo)

def tablaEquivalenciasAngulos():
    """
    Returns
    None.
    """
    angulosGrados=[0,30,45,60,90,180,270,360]
    print('{:^7} {:^7}'.format('Grados', 'Radianes'))

    for deg in angulosGrados:
        print('{:^7d} {:^7.2f}'.format(deg, conversionMedidasAngulo(deg)))

    angulosRadianes=[0,math.pi/6,math.pi/4,math.pi/3,math.pi/2,math.pi,3*math.pi/2,2*math.pi]
    print('{:^7} {:^7}'.format('Radianes', 'Grados'))

    for rad in angulosRadianes:
        print('{:^7.2f} {:^7.2f}'.format(rad, conversionMedidasAngulo(rad,"grados")))

tablaEquivalenciasAngulos()

```

Figura 30. Código de implementación del ejercicio 4. Fuente: elaboración propia.

El resultado obtenido es el siguiente:

Grados	Radianes
0	0.00
30	0.52
45	0.79
60	1.05
90	1.57
180	3.14
270	4.71
360	6.28
Radianes	Grados
0.00	0.00
0.52	30.00
0.79	45.00
1.05	60.00
1.57	90.00
3.14	180.00
4.71	270.00
6.28	360.00

Figura 31. Salida de la ejecución del código del ejercicio 4. Fuente: elaboración propia.

En el ejemplo se ha usado la biblioteca `math`, que es un módulo de Python en el que están implementadas las principales funciones matemáticas y trigonométricas, además de constantes matemáticas como el número  $\pi$  y el número  $e$ .

Se observa que la función de conversión ha sido invocada con un parámetro (se usa el valor por defecto) y con dos, en los que no se tiene en cuenta el valor por defecto.

---

Para conocer a fondo la biblioteca `math` y otras relacionadas pueden consultar la [librería digital](#) de la página oficial de Python.

---

## Ejercicio 5. Implementación de jerarquía de cuadriláteros y cálculo de sus áreas

La implementación de jerarquías de figuras geométricas es un problema clásico que permite mostrar el uso de la herencia en problemas orientados a objetos y la aplicación del polimorfismo. En la figura 32 se muestra una jerarquía bastante completa en la que las figuras se manejan en un tablero de dibujo. Este modelo está realizado en lenguaje unificado de modelado (UML, por sus siglas en inglés) (Booch *et al.*, 2005).

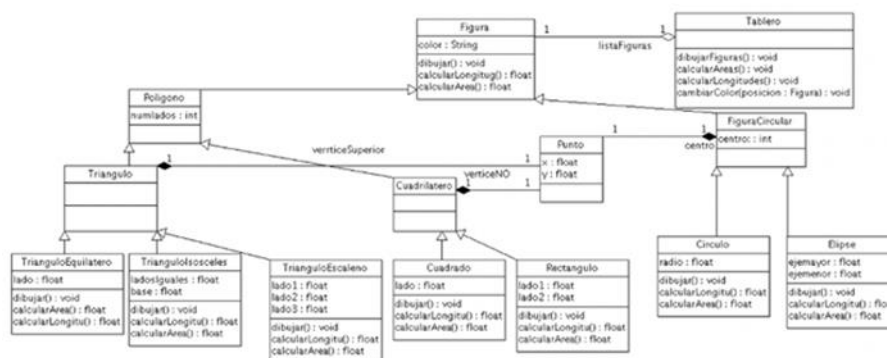


Figura 32. Jerarquía de figuras geométricas en UML. Fuente: Booch *et al.*, 2005.

Se quiere implementar una jerarquía de cuadriláteros, todos ellos tienen definido un color y se pretende implementar un método para calcular el área de los distintos

cuadriláteros, además de manejar una lista de estos, cuyos datos serán introducidos por el usuario. Por tanto, cada ejecución será distinta, pero en todas las figuras serán manejadas a través de la lista.

```
from abc import ABC, abstractmethod

class Cuadrilatero(ABC):

    def __init__(self, color):
        self.color = color

    def getColor(self):
        return color

    @abstractmethod
    def area(self):
        pass #metodo abstracto y polimórfico
            #sentencia nula

    def __str__(self):
        pass

class Rectangulo(Cuadrilatero):

    def __init__(self, base, altura, color):
        Cuadrilatero.__init__(self, color) #invoca al constructor superior
        #esto es lo mismo que super().__init__(color)
        #en el caso de la herencia multiple ayuda a la cualificación
        self.base = base
        self.altura = altura

    def area(self):
        #redefinición del método abstracto de la base
        return self.base * self.altura

    def __str__(self):
        texto = self.__class__.__name__
        texto += "\nSu base es " + str(self.base) + "\nSu altura es " + str(self.altura)
        texto += "\nSu color es " + str(super().getColor())
        return texto
```

Figura 33. Código de implementación del ejercicio 5. Clases Cuadrilátero y Rectángulo. Fuente: elaboración propia.

En la Figura 33 se muestra el código de implementación de la clase base abstracta con el método abstracto y polimórfico area.

En el constructor de la clase Rectangulo se invoca al constructor de la base, esta vez con una sintaxis equivalente a la usada en otros ejemplos Cuadrilatero.\_\_init\_\_(self, color). Esta sintaxis, al usar explícitamente el nombre de la clase base, es aconsejable en la herencia múltiple porque cualifica la invocación al contrario que super().\_\_int\_\_, que se llama igual para todas las clases.

En esta clase se redefine el método polimórfico area, además del método \_\_str\_\_.



```

class Cuadrado(Cuadrilatero):
    def __init__(self,lado,color):
        super().__init__(color)
        self.lado=lado

    def area(self): #redefinición del método abstracto de la base
        return self.lado**2

    def __str__(self):
        texto=self.__class__.__name__
        texto+="\nSu lado es "+str(self.lado)
        texto+="\nSu color es "+str(super().getColor())
        return texto

class Trapecio(Cuadrilatero):
    def __init__(self,baseM,baseM, altura,color):
        Cuadrilatero.__init__(self,color)
        self.baseM=baseM
        self.baseM=baseM
        self.altura=altura

    def area(self): #redefinición del método abstracto de la base
        return self.altura*(self.baseM+self.baseM)/2

    def __str__(self):
        texto=self.__class__.__name__ #nombre de la clase del objeto
        texto+="\nSu base mayor es "+str(self.baseM)+"\nSu base menor es "+str(self.baseM)+"\nSu altura es "+str(self.altura)
        texto+="\nSu color es "+str(super().getColor())
        return texto

```

Figura 34. Código de implementación del ejercicio 5. Clases Cuadrado y Trapecio. Fuente: elaboración propia.

En la Figura 34 se muestra la implementación de las clases Cuadrado y Trapecio de forma similar a Rectángulo, redefiniendo los métodos necesarios.

```

#implementación del polimorfismo
listaFiguras=list() #lista en la que se manejarán las figuras
numFiguras=int(input("Indique cuántas figuras quiere insertar: "))
for i in range(numFiguras): #for para insercion de figuras, en cada ejecución serán distintas
    tipo=input("Inserte el tipo de Figura: R(rectangulo), C(cuadrado), T(trapezio): ")
    if tipo=="R":
        color=input("Inserte el color de la figura: ")
        base=float(input("Inserte el valor de la base: "))
        altura=float(input("Inserte el valor de la altura: "))
        listaFiguras.append(Rectangulo(base,altura,color))
    elif tipo=="C":
        color=input("Inserte el color de la figura: ")
        lado=float(input("Inserte el valor del lado: "))
        listaFiguras.append(Cuadrado(lado,color))
    else:
        color=input("Inserte el color de la figura: ")
        baseM=float(input("Inserte el valor de la base mayor: "))
        baseM=float(input("Inserte el valor de la base menor: "))
        altura=float(input("Inserte el valor de la altura: "))
        listaFiguras.append(Trapecio(baseM,baseM,altura,color))

for figura in listaFiguras:
    texto=""
    #se recorre la lista
    texto+="\n\nLa figura "+str(int(listaFiguras.index(figura)+1))+" es un "
    print(texto,end="")
    print(figura)
    print("Su area es: "+str(figura.area())) #se invocará a un método area de la clase del objeto que se enlaza en tiempo de ejecución

```

Figura 35. Código de implementación del ejercicio 5. Implementación de la lista de figuras y del polimorfismo. Fuente: elaboración propia.

En la Figura 35 se muestra el código que crea la lista de figuras (cuadriláteros) que cambiará con cada ejecución. En el recorrido de la lista se invocará al método polimórfico `area`. La forma de invocación `figura.area()` es transparente a lo que de

verdad se invoca, ya que, dependiendo del tipo del objeto que se encuentre en la lista en cada momento, se invocará a un método de la clase del objeto que se liga en de forma dinámica, es decir, en tiempo de ejecución.

```
Indique cuantas figuras quiere insertar: 3
Inserte el tipo de Figura: R(rectangulo), C(cuadrado), T(trapezio): R
Inserte el color de la figura: Rojo
Inserte el valor de la base: 2
Inserte el valor de la altura: 3
Inserte el tipo de Figura: R(rectangulo), C(cuadrado), T(trapezio): T
Inserte el color de la figura: Verde
Inserte el valor de la base mayor: 7
Inserte el valor de la base menor: 3
Inserte el valor de la altura: 8
Inserte el tipo de Figura: R(rectangulo), C(cuadrado), T(trapezio): C
Inserte el color de la figura: Azul
Inserte el valor del lado: 4

La figura 1 es un Rectangulo
Su base es 2.0
Su altura es 3.0
Su color es Azul
Su area es: 6.0

La figura 2 es un Trapecio
Su base mayor es 7.0
Su base menor es 3.0
Su altura es 8.0
Su color es Azul
Su area es: 40.0

La figura 3 es un Cuadrado
Su lado es 4.0
Su color es Azul
Su area es: 16.0
```

Figura 36. Salida del programa implementado. Fuente: elaboración propia.

## Documentación de Python – 3.9.2

Documento de Python (<https://docs.python.org/es/3/>).

Documentación de la última versión de Python, proporcionada por Python Software Foundation.

## Tutorial de Python

Python (s. f.). *Tutorial de Python* (3.9.6) [Tutoriales]. Python. <https://docs.python.org/es/3/tutorial/index.html>

Tutorial en español de la Python Software Foundation que le ayudará a profundizar en algunos de los conceptos tratados en el tema.

## Manuales y recursos de entrenamiento para la programación en Python

Programación en Python – Nivel básico (<https://entrenamiento-python-basico.readthedocs.io/es/latest/index.html>)

Existe un importante repositorio de manuales y recursos para entrenamiento y aprendizaje de la programación con Python.

## Tutorial sobre funciones de entrada y salida en Python

Python (s. f.). 7. *Entrada y salida* (3.9.6) [Tutoriales]. Python.  
<https://docs.python.org/es/3/tutorial/inputoutput.html>

En los códigos de ejemplo se han usado operaciones de lectura y escritura básicas mediante el uso de las funciones `input()` y `print()`. Pero existe muchas más opciones y formas de manipular las salidas, este tutorial de Python muestra algunas.

## Tutorial sobre el manejo de archivos

Python (s. f.). 7.2. *Leyendo y escribiendo archivos* (3.9.6) [Tutoriales]. Python.  
<https://docs.python.org/es/3/tutorial/inputoutput.html#reading-and-writing-files>

El uso de archivo para obtener datos y almacenarlos es fundamental en el tipo de aplicaciones que se implementan con Python, en el que se suelen manejar grandes cantidades de datos. El siguiente tutorial ayuda a profundizar sobre su uso.

## Tutorial sobre los módulos de Python

Python (s. f.). 6. *Módulos* (3.9.6) [Tutoriales]. Python.  
<https://docs.python.org/es/3/tutorial/modules.html>

Las aplicaciones de gran tamaño y complejidad requieren del uso de muchos recursos distintos y, cuanto mayor sea la especialización, más específicas son las operaciones que se deben usar.

Por ello, ya desde hace muchos años, se utilizan los lenguajes de diseño para construir aplicaciones mediante el ensamblaje de módulos como recursos especializados. En Python se cuentan con muchos módulos o bibliotecas predefinidas. Consulte el siguiente tutorial para profundizar sobre ello.

## Tutorial POO

Python (s. f.). 9. *Clases* (3.9.6) [Tutoriales]. Python.  
<https://docs.python.org/es/3.8/tutorial/classes.html>

La programación POO a objetos cuenta con muchas características, demasiadas para abordar en su totalidad en este tema, pero este tutorial puede ayudar a completar la información.

1. El lenguaje Python:
  - A. Es un lenguaje multiparadigma.
  - B. Es un lenguaje funcional.
  - C. Es un lenguaje declarativo.
  - D. Solo soporta el paradigma de programación imperativa.
  
2. En Python el tipo de una variable:
  - A. Se asigna en tiempo de compilación.
  - B. Se asigna en tiempo de ejecución.
  - C. No existe.
  - D. Solo puede ser numérico.
  
3. Los tipos de datos compuestos que son secuencias ordenadas son:
  - A. Los diccionarios y los conjuntos.
  - B. Las listas y las tuplas.
  - C. Todos son secuencias ordenadas.
  - D. Ninguno son secuencias ordenadas.

4. El tipo en Python para los que no se pueden cambiar los valores son:
- A. Las listas.
  - B. Las tuplas.
  - C. Los conjuntos.
  - D. Los diccionarios.
5. Los tipos numéricos en Python:
- A. Ninguno es compatible con otro.
  - B. Solo son compatibles el entero y el complejo.
  - C. Solo son compatibles el complejo y el float.
  - D. Son todos compatibles.
6. Si se tiene definido un diccionario, se puede recorrer mediante un for:
- A. Solo las claves.
  - B. Solo los valores.
  - C. Solo los pares.
  - D. Las claves, los valores y elementos o pares.
7. Las funciones en Python:
- A. No tienen valores por defecto en los parámetros.
  - B. Pueden tener parámetros por defecto que se asignan del primero al último.
  - C. Pueden tener parámetros por defecto que se asignan del último al primero.
  - D. Pueden tener parámetros por defecto que se asignan en cualquier posición.
8. El sistema de excepciones en Python:
- A. Solo controla excepciones predefinidas.
  - B. Solo controla excepciones numéricas.
  - C. Controla también excepciones definidas por el usuario.
  - D. Solo controla excepciones de entrada/salida de archivos.

9. Dado el siguiente atributo de una clase definido en el constructor como `__id`.
- A. Es un atributo público.
  - B. No se pueden usar identificadores como este.
  - C. Es un atributo de clase.
  - D. Es un atributo oculto.
10. El polimorfismo se implementa:
- A. Redefiniendo un método en las clases de una jerarquía.
  - B. Cambiando los tipos de las variables.
  - C. Definiendo métodos ocultos en una clase.
  - D. Definiendo métodos ocultos en una jerarquía de clases.