

Documentación curso PyQGIS

version testing

Damián Ortega Terol

abril 03, 2023

Contenido

Welcome to QGIS Python Programming Recipes's documentation!	1
Introducción	1
Documentación de referencia	2
Documentación oficial QGIS de referencia	2
<i>PyQGIS online resources</i> (Webgrafía)	3
Interactuar con otros usuarios	3
Resolución de dudas	3
Reporte de bugs	3
Blogs y foros	3
En otros idiomas	4
En castellano	4
Otros recursos interesantes	4
Libros de referencia	5
Apuntes Lenguaje de programación Python	5
Parte I: Introduccion a Python	5
Uso de Python en GIS	5
Recursos para el aprendizaje del lenguaje de programación Python	6
Recursos en línea en castellano	6
Recursos en línea en otros idiomas	7
Características principales de Python	7
Variables	8
Definición de variables	8
Definición de constantes	9
Tipos básicos de variables	9
Números	9
Enteros	10
Reales	10
Operadores aritméticos	10
Funciones de conversión entre números	10
Cadenas de caracteres	11
Caracteres especiales	11
Métodos de objeto cadenas de texto	12
Booleanos	13
Operadores lógicos o condicionales	13
Operadores relacionales o de comparación	13
Eliminación de una variable	14
Reglas de nomenclatura de variables	14
Asignación multiple	15
Asignaciones aumentadas	15
Definición de comentarios	16

Estructuras o colecciones de datos	16
Listas	16
Métodos del objeto lista	18
Tuplas	18
Diccionarios	19
Métodos del objeto diccionario	21
Control de flujo	22
Sentencias condicionales	22
Estructura condicional simple <code>if</code>	22
Estructura condicional doble <code>if ... else</code>	22
Estructura condicional múltiple <code>if ... elif ... elif ... else</code>	23
Bucles o estructuras repetitivas o iterativas	23
Bucle indefinido: <code>while</code>	24
Bucle definido: <code>for .. in</code>	24
Control de bucles: sentencias <code>break</code> , <code>continue</code> , <code>pass</code>	24
Funciones definidas por el usuario	26
Definición de funciones con un solo argumento	26
Definición y uso de funciones con varios argumentos	27
Definición y uso de funciones sin argumentos	27
Definición de funciones con devolución de valores	28
Funciones con argumentos por defecto	29
Funciones con argumentos de longitud variable	29
Paso de variables por referencia o por valor	29
Ámbito de las variables: variables locales y globales	30
Programación orientada a objetos en Python	31
Paradigma de POO	31
Conceptos básicos de la POO	32
Implementación de clases en Python	33
Características de la POO	34
Ventajas de la POO	38
Excepciones	38
Módulos y paquetes	40
Módulos	40
Paquetes	40
Escritura y lectura de ficheros de texto	40
Formateo de la salida	40
Manejo de ficheros de texto	41
Escritura de ficheros	42
Lectura de ficheros	42
Mover el puntero de lectura/escritura	42
Extensión de la funcionalidad de QGIS mediante plugins	43
Introducción	43

Complementos del núcleo de QGIS	43
Complementos externos a QGIS	43
Inicialización de complementos	43
Complementos recomendados	43
Ideas finales	43
Descripción de las APIs de QGIS: API C++ y API Python	43
Introducción	43
La API de QGIS explicada a través de ejemplos	43
Consulta de la API QGIS en la consola de Python de QGIS	43
Configuración de QGIS y PyCharm para el desarrollo de plugins	43
Instalación de QGIS	43
Instalación de PyCharm	43
Configuración del entorno de desarrollo de QGIS	43
Primeros pasos con PyCharm	43
Estilo de codificación propuesto	43
Desarrollo de complementos en QGIS	43
Creación de un complemento básico de QGIS	44
Diseño y creación de la Interfaz de Usuario	44
Implementación de la funcionalidad del complemento	44
Indices and tables	44

Welcome to QGIS Python Programming Recipes's documentation!

Introducción

En este sitio encontrarás documentación en idioma español específica para el desarrollo de complementos en QGIS en lenguaje Python, complementaria a los recursos ofrecidos en el [sitio oficial del proyecto QGIS](#). Esta documentación ha sido construida con la herramienta [Sphinx](#) escrita por Georg Brandl bajo licencia BSD, utilizando el tema provisto por [Read the Docs](#).

Note

Si lo prefieres puedes descargar esta documentación en formato [pdf](#).

A continuación, se aportan unas indicaciones sobre la notación empleada en este documento:

- Se amplían los contenidos de las secciones más importantes con [videotutoriales](#)
- Los conceptos complementarios se agrupan en pestañas para facilitar su consulta:

`continue`

`continue` regresa al comienzo del bucle, ignorando todas las sentencias que quedan en la iteración actual del bucle e inicia la siguiente iteración.

`break`

`break` termina el bucle actual y continua con la ejecución de la siguiente instrucción.

`pass`

`pass` tal como su nombre lo indica es una operación nula, o sea que no pasa nada cuando se ejecuta.

- Se utiliza *tipografía cursiva* en palabras y acrónimos en inglés generalmente provenientes del mundo geomático: *plugin*.
- Los enlaces Web a recursos en línea consultados aparecen en color azul con su correspondiente hipervínculo: [Grupo de usuarios de QGIS España](#).
- Datos y código fuente de los contenidos se encuentran alojados en el siguiente repositorio de [GitHub](#).
- El código fuente de scripts y ficheros de configuración se tipografían con esta *fuerza*, resaltando en amarillo los bloques de código referidos a los conceptos que se están describiendo. Por ejemplo:

```
1 for letra in "PyQGIS":
2     if letra == "Q":
3         continue
4     print("Letra actual: ", letra)
5 print("Adios")
```

- Para enfatizar ciertos aspectos de la documentación se aportan notas, consejos, conceptos importantes y mensajes de error. Por ejemplo:

Tip

El tipo de licencia de QGIS permite inspeccionar el código fuente de los complementos, constituyéndose como un recurso imprescindible y recomendado para el aprendizaje de estas herramientas: *“La mejor escuela es instalar y leer el código de plugins”*.

Note

Nos referiremos a *argumentos* y *parámetros* indistintamente en este tutorial.

Important

El índice del primer elemento de una lista es 0 y no 1.

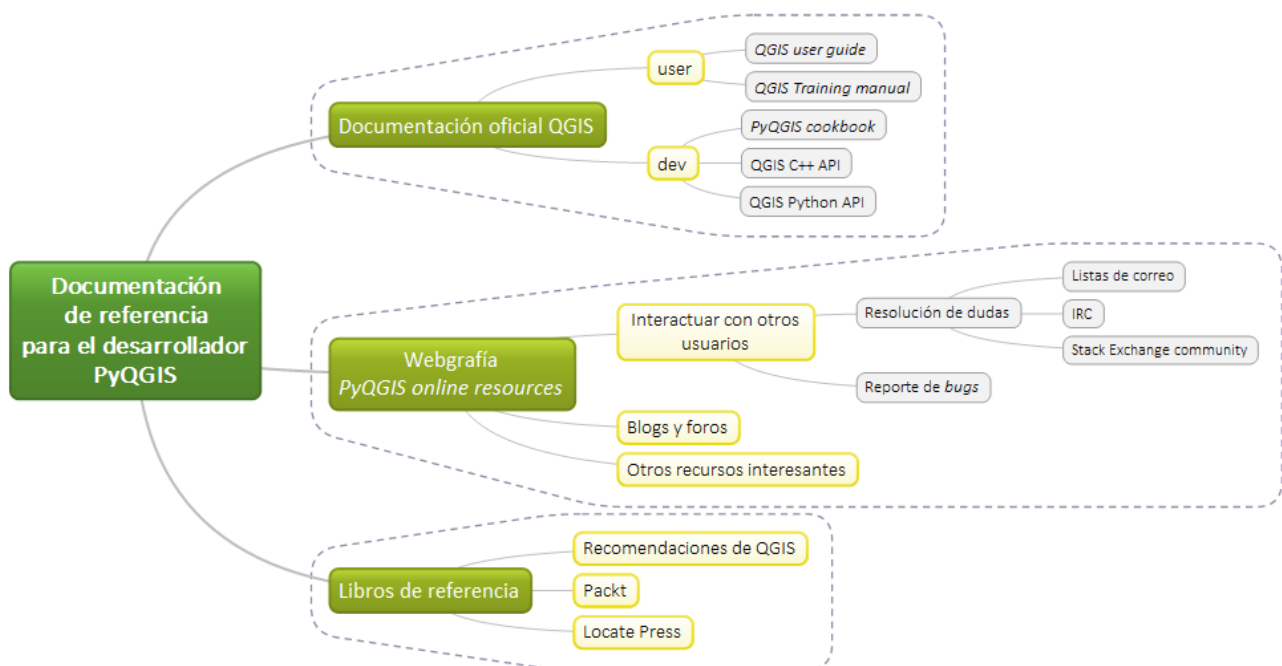
Error

Si se introduce como denominador el valor cero, `divide(2,0)`, obtendremos el siguiente resultado:

```
Denominador cero!!!  
Ejecución cláusula finally
```

Documentación de referencia

En esta sección se facilitan una serie de recursos para el aprendizaje de QGIS, específicamente enfocados a la programación de QGIS con Python. El contenido se estructura en los siguientes apartados:



Documentación oficial QGIS de referencia

La [documentación oficial](#) de QGIS se facilita en idioma inglés, pero algunas partes están traducidas a varios idiomas. Las principales referencias para el **desarrollador** son la [API QGIS C++](#), la [API QGIS Python](#) y el libro de recetas para el desarrollador PyQGIS ([PyQGIS cookbook](#)). Estos recursos se complementan con la guía de usuario ([QGIS User guide](#)) y el manual de aprendizaje ([QGIS Training manual](#)).

Note

De forma genérica, los **manuales de usuario** (*user guide*) son utilizados para describir las características y funcionalidades disponibles en una determinada aplicación informática y su forma de uso. Los manuales de usuario explican todas las pantallas y campos.

Un **manual de aprendizaje** (*training manual*) se redacta de forma diferente y puede estar acompañado de ejercicios para que el usuario los complete. Describen una parte específica del sistema, paso a paso, pero no explican todas las funcionalidades y características de la herramienta informática. Suelen tener más capturas de pantallas que los manuales de usuario.

PyQGIS online resources (Webgrafía)

En este apartado se aportan un conjunto de recursos disponibles en línea de PyQGIS.

Interactuar con otros usuarios

Una de las ventajas de los proyectos de código abierto es que se puede hablar directamente con otros desarrolladores y frecuentemente con los desarrolladores principales del proyecto.

Resolución de dudas

QGIS tiene tres formas oficiales de apoyar el desarrollo y la resolución de problemas. En las listas de correo, canales IRC y redes sociales técnicas se puede obtener soporte de otros usuarios:

1. **Listas de correo:** Las listas de correos están divididas por temas y grupos determinados. Las consultas sobre el desarrollo del núcleo de QGIS o complementos serán dirigidas a la lista de desarrolladores ([Developer list](#)) y para las relacionadas con la discusión de aspectos generales de QGIS se utilizará la lista de usuarios ([User list](#)).
2. **Internet Relay Chat (IRC):** para obtener asistencia de usuarios y desarrolladores en tiempo real se puede acceder al [canal de QGIS](#). Esta ayuda es voluntaria y está condicionada a su disponibilidad.
3. **The Stack Exchange community:** Esta red social técnica tiene un subproyecto GIS (*Geographic Information Systems Stack Exchange*) con 36 etiquetas relacionadas con QGIS y PyQGIS en el siguiente [enlace](#).

Reporte de bugs

Una manera importante de respaldar el proyecto QGIS, además de financiarlo, es mediante el reporte problemas (solicitudes de nuevas funcionalidades o errores) aportando un caso de uso detallado y los datos que permiten a otros replicar el problema, permitiendo acelerar su corrección. Se recomienda seguir las siguientes [indicaciones](#) antes de reportar un error.

Por otro lado, cada parte del proyecto QGIS tiene un lugar específico donde los problemas se pueden informar, gestionar y discutir, dependiendo del área en la que haya encontrado. A continuación, se indican los repositorio adecuados para dirigir las consultas:

- Aplicaciones (QGIS Desktop, QGIS Server) -> <https://github.com/qgis/QGIS/issues>
- Sitio web de QGIS -> <https://github.com/qgis/QGIS-Website/issues>
- Documentación QGIS -> <https://github.com/qgis/QGIS-Documentation/issues>

Para ello será necesario disponer de una [cuenta de GitHub](#).

Los enlaces para reportar errores de complementos externos de terceros se pueden localizar en los repositorios especificados por los autores de los mismos. Los enlaces a dichos repositorios son mostrados en el Administrador de Complementos de QGIS, accesible desde el menú Complementos ■ Administrar e instalar complementos....

Blogs y foros

A continuación, se aporta un listado de Blogs, canales de noticias y foros *activos* de interés para el desarrollador:

En otros idiomas

- [QGIS.org blog](#): Blog oficial del proyecto QGIS.
- [QGIS Plugins Planet](#): Este sitio incluye una lista con múltiples Blogs en el panel izquierdo.
- [Planet OSGeo](#): Planet OSGeo es una ventana al mundo, el trabajo y las vidas de los miembros de OSGeo, *hackers* y colaboradores donde se agregan además Blogs de terceros. Se puede solicitar añadir tu propio Blog este portal.
- [Spatial Galaxy](#): Blog Gary Sherman, [godfather de QGIS](#), sobre GIS open source, QGIS, PyQGIS, Python, programación, etc. en el ámbito geoespacial.
- [Free and Open Source GIS Ramblings](#): Blog de Anita Graser, que formó parte del [Comité Directivo del Proyecto QGIS](#) y de la Junta Directiva de OSGeo. Tiene numerosas publicaciones de referencia sobre QGIS (¹, ²).
- [Blog de OpenGIS](#): Interesante Blog de los desarrolladores de QGIS para Android y QField para QGIS.
- [Blog Lutra consulting](#): Blog de Lutra Consulting, empresa que proporciona servicios de consultoría, migración, desarrollo de software, capacitación y soporte comercial para QGIS y otros proyectos SIG de código abierto.
- [Blog Kartoza](#): Blog de Tim Sutton, miembro honorario del Comité Directivo del Proyecto QGIS. Anterior Linfiniti Geo Blog.
- [Blog de North Road](#): Fundado por Nyal Dawson, analista espacial y miembro establecido de la comunidad de desarrollo de QGIS.
- [Blog de North River Geographic Systems](#): Empresa de consultoría geoespacial, dedicada principalmente al FOSS4G.
- [Blog Oslandia](#): Blog de Oslandia, empresa privada creada por expertos en SIG y datos espaciales.

En castellano

- [Blog de mappingGIS](#): Completo Blog que contiene numerosas entradas relacionadas con la difusión de las Tecnologías de la Información Geográfica. Permanentemente actualizado.
- [El Blog de José Guerrero](#): Especialmente dedicado a GNU/Linux, código Python y Sistemas de Información Geográfica (SIG). En su nube de categorías tiene mucho peso la etiqueta `PyQGIS`.
- [Foro de QGIS en castellano](#): Foro de QGIS en castellano.
- [Blog de GeoTux](#): Soluciones geoinformáticas libres. Geo-noticias, geo-blogs y geo-foros.
- [El blog de Franz](#): Con un interesante apartado de preguntas y respuestas. Muy orientado a ArcGIS.

Otros recursos interesantes

- [QGIS Python Plugins Repository](#): Repositorio de plugins de QGIS. Recurso recomendado para el estudio del código de otros desarrolladores.

Tip

El tipo de licencia de QGIS permite inspeccionar el código fuente de los complementos, constituyéndose como un recurso imprescindible y recomendado para el aprendizaje de estas herramientas: *“La mejor escuela es instalar y leer el código de plugins”*.

- [Asociación QGIS España](#): Soporte para el grupo de usuarios Españoles de QGIS.
- [QGIS Visual Changelog](#): Registro visual de cambios desde la versión QGIS 2.0.
- [QGIS tutorials and tips](#): Blog de Ujaval Gandhi, fundador también del [Spatial Thoughts](#). Ver Python Scripting (PyQGIS).
- [Visual Style Guide](#): Guía de estilo visual de QGIS con la imagen de QGIS.

Libros de referencia

QGIS facilita en su sitio Web una [lista de libros](#) de otros editores, que no administra y se ofrecen sólo como ayuda.

Por su parte, la editorial [Locate Press](#), especializada en libros geospaciales de código abierto, publica varios libros que son referencia en QGIS. A nivel usuario se citan ³, ⁴ y específicos de programación para QGIS se destacan ⁵, ⁶.

Finalmente, la editorial [Packt Publishing](#), especializada en la publicación de recursos de aprendizaje de proyectos de código abierto, contiene numerosos libros y video tutoriales sobre QGIS de nivel usuario hasta nivel experto programador. En este caso se recomiendan los siguientes: ⁷, ⁸

- 1 **A. Graser**, *Learning QGIS - Third Edition*, 3rd Revised edition edition (Packt Publishing - ebooks Account, Birmingham Mumbai, **2016**)
- 2 **A. Graser, G. N. Peterson, G. Sherman**, *QGIS Map Design: With New and Updated Workflows for QGIS 3.4* (Locate Press, Chugiak, AK, **2018**)
- 3 **K. Menke**, *Discover QGIS 3.x: A Workbook for the Classroom or Independent Study* (Locate Press, Chugiak, Arkansas, **2019**)
- 4 **H. van der Kwast and K. Menke**, *QGIS for Hydrological Applications: Recipes for Catchment Hydrology and Water Management* (Locate Press, Chugiak, AK, **2019**)
- 5 **G. Sherman**, *The PyQGIS Programmer's Guide: Extending QGIS 3 with Python 3* (Locate Press, **2018**)
- 6 **T. Mitchell**, *Geospatial Power Tools: GDAL Raster & Vector Commands*; (Locate Press, Chugiak, AK, **2014**)
- 7 **S. Islam, S. Miles, G. Menke, G. Smith, L. Pirelli, G. Van Hoesen, and an O. M. C. Safari**, *Mastering Geospatial Development with QGIS 3.x - Third Edition* (Packt Publishing, **2019**)
- 8 **B. Mearns, A. Mandel, A. Bruy, V. Olaya, and A. Graser**, *QGIS: Becoming a GIS Power User* (Packt Publishing, Birmingham, **2017**).

Apuntes Lenguaje de programación Python

Parte I: Introduccion a Python

Se amplían los contenidos de esta sección en el siguiente [videotutorial](#)

Existen numerosos recursos disponibles para el aprendizaje del lenguaje de programación Python. En esta sección se facilita una introducción de conceptos y descripciones enfocados a la extensión de funcionalidades de QGIS haciendo uso de este lenguaje de programación.

Uso de Python en GIS

El lenguaje Python se ha convertido en el estándar *de facto* para la programación en investigaciones científicas (Millman and Aivazis 2011). Lo anterior también se cumple en el contexto de las aplicaciones geomáticas, tanto en el mundo de los Sistemas de Información Geográfica (SIG) de código abierto como de software propietario, donde Python se constituye como el lenguaje de *scripting* estándar *de facto*: la mayoría de los principales software SIG proporcionan librerías de Python para la integración, análisis y automatización de tareas de geoprocetamiento, incluidos QGIS y PostGIS (Kuiper et al. 2014).

Note

El [índice TIOBE](#) que establece un ranking mensual clasificando los lenguajes de programación más populares del momento, sitúa a Python en mes de marzo de 2023 en primera posición de los lenguajes más populares y demandados en el mercado laboral.

La principal ventaja de Python en relación con otros lenguajes de programación de nivel superior, como C++, es que Python presenta una mayor pendiente en su curva de aprendizaje y una sintaxis simple y clara, favoreciendo el

código legible asimilable al lenguaje natural. Al venir integrado en las herramientas que lo contemplan como lenguaje de *scripting*, el despliegue del entorno de desarrollo es sencillo. Además, la importante comunidad de usuarios se traduce en la disponibilidad de numerosas librerías y aplicaciones, cuya integración no sólo permite incrementar el rendimiento en la implementación del desarrollo, al evitar arrancar de cero, sino que también proporciona estabilidad, al utilizar herramientas que han sido ampliamente probadas en diferentes entornos y contextos.

Sin embargo, Python presenta inconvenientes frente a C++, principalmente que es un lenguaje interpretado o de *script*, que se ejecuta utilizando un programa intermedio llamado intérprete, traduciendo el código instrucción a instrucción a medida que lo va ejecutando (*runtime*), de forma que la compilación no sucede antes del tiempo de ejecución. Esto se traduce en (i) una mayor dificultad en el proceso de depuración de errores al no informar fácilmente sobre éstos hasta que el código se ejecuta y, (ii) una menor velocidad de ejecución, lo que puede ser crítico en el desarrollo de aplicaciones donde el requisito de rendimiento sea necesario, como es el caso de muchas aplicaciones geomáticas, motivado por el gran volumen de información a tratar.

Tip

Para aprovechar las ventajas de ambos lenguajes y minimizar sus problemas, se recomienda utilizar la combinación de ambos lenguajes de programación, embebiendo herramientas escritas en lenguaje C++ para ser utilizadas en aplicaciones implementadas en código Python. Esta técnica presenta la ventaja de se puede mezclar código compilado en C++ y código de Python que se compila en tiempo de ejecución.

Recursos para el aprendizaje del lenguaje de programación Python

La redacción de estos apuntes sobre Python 3 está basada principalmente en los recursos disponibles del tutorial «Python para todos»⁹, mejorados y ampliados con otros recursos consultados de la bibliografía recomendada (¹⁰, ¹¹, ¹², ¹³) y otros disponibles en línea, adaptando sus contenidos al objeto de este tutorial.

- | | |
|----|--|
| 9 | R. González Duque , <i>Python Para Todos</i> (2008) |
| 10 | Óscar Ramírez Jiménez , <i>Python a fondo. Domine el lenguaje del presente y futuro</i> (2021) |
| 11 | G. Sherman , <i>The PyQGIS Programmer's Guide: Extending QGIS 3 with Python 3</i> (Locate Press, 2018). |
| 12 | P. Gerrard , <i>Lean Python: Learn Just Enough Python to Build Useful Tools</i> (Springer Science+Business Media, New York, NY, 2016) |
| 13 | Mohit and B. N. Das , <i>Learn Python in 7 Days: Get up-and-Running with Python</i> (2017) |

Para ampliar conocimientos sobre este lenguaje se recomiendan los siguientes recursos disponibles en línea:

Recursos en línea en castellano

- [Tutorial oficial](#) de Python versión 3.9 en castellano dentro de la [documentación oficial](#) de la [Python Software Foundation](#) (PSF).
- [Introducción a la programación en Python 3](#): Completo manual de la Universitat Jaume I de Castellón.
- [Solución de problemas con algoritmos y estructuras de datos usando Python](#): Traducción de Mauricio Orozco-Alzate del libro de Brad Miller y David Ranum.
- [Python para principiantes](#): Manual en castellano escrito por Eugenia Bahit
- [Python para impacientes](#): Recomendado para programadores con conocimientos.
- [Programación en Python - Nivel básico Covantec](#): Materiales del curso de Programación en Python - Nivel básico realizado por la empresa [Covantec R.L.](#)
- [Materiales curso sencillo de iniciación a Python](#): Estos apuntes del Curso de iniciación a la programación en Python se impartieron en la segunda mitad del curso 2019/2020 en el módulo Lenguaje de Marcas y Sistemas de Gestión de la Información del ciclo formativo Administración de Sistemas Informáticos en Red (ASIR) en el IES Abastos de Valencia (España).

Recursos en línea en otros idiomas

- [A byte of Python](#): Se trata de un libro gratuito sobre programación utilizando el lenguaje Python que sirve como tutorial o guía del lenguaje Python para una audiencia principiante. Se puede descargar una versión en formato pdf o epub desde su [página de GitHub](#).
- [Python 101!](#): La audiencia de este libro son principalmente personas que han programado en el pasado pero que quieren aprender Python. Está organizado en 5 partes que abarcan desde nivel principiante a nivel intermedio.
- [Python Foundation for Spatial Analysis](#): Curso de Ujaval Gandhi con una introducción suave a la programación de Python con un claro enfoque en los datos espaciales.
- [Python Notes for Professionals book](#): Completo libro con más de 800 páginas descargable en formato pdf.
- [Dive Into Python 3](#): Tutorial de Mark Pilgrim para programadores con experiencia.
- [Python for Everybody](#): Introducción a los conceptos básicos de Python 3 con énfasis en el uso práctico por Charles Severance (Dr. Chuck).
- [PyVideo.org](#): Índice de recursos de libre disposición para aprender el lenguaje de programación Python.

Características principales de Python

Python es un lenguaje de programación de alto nivel y de propósito general muy poderoso y flexible, a la vez que sencillo y fácil de aprender. Fue creado por Guido van Rossum y su nombre está inspirado en la serie «*Monty Python's Flying Circus*». La primera versión fue lanzada en febrero de 1991. Está administrado por la [Python Software Foundation](#), todas sus versiones son de [código abierto](#), se distribuye bajo la licencia denominada [Python Software Foundation License](#) y está implementado en la mayoría de plataformas y sistemas operativos habituales.

Python es un lenguaje de alto nivel, de propósito general, multiplataforma, multiparadigma, principalmente imperativo, orientado a objetos y funcional, de tipado dinámico y fuertemente tipado. Se extienden a continuación las definiciones de estas **características principales** a nivel de lenguaje de programación:

- **Propósito general**: Con Python se pueden crear todo tipo de programas: programación a nivel de sistema operativo; aplicaciones con interfaz de usuario utilizando por ejemplo el *binding* de Qt para Python (PyQt); aplicaciones Web e interacción con servicios Web; aplicaciones científicas; gestión de contenido; *Big Data*, inteligencia artificial y *deep learning*.
- **Interpretado o de script**: Se ejecuta utilizando un programa intermedio llamado intérprete, que traduce el código instrucción por instrucción a medida que lo va ejecutando, en lugar de compilar el código a código máquina que pueda comprender y ejecutar directamente una computadora (lenguajes compilados). Esto proporciona ventajas como la rapidez de desarrollo al no tener que compilar los programas cada vez que se modifica el código e inconvenientes como una menor velocidad de ejecución. Esta característica permite que el código no sea dependiente del hardware en el que se ejecuta, y ayuda a que el lenguaje sea **Multiplataforma**, aunque originalmente se desarrolló para Unix. En Python, la primera vez que se ejecuta un *script* se compila, traduciendo el código fuente a pseudocódigo máquina intermedio llamado *bytecode*, generando archivos `.pyc`, que son los que se ejecutarán en sucesivas ocasiones.
- **Con tipado dinámico**: No es necesario declarar el tipo de dato que va a contener una determinada variable, sino que su tipo se determinará en tiempo de ejecución según el tipo del valor al que se asigne, y el tipo de esta variable puede cambiar si se le asigna un valor de otro tipo.
- **Organizado y extensible**: Dispone de múltiples formas de organizar código tales como funciones, clases, módulos y paquetes. Si hay áreas que son lentas se pueden reemplazar por plugins en C o C++, siguiendo la API para extender Python en una aplicación.
- **Fuertemente tipado**: No se permite tratar a una variable como si fuera de un tipo distinto al que tiene, es necesario convertir de forma explícita dicha variable al nuevo tipo previamente.
- **Orientado a objetos**: La orientación a objetos es un paradigma de programación en el que los conceptos del mundo real relevantes para nuestro problema se trasladan a clases y objetos en nuestro programa. Esto facilita el desarrollo de programas con componentes reutilizables. En el capítulo de programación orientada a objetos en Python se abordan los conceptos más importantes de este paradigma de programación.

Respecto a las **herramientas básicas** para la implementación del código fuente, existen dos formas de ejecutar código Python: (i) se pueden escribir líneas de código en el intérprete y obtener una respuesta para cada línea

(sesión *interactiva*) o bien (ii) se puede escribir el código de un programa en un archivo de texto (extensión `.py`) y ejecutarlo. En uno y otro caso, se utilizarán en este tutorial la consola de Python de QGIS y el IDE PyCharm versión *Community Edition*.

Important

En Python hay dos formas de ejecutar un programa: mediante una sesión interactiva utilizando el *interactive interpreter prompt* o utilizando un fichero con el código fuente.

Pastebin

Además de los IDEs existen otras opciones para implementar código desde un navegador Web. repl.it es una plataforma ideal para programar y no es necesario instalar ni pagar nada para utilizarla. En esta misma línea existen aplicaciones Web que permiten a los usuarios subir pequeños textos, generalmente ejemplos de código fuente, para que estén visibles al público en general. [Linkcode](https://linkcode.io) es un ejemplo de *pastebin*.

Otras **ventajas** de este lenguaje son su sintaxis simple, clara y sencilla, el gestor de memoria, la gran cantidad de librerías disponibles (más de 100.000 paquetes de librerías han sido compartidas por su comunidad de usuarios) y la potencia del lenguaje, entre otros, que hacen que desarrollar una aplicación en Python sea sencillo, muy rápido y, lo que es más importante, divertido.

La sintaxis de Python es tan sencilla y cercana al lenguaje natural que los programas elaborados en Python parecen pseudocódigo. Por este motivo se trata además de uno de los mejores lenguajes para aprender y entender la programación.

Python no es adecuado sin embargo para la programación de bajo nivel o para aplicaciones en las que el rendimiento sea crítico.

Otro aspecto característico de Python es el concepto de **indentación**, característica singular de este lenguaje. La sangría no solo hace que el código de Python sea legible, sino que también distingue cada bloque de código del otro. Es muy común utilizar una indentación de 4 espacios.

Finalmente es necesario reseñar que la **versión** del intérprete de **Python** utilizada para documentar esta sección es la distribuida por el propio instalador de QGIS en la siguiente ruta: C:/Program Files/QGIS 3.28.4/bin.

Para comprobar la versión de QGIS que se está utilizando:

```
1 >>> import sys
2 >>> sys.version
3 '3.9.5 (tags/v3.9.5:0a7dcbbd, May 3 2021, 17:27:52) [MSC v.1928 64 bit (AMD64)]'
```

Note

Uno de los **cambios** más importantes del paso de versión de QGIS 2.x a 3.x es el paso de la versión de Python 2.x a 3.x.

Variables

Se amplían los contenidos de esta sección en el siguiente [videotutorial](#)

Definición de variables

La sintaxis para definir una variable en Python es la siguiente:

```
<variable_name> = <expression>
```

Error

Python tiene variables locales y globales como la mayoría de los lenguajes, pero no tiene declaraciones explícitas de variables. Las variables aparecen al asignarles un valor y son automáticamente destruidas cuando salimos de su ámbito.

```
1 >>> mi_variable
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4   NameError: name 'mi_variable' is not defined
```

```
>>> mi_variable = 1
>>> mi_variable
1
```

Definición de constantes

Por el contrario, las **constantes** son un tipo particular de variable cuyo valor no se pueden alterar durante la ejecución de un programa. Se declaran usualmente en módulos a parte. Cada módulo es un archivo que se importa en el archivo principal.

Tip

Por convención las constantes son escritas en letras mayúsculas y separadas por guión bajo _, en caso que sean varias palabras. Por ejemplo:

```
PASSWORD_DB_SERVER = "123456"
```

Tipos básicos de variables

Los cuatro tipos de dato básicos o primitivos de variables en Python son el número entero (`int`), el número de coma flotante (`float`), la cadena de caracteres (`str`) y el booleano (`bool`). Sin embargo, Python incorpora un quinto tipo de dato que estrictamente hablando se llama `NoneType` y cuyo único valor posible es `None` (pronunciado llanamente «nan»).

Tipo	Descripción	Rango	Ejemplo
<code>int</code>	Numérica entera	??	2
<code>float</code>	Numérica flotante	$[-2^{308}, 2^{308}]$	3.141592
<code>str</code>	Cadena de caracteres	No aplica	«PyQGIS» o "PyQGIS"
<code>bool</code>	Booleano	True / False	True
<code>None</code>	Tipo especial		

Por tanto, las siguientes definiciones de variables no son equivalentes:

```
>>> fecha = 2021
>>> fecha = 2021.0
>>> fecha = "2021"
>>> fecha = [30, "noviembre", 2021]
```

En el primer caso la variable `fecha` está almacenando un número entero, en el segundo `fecha` está almacenando un número decimal y en el tercero `fecha` está almacenando una cadena de cuatro letras. En el cuarto, `fecha` está almacenando una lista (un tipo de variable que puede contener varios elementos ordenados). Este ejemplo demuestra también que se puede volver a definir una variable, modificando su tipo automáticamente.

Números

En Python se pueden representar números enteros, reales y complejos.

Enteros

En versiones anteriores se diferenciaba entre el tipo `int` (de *integer*, entero) y el tipo `long`. Actualmente no existe esta distinción, ya que no hay un límite para el valor de los números enteros.

Reales

En Python se expresan mediante el tipo `float`. Para representar un número real en Python se escribe primero la parte entera, seguido de un punto `.` y por último la parte decimal.

```
>>> pi = 3.141592
```

Operadores aritméticos

Las expresiones aritméticas comprenden operando y operadores:

Descripción	a	Operador	b	r	type(r)
Suma	2	+	3	5	int
	2		3.0	5.0	float
Resta	3	-	2	1	int
	3.0		2.0	1.0	float
Multiplicación	3	*	2	6	int
	3		2.0	6.0	float
División de flotantes	3	/	2	1.5	float
	-6		2	-3	int
	-6.0		2	-3.0	float
División de enteros	9	//	4	2	int
Módulo (resto de una división)	15	%	4	3	int
Exponente	2	**	8	256	int
	-4		3	-64	int
	4		-3	0.015625	float

En los siguientes bloques de código se muestra el resultado de dos operaciones y la comprobación del tipo de variable mediante el uso de la palabra reservada `type`:

```
>>> suma = 2 + 3
>>> suma, type(suma)
(5, <class 'int'>)
```

```
>>> suma = 2 + 3.0
>>> suma, type(suma)
(5.0, <class 'float'>)
```

Para operaciones más complejas se puede importar al módulo `math`.

```
>>> import math
>>> math.pi
3.141592653589793
>>> radio = 2
>>> longitud_circunferencia = 2 * math.pi * radio
>>> longitud_circunferencia
12.566370614359172
```

Funciones de conversión entre números

Existen además funciones de conversión de tipo entre números `int` y `float`:

Función	r	type(r)
>>> int(1.234)	1	int
>>> int(-1.234)	-1	int
>>> float(4)	4.0	float
>>> float('4.321')	4.321	float

Cadenas de caracteres

Las cadenas no son más que texto encerrado entre comillas simples ('cadena') o dobles ("cadena").

```
>>> geom_wkt = "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"
```

Su comportamiento es análogo al de una tupla tal y como se verá más adelante, lo que significa que no se pueden modificar sus componentes una vez inicializados. Además, se puede acceder a cada carácter por su posición por su índice:

```
1 >>> geom_wkt = "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"
2 >>> geom_wkt[0]
3 'P'
```

Tip

Antes de continuar con el desarrollo de este apartado, se aporta un breve introducción a las funciones de entrada y salida estándar en Python. Los programas serían de muy poca utilidad si no fueran capaces de interactuar con el usuario; en Python las instrucciones que permiten leer información de teclado y mostrar información por pantalla son `input` y `print`, respectivamente.

```
1 >>> edad = int(input('Introduzca su edad: ')) # entrada de entero
2 >>> peso = float(input('Introduzca su peso: ')) # entrada de flotante
3 >>> nombre = input('Introduzca su nombre: ') # entrada de cadena
4 >>> print(nombre, edad, 'años', peso, 'kg') # muestra datos
```

Caracteres especiales

Dentro de las comillas se pueden añadir caracteres especiales escapándolos con `\`:

Escape	Significado de la secuencia
<code>\b</code>	Suprimir un carácter
<code>\n</code>	Carácter de nueva línea
<code>\t</code>	Carácter tabulación
<code>'</code>	<code>'</code>
<code>"</code>	<code>"</code>

Por ejemplo:

```
>>> print("Las cadenas son texto encerrado entre comillas simples (\')")
Las cadenas son texto encerrado entre comillas simples (')
>>> print("... y también entre comillas dobles (\'")")
... y también entre comillas dobles ("")
```

```
>>> levantamiento = "Nº\tX\tY\tCOD\n1\t728762.67\t4328983.25\t\"bordillo\""
>>> print(levantamiento)
```

Nº	X	Y	COD
1	728762.67	4328983.25	"bordillo"

También es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma se podrá escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter `\n`, así como las comillas sin tener que escaparlas.

```
>>> levantamiento = """Nº\tX\tY\tCOD
1\t728762.67\t4328983.25\t"bordillo"
2\t728785.42\t4328998.43\t'acera.\b\'"""
>>> print(levantamiento)
Nº      X      Y      COD
1      728762.67  4328983.25  "bordillo"
2      728785.42  4328998.43  'acera'
```

Métodos de objeto cadenas de texto

En este apartado se facilitan la sintaxis de algunos métodos del objeto cadena de texto, aportando para cada uno de ellos una descripción y un ejemplo de aplicación.

```
S.count(sub[, start[, end]])
```

Devuelve el número de veces que se encuentra `sub` en la cadena.

```
1 >>> geom_wkt = "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"
2 >>> lados_poligono = geom_wkt.count(",")
3 >>> lados_poligono
4 4
```

Los parámetros opcionales `start` y `end` definen una subcadena en la que buscar.

```
1 >>> cadena_texto = "Esto es un cadena de 34 caracteres"
2 >>> subcadena = "ca"
3 >>> num_cadena_ca = cadena_texto.count(subcadena, 0, 33)
4 >>> num_cadena_ca
5 2
6 >>> num_cadena_ca = cadena_texto.count(subcadena, 0, 17)
7 >>> num_cadena_ca
8 1
```

```
S.find(sub[, start[, end]])
```

Devuelve la posición en la que se encontró por primera vez `sub` en la cadena o `-1` si no se encontró.

```
1 >>> cadena_texto = "Curso de programación en QGIS con Python"
2 >>> cadena_texto.find("QGIS")
3 25
4 >>> cadena_texto.find("SIG")
5 -1
```

```
S.replace(old, new[, count])
```

Devuelve una cadena en la que se han reemplazado todas las ocurrencias de la cadena `old` por la cadena `new`. Si se especifica el parámetro `count`, este indica el número máximo de ocurrencias a reemplazar.

```
1 >>> coor_pto = "547387.35, 43789234.98"
2 >>> coor_pto_separador_csv = coor_pto.replace(",", ";")
3 >>> coor_pto_separador_csv
4 '547387.35; 43789234.98'
5 >>> coor_pto_separador_punto_decimal = coor_pto_separador_csv.replace(".", ",")
6 >>> coor_pto_separador_punto_decimal
7 '547387,35; 43789234,98'
```

Las cadenas también admiten operadores como +, que funciona realizando una concatenación de las cadenas utilizadas como operandos y *, en la que se repite la cadena tantas veces como lo indique el número utilizado como segundo operando.

```
1 >>> concatena = "con" + "ca" + "te" + "nar"
2 >>> concatena
3 'concatenar'
4 >>> repite = concatena * 2
5 >>> repite
6 'concatenarconcatenar'
```

Para la comparación de dos cadenas de texto se puede utilizar el operador ==:

```
1 >>> geom_wkt_2 = "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"
2 >>> geom_wkt_3 = "LINESTRING (30 10, 10 30, 40 40)"
3 >>> geom_wkt == geom_wkt_2
4 True
5 >>> geom_wkt == geom_wkt_3
6 False
```

Booleanos

Una variable de tipo booleano sólo puede tener dos valores: True y False. Estos valores son especialmente importantes para las expresiones condicionales y los bucles. En realidad el tipo bool (el tipo de los booleanos) es una subclase del tipo int.

Operadores lógicos o condicionales

Estos son los distintos tipos de operadores lógicos o condicionales con los que se puede trabajar con valores booleanos:

Operador	Descripción	Ejemplo	r
and	¿se cumple a y b?	>>> r = True and False	False
or	¿se cumple a o b?	>>> r = True or False	True
not	No a	>>> r = not True	False

Operadores relacionales o de comparación

Como en otros lenguajes, Python también soporta operadores de comparación entre valores. Estos operadores devuelven True o False.

```
>>> x=1
>>> y=2
>>> y==x
False
```

Estos son los tipos de operadores relacionales:

Operador	Descripción	Ejemplo	r
==	¿son iguales a y b?	>>> r = 5 == 3	False
!=	¿son distintos a y b?	>>> r = 5 != 3	True
<	¿es a menor que b?	>>> r = 5 < 3	False
>	¿es a mayor que b?	>>> r = 5 > 3	True
<=	¿es a menor o igual que b?	>>> r = 5 <= 3	False

>=	¿es a mayor o igual que b?	>>> r = 5 >= 3	True
----	----------------------------	----------------	------

Eliminación de una variable

La palabra reservada `del` borra completamente una variable.

```
1 >>> geom_wkt_2 = "POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))"
2 >>> geom_wkt_2
3 'POLYGON ((30 10, 40 40, 20 40, 10 20, 30 10))'
4 >>> del(geom_wkt_2)
```

Error

Si se quiere acceder a la variable eliminada se mostrará el siguiente mensaje de error:

```
>>> geom_wkt_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'geom_wkt_2' is not defined
```

Reglas de nomenclatura de variables

A continuación se enumeran algunas reglas para nombrar variables:

Error

Las palabras reservadas **no** pueden ser utilizadas para nombrar a las variables.

```
1 >>> del = 1
2   File "<stdin>", line 1
3       del = 1
4           ^
5 SyntaxError: invalid syntax
```

Para conocer las palabras reservadas:

```
>>> help("keywords")
Here is a list of the Python keywords. Enter any keyword to get more
help.
```

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

- Los nombres de variables que empiezan por guión bajo (simple `_` o doble `__`) se reservan para variables con significado especial.
- Python es sensible a mayúsculas y minúsculas.

Error

Los espacios en blanco no están permitidos.

```
1 >>> mi variable
2   File "<stdin>", line 1
3     mi variable
4         ^
5 SyntaxError: invalid syntax
```

Tip

Finalmente, aunque no es obligatorio, se recomienda que el nombre de la variable esté relacionado con la información que se almacena en ella, para que sea más fácil entender el programa.

Asignación multiple

En una misma línea se pueden definir simultáneamente varias variables, con el mismo valor ...

```
1 >>> x_min_mancanvas = y_min_mapcanvas = 0.0
2 >>> x_min_mancanvas
3 0.0
4 >>> y_min_mancanvas
5 0.0
```

... o con valores distintos:

```
1 >>> nombre, edad = "Manuel", 35
2 >>> nombre
3 'Manuel'
4 >>> edad
5 35
```

Asignaciones aumentadas

Cuando una variable se modifica a partir de su propio valor, se puede utilizar la denominada *asignación aumentada*, una notación compacta que existe también en otros lenguajes de programación como C++, por ejemplo.

```
>>> a = 10
>>> a += 5
>>> a
15
```

es equivalente a:

```
>>> a = 10
>>> a = a + 5
>>> a
15
```

En general se tiene el siguiente cuadro de equivalencias aumentadas:

Asignación aumentada	Equivalencia
<code>a += b</code>	<code>a = a + b</code>
<code>a -= b</code>	<code>a = a - b</code>
<code>a *= b</code>	<code>a = a * b</code>
<code>a /= b</code>	<code>a = a / b</code>

<code>a **= b</code>	<code>a = a ** b</code>
<code>a /= b</code>	<code>a = a / b</code>
<code>a %= b</code>	<code>a = a % b</code>

Note

Lo que no se permite en Python son los operadores incremento (++) o decremento (--) que sí existen en otros lenguajes de programación.

Definición de comentarios

En Python utilizamos el carácter # (numeral) para indicar al intérprete que dicha línea es un comentario y no la debe procesar como una instrucción de Python. Estos pueden establecerse en una línea individual o en la misma línea de código.

Note

Se pueden utilizar triples comillas dobles (""") o triples comillas simples (''') para establecer comentarios en varias líneas, pero se recomienda no utilizar esta fórmula, ya que las triples comillas suelen utilizarse para crear la documentación de funciones (*docstring*), tal y cómo se verá más adelante. Más información en el siguiente recurso: («No uses triples comillas como comentarios»).

Estructuras o colecciones de datos

Se amplían los contenidos de esta sección en el siguiente [videotutorial](#)

En la sección anterior se definieron las variables que permiten almacenar *un único valor*. En Python existen varias estructuras de datos (*data structures*) que permiten almacenar *un conjunto de datos*.

Listas

La lista es un tipo de colección ordenada secuencialmente, equivalente a lo que en otros lenguajes se conoce por *arrays* o *vectores*. Pueden contener cualquier tipo de dato: números, cadenas, booleanos y también listas.

Crear y *definir* una lista es tan sencillo como indicar entre corchetes [], y separados por comas , , los valores que se quieren incluir en la lista:

```
>>> mi_lista = [22, True, "PyQGIS", [1, 2]]
```

El acceso a cada uno de los elementos de la lista se realiza escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes [].

Important

El índice del primer elemento de una lista es 0 y no 1.

```
1 >>> mi_lista = [22, True, "PyQGIS", [1, 2]]
2 >>> mi_variable = mi_lista[0]
3 >>> mi_variable
4 22
```

Si se quiere acceder a un elemento de una lista incluida dentro de otra lista se tendrá que utilizar dos veces este operador, primero para indicar a qué posición de la lista exterior queremos acceder, y el segundo para seleccionar el elemento de la lista interior:

```

1  >>> mi_variable = mi_lista[3][0]
2  >>> mi_variable
3  1

```

Error

Si se quiere acceder a un elemento fuera de rango, se generará el siguiente error:

```

>>> mi_variable = mi_lista[1][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bool' object is not subscriptable

```

Note

Las matrices no son una estructura propia de Python. Simplemente, una matriz es una lista de listas que nosotros interpretamos desde el punto de vista matemático. Es decir, la estructura `m = [[1,2],[3,4]]` nosotros la interpretamos como la matriz 2x2 cuya primera fila es (1,2) y cuya segunda fila es (3,4), pero esto no deja de ser una interpretación.

También se puede utilizar el operador `[]` para *modificar* un elemento de la lista si se coloca en la parte izquierda de una asignación. Por tanto, los elementos de una lista pueden variar a lo largo de su ciclo de vida (son *mutables*). Por ejemplo:

```

1 >>> mi_lista[1] = False
2 >>> mi_lista
3 [22, False, 'PyQGIS', [1, 2]]

```

Una curiosidad sobre el operador `[]` de Python es que se puede utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con `[-1]` se accedería al último elemento de la lista, con `[-2]` al penúltimo, con `[-3]`, al antepenúltimo, y así sucesivamente.

```

1 >>> mi_lista[-1]
2 [1, 2]
3 >>> mi_lista[-2]
4 'PyQGIS'

```

Otra técnica inusual es lo que en Python se conoce como *slicing* o particionado, y que consiste en ampliar este mecanismo para permitir seleccionar porciones de la lista. Si en lugar de un número se escriben dos números inicio y fin separados por dos puntos (`inicio:fin`), Python interpretará que se quiere una lista que vaya desde la posición inicio a la posición fin, sin incluir este último.

```

1 >>> mi_variable = mi_lista[0:2]
2 >>> mi_variable
3 [22, False]

```

Si se escriben tres números (`inicio:fin:salto`) en lugar de dos, el tercero se utiliza para determinar cada cuantas posiciones añadir un elemento a la lista.

```

1 >>> mi_variable = mi_lista[0:4:2]
2 >>> mi_variable
3 [22, 'PyQGIS']

```

En todo caso las listas ofrecen mecanismos más cómodos para ser modificadas a través de las métodos de la clase correspondiente tal y como se detalla a continuación.

Métodos del objeto lista

En este apartado se facilitan una serie de ejemplo con la sintaxis de distintos métodos útiles del objeto lista, aportando una descripción y un ejemplo de aplicación.

```
L.append(object)
```

Añade un objeto `object` al final de la lista

```
1 >>> mi_lista.append(3.141592)
2 >>> mi_lista
3 [22, False, 'PyQGIS', [1, 2], 3.141592]
```

```
L.pop([index])
```

Devuelve el valor en la posición `index` y lo elimina de la lista. Si no se especifica la posición, se utiliza el último elemento de la lista.

```
1 >>> mi_lista.pop()
2 3.141592
3 >>> mi_lista
4 [22, False, 'PyQGIS', [1, 2]]
```

```
L.insert(index, object)
```

Inserta el objeto `object` en la posición `index`

```
1 >>> mi_lista.insert(3, 3.141592)
2 >>> mi_lista
3 [22, True, 'PyQGIS', 3.141592, [1, 2]]
```

```
del(L[index])
```

Borra un elemento de la posición `index` de la lista.

```
1 >>> del(mi_lista[2])
2 >>> mi_lista
3 [22, False, 3.141592, [1, 2]]
```

También se puede consultar el número de elementos de una lista mediante el método `len()`:

```
1 >>> len(mi_lista)
2 4
```

Tuplas

Un tupla es simplemente una lista **inmutable**. Al igual que las cadenas de texto, una vez creadas, no se pueden modificar sus valores y tienen además un tamaño fijo, de forma que no se pueden añadir nuevos elementos ni eliminar los existentes. Lo explicado para listas es aplicable a tuplas salvo en la forma de *definición*.

```
>>> mi_tupla = (22, True, "PyQGIS", [1, 2])
```

Note

En realidad el constructor de la tupla es la coma `,`, no el paréntesis `()`, pero el intérprete muestra los paréntesis, y nosotros deberíamos utilizarlos, por claridad.

```
>>> mi_tupla = 22, True, "PyQGIS", [1, 2]
>>> type(mi_tupla)
<class 'tuple'>
```

El acceso a elementos es igual que las listas. Se puede utilizar el operador `[]` debido a que las tuplas, al igual que las listas, forman parte de un tipo de objetos llamados *secuencias*.


```
1 >>> mi_variable = mi_tupla[0]
2 >>> mi_variable
3 22
```

También soporta el particionado:

```
1 >>> mi_variable = mi_tupla[1:2]
2 >>> mi_variable
3 (True,)
```

Error

Si se intenta modificar un elemento de un tupla, se tendrá la siguiente respuesta del intérprete:

```
>>> mi_tupla[1] = False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tip

Las tuplas son más ligeras y rápidas que las listas. Se utilizarán cuando se precisa iterar sobre una estructura de datos que nunca va a ser cambiada, por ejemplo, días de la semana y meses del año.

Finalmente, se pueden convertir una lista en tupla y viceversa:


```
>>> mi_lista = [22, True, "PyQGIS", [1, 2]]
>>> mi_tupla = tuple(mi_lista)
>>> mi_tupla
(22, True, 'PyQGIS', [1, 2])
```

```
>>> mi_lista = list(mi_tupla)
>>> mi_lista
[22, True, 'PyQGIS', [1, 2]]
```

Diccionarios

Los diccionarios, también llamados *matrices asociativas* (o mapeados en otros lenguajes), deben su nombre a que son colecciones que relacionan una *clave* y un *valor* (*key – value*). El primer valor se trata de la clave y el segundo del valor asociado a la clave. Como clave se puede utilizar cualquier valor *inmutable*: se pueden usar números, cadenas, booleanos, tuplas, ... pero no listas o diccionarios, dado que son *mutables*.

Para *definir* un diccionario en Python se indicará entre llaves { }, las parejas de elementos que lo conforman. En cada pareja indicaremos primero el valor de la clave para acceder al elemento, y después el valor que contendrá separado por :. En el siguiente ejemplo se muestra un diccionario creado a partir de los CRS-EPG en los que se sirve el servicio WMTS del PNOA de máxima actualidad:

 Añadir capa(s) desde un servidor WM(T)S ? X

Capas Orden de capas Conjuntos de teselas Búsqueda de servidor

Capa	Formato	Título	Estilo	Conjunto de teselas	SRC
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:4258	EPSG:4258
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:4258	EPSG:4258
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:4326	EPSG:4326
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:4326	EPSG:4326
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:32630	EPSG:32630
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:32630	EPSG:32630
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:25830	EPSG:25830
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:25830	EPSG:25830
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	GoogleMapsCompatible	EPSG:3857
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	GoogleMapsCompatible	EPSG:3857
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:32628	EPSG:32628
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:32628	EPSG:32628
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:25828	EPSG:25828
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	EPSG:25828	EPSG:25828
OI.OrtoimageCoverage	image/jpeg	Imágenes de satélite Spot y ortofotos PNOA	default	InspireCRS84Quad	
OI.OrtoimageCoverage	image/png	Imágenes de satélite Spot y ortofotos PNOA	default	InspireCRS84Quad	

Nombre de la capa

Añadir Cerrar Ayuda

Seleccionar capa(s) o un conjunto de teselas

```
>>> mi_diccionario = {"3857": "WGS 84/Pseudo Mercator",
    "4258": "ETRS89",
    "4326": "WGS 84",
    "25828": "ETRS 89/UTM 28N",
    "25830": "ETRS 89/UTM 30N",
    "32628": "WGS 84/UTM 28N",
    "32630": "WGS 84/UTM 30N"}
```

La diferencia principal entre los diccionarios y las listas o las tuplas es que a los valores almacenados en un diccionario se les *accede* no por su índice, porque de hecho no tienen orden, sino por su clave, utilizando de nuevo el operador [].

```
>>> mi_diccionario["25830"]
'ETRS 89/UTM 30N'
```

Al igual que en listas también se puede utilizar este operador para reasignar valores.

```
1 >>> mi_diccionario["25830"] = "Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte"
2 >>> mi_diccionario["25830"]
3 'Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte'
```

También se pueden añadir nuevos pares key – values al diccionario:

```
1 >>> mi_diccionario["25831"] = "ETRS 89/UTM 31N"
2 >>> mi_diccionario
3 {'3857': 'WGS 84/Pseudo Mercator',
4  '4258': 'ETRS89',
5  '4326': 'WGS 84',
6  '25828': 'ETRS 89/UTM 28N',
7  '25830': 'Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte',
8  '32628': 'WGS 84/UTM 28N',
9  '32630': 'WGS 84/UTM 30N',
10 '25831': 'ETRS 89/UTM 31N'}
```

Sin embargo, en este caso no se puede utilizar *slicing*, entre otras cosas porque los diccionarios no son secuencias, sino *mappings* (mapeados, asociaciones).

Métodos del objeto diccionario

En este apartado se facilita la sintaxis de distintos métodos del objeto diccionario de texto, aportando una descripción y un ejemplo de aplicación.

```
D.get(k[, d])
```

Busca el valor de la clave *k* en el diccionario. Es equivalente a utilizar *D[k]* pero al utilizar este método podemos indicar un valor a devolver por defecto si no se encuentra la clave, mientras que con la sintaxis *D[k]*, de no existir la clave se lanzaría una excepción.

```
1 >>> mi_diccionario.get("25830", "Unknown SRC")
2 'Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte'
3 >>> mi_diccionario.get("25832", "Unknown SRC")
4 'Unknown SRC'
```

```
k in D
```

Comprueba si el diccionario tiene la clave *k*.

```
1 >>> "25830" in mi_diccionario
2 True
3 >>> "25832" in mi_diccionario
4 False
```

```
D.items()
```

Devuelve una lista de tuplas con pares clave-valor.

```
1 >>> mi_diccionario.items()
2 dict_items([
3  ('3857', 'WGS 84/Pseudo Mercator'),
4  ('4258', 'ETRS89'),
5  ('4326', 'WGS 84'),
6  ('25828', 'ETRS 89/UTM 28N'),
7  ('25830', 'Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte'),
8  ('32628', 'WGS 84/UTM 28N'),
9  ('32630', 'WGS 84/UTM 30N'),
10 ('25831', 'ETRS 89/UTM 31N')
11 ])
```

```
D.keys()
```

Devuelve una lista de las claves del diccionario.

```
1 >>> mi_diccionario.keys()
2 dict_keys(['3857', '4258', '4326', '25828', '25830', '32628', '32630', '25831'])
```

```
D.values()
```

Devuelve una lista de los valores del diccionario.

```
1 >>> mi_diccionario.values()
2 dict_values(['WGS 84/Pseudo Mercator', 'ETRS89', 'WGS 84', 'ETRS 89/UTM 28N',
3 'Sistema de referencia ETRS 89 - P.C. UTM huso 30 Norte',
4 'WGS 84/UTM 28N', 'WGS 84/UTM 30N', 'ETRS 89/UTM 31N'])
```

Finalmente, se pueden recorrer diccionarios con la estructura repetitiva `for .. in` que será descrita en el siguiente apartado:

```
>>> for key, value in mi_diccionario.items():
>>> for key in mi_diccionario.keys():
>>> for value in mi_diccionario.values():
```

Control de flujo

Se amplían los contenidos de esta sección en el siguiente [videotutorial](#)

Una estructura de control, es un bloque de código que permite agrupar instrucciones de manera controlada. En este capítulo tratamos dos estructuras de control:

- Estructuras de control condicionales
- Estructuras de control iterativas o repetitivas

Sentencias condicionales

En las estructuras de control condicional se decide el camino a seguir dentro del flujo del programa a partir la evaluación de una expresión.

Estructura condicional simple `if`

La forma más simple de un estamento condicional es un `if` seguido de la condición a evaluar, dos puntos (`:`) y en la siguiente línea e indentado, el código a ejecutar en caso de que se cumpla dicha condición.

```
1 if len(path_file_results) == 0:
2     self iface.messageBar().pushMessage(c.APPLICATION_NAME,
3                                         "Path filename is empty",
4                                         QgsMessageBar.CRITICAL,
5                                         10)
```

La estructura `if` contiene una condición, si dicha condición se verifica verdadera, se ejecutan todas las instrucciones que se encuentran indentadas.

Note

En Python todo aquello innecesario no hay que escribirlo (`;`, `{, }`). En otros lenguajes de programación los bloques de código se determinan encerrándolos entre llaves y el indentarlos no se trata más que de una buena práctica para que sea más sencillo seguir el flujo del programa con un solo golpe de vista. Sin embargo, en Python esta indentación es una obligación, no una elección.

Estructura condicional doble `if ... else`

Para la ejecución de un cierto número de órdenes en el caso de que no se cumpla una primera condición, se utiliza el operador `else`. Su sintaxis es la siguiente:

```
if condition :
    statements-1
else:
    statements-2
```

Un ejemplo de aplicación en el contexto de estos curso se muestra en el siguiente bloque de código con un ejemplo de creación de un objeto de la clase `QgsVectorLayer`. Utilizamos la cláusula `if` (línea 4) para evaluar la condición la validez de la capa vectorial creada (línea 5). Si ésta es válida, devolverá el objeto creado. En caso contrario (fuente de datos incorrecta u otro problema) `else` (línea 6), mostrará un mensaje integrado en QGIS informando del error y devolverá un valor `None` (línea 12).

```

1 QgsVectorLayer = QgsVectorLayer(datasource,
2                                 str_layer_name,
3                                 provider_name)
4 if QgsVectorLayer.isValid():
5     return QgsVectorLayer
6 else:
7     str_msg = "Failed to create QgsVectorLayer " + str_layer_name
8     self.iface.messageBar().pushMessage(c.CONST_APPLICATION_NAME,
9                                         str_msg,
10                                        QgsMessageBar.CRITICAL,
11                                        10)
12     return None

```

Estructura condicional múltiple `if ... elif ... elif ... else`

Cuando tenemos más de dos opciones, Python proporciona dos formas de resolver la estructura condicional. Utilizamos como ejemplo un ejercicio que permita determinar el signo de un número introducido por pantalla, de forma que tendremos tres posibles resultados: que sea positivo, negativo o que sea cero.

La primera forma de resolver el ejercicio propuesto sería la siguiente:

```

1 numero = int(input("Dame un número entero: "))
2 if numero < 0:
3     print("Negativo")
4 else:
5     if numero > 0:
6         print("Positivo")
7     else:
8         print("Cero")

```

Otra forma de resolver el ejercicio es a través de `elif` que es una contracción de *else if*. Es decir, primero se evalúa la condición del `if`. Si es cierta, se ejecuta su código y se continúa ejecutando el código posterior al condicional; si no se cumple, se evalúa la condición del `elif`. Si se cumple la condición del `elif` se ejecuta su código y se continúa ejecutando el código posterior al condicional; si no se cumple y hay más de un `elif` se continúa con el siguiente en orden de aparición. Si no se cumple la condición del `if` ni de ninguno de los `elif`, se ejecuta el código del `else`.

```

1 numero = int(input("Dame un número entero: "))
2 if numero < 0:
3     print("Negativo")
4 elif numero > 0:
5     print("Positivo")
6 else:
7     print("Cero")

```

Bucles o estructuras repetitivas o iterativas

Mientras que los *condicionales* permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los *bucles* permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición. Cada ejecución de las sentencias que se repiten se denomina *iteración*. Siempre ha de existir una condición de parada, es decir, hay que garantizar que en algún momento se darán las condiciones adecuadas para que el bucle pare. En caso contrario se produciría un bucle infinito.

Todo bucle contiene por tanto, los siguientes elementos (aunque no necesariamente en ese orden):

- Iniciación de las variables o contadores referentes al bucle.
- Decisión / condición de finalización → continuar con el bucle o terminar.

- Cuerpo del bucle: instrucciones que se repiten.

En el cuerpo del bucle, necesariamente habrá alguna (o algunas) instrucciones que modifiquen las condiciones de la expresión a evaluar para poder permitir y asegurar la salida del bucle.

Bucle indefinido: while

El bucle `while` ejecuta un fragmento de código *mientras* se cumpla una condición. Se utiliza cuando no se conoce a priori el número de iteraciones. Para ello, se comprueba en primer lugar la condición (línea 2) y después se ejecuta el cuerpo del bucle (líneas 3-4). Es necesario que las variables de la condición tengan un valor asignado antes del bucle (línea 1). Finalmente, el cuerpo del bucle debe alterar de alguna forma la condición de salida para que el bucle tenga sentido (línea 4).

```
1 contador = 0
2 while contador <= 10:
3     print(contador)
4     contador += 1
```

Bucle definido: for .. in

En Python `for` se utiliza como una forma genérica de iterar sobre una secuencia, ejecutando un bloque de código para cada elemento que tengamos en la *secuencia*.

```
secuencia = ["uno", "dos", "tres"]
for elemento in secuencia:
    print(elemento)
```

Python proporciona la función llamada `range` que permite generar una lista al vuelo que va desde el primer número que se le indique al segundo, con un determinado paso. Su sintaxis es la siguiente:

```
range(start-value, end-value, difference between the values)
```

Se propone al alumno probar la ejecución de los siguientes bucles `for`:

```
>>> for i in range(0,10): # recorre valores del 0 a 9
    print(i)
>>> for i in range(0,10,2): # recorre valores de 2 en 2 del 0 a 8
    print(i)
>>> for i in range(10,0,-1): # recorre valores de 10 al 1
    print(i)
```

Control de bucles: sentencias break, continue, pass

En este apartado estudiamos las sentencias `continue`, `break` y `pass` para manipular el comportamiento normal de los bucles.

Note

Estas sentencias se pueden utilizar tanto en bucles `for` como `while`.

continue

`continue` regresa al comienzo del bucle, ignorando todas las sentencias que quedan en la iteración actual del bucle e inicia la siguiente iteración.

```
:linenos:

for letra in "PyQGIS":
    if letra == "Q":
        continue
    print("Letra actual: ", letra)
print("Adios")
```

Produce la siguiente salida:

```
Letra actual: P
Letra actual: y
Letra actual: G
Letra actual: I
Letra actual: S
Adios
```

```
1 numero = int(input("Dame un número entero: "))
2 while numero > 0:
3     numero = numero -1
4     if numero == 5:
5         continue
6     print("Valor número actual:", numero)
7 print("Adios")
```

Produce la siguiente salida:

```
Valor variable actual: 9
Valor variable actual: 8
Valor variable actual: 7
Valor variable actual: 6
Valor variable actual: 4
Valor variable actual: 3
Valor variable actual: 2
Valor variable actual: 1
Valor variable actual: 0
Adios
```

break

break termina el bucle actual y continua con la ejecución de la siguiente instrucción.

```
1 for letra in "PyQGIS":
2     if letra == "Q":
3         break
4     print("Letra actual: ", letra)
5 print("Adios")
```

Produce la siguiente salida:

```
Letra actual: P
Letra actual: y
Adios
```

```
1 numero = int(input("Dame un número entero: "))
2 while numero > 0:
3     numero = numero -1
4     if numero == 5:
5         break
6     print("Valor número actual:", numero)
7 print("Adios")
```

Produce la siguiente salida:

```
Valor variable actual: 9
Valor variable actual: 8
Valor variable actual: 7
Valor variable actual: 6
Adios
```

pass

pass tal como su nombre lo indica es una operación nula, o sea que no pasa nada cuando se ejecuta.

```

1 for letra in "PyQGIS":
2     if letra == "Q":
3         pass
4     print("Letra actual: ", letra)
5 print("Adios")

```

Produce la siguiente salida:

```

Letra actual: P
Letra actual: y
Letra actual: Q
Letra actual: G
Letra actual: I
Letra actual: S
Adios

```

Tip

`pass` se puede utilizar cuando una sentencia es requerida por la sintaxis pero el programa no requiere ninguna acción. Se usa normalmente para crear clases, funciones y bucles en su mínima expresión, por ejemplo:

```

>>> class MyEmptyClass:
        pass # TODO: acuerdate de implementar esto

```

Funciones definidas por el usuario

Se amplían los contenidos de esta sección en el siguiente [videotutorial](#)

Una **función** es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor.

Tip

Una buena práctica en programación consiste en diseñar la finalidad de una función para realizar una única acción, reutilizable y por tanto, tan genérica como sea posible.

Definición de funciones con un solo argumento

En Python las funciones se declaran con la palabra clave `def` que es la abreviatura de *define* seguida del nombre de la función y entre paréntesis () y los argumentos separados por comas ,. A continuación, en otra línea, indentado y después de los dos puntos : se tendrían las líneas de código que conforman el código a ejecutar por la función.

Note

Nos referiremos a *argumentos* y *parámetros* indistintamente en este tutorial. Por otro lado, en Python a las funciones de las clases se les denomina métodos.

```

1 def func(pass_argument):
2     """
3     Example function definition with a simple argument
4     :param pass_argument: string to print
5     :type pass_argument: str
6     :rtype: None

```



```

7     """
8     print(pass_argument)

```

En el código anterior se puede introducir una cadena de texto (línea 2-6) como primeras líneas del cuerpo de la función denominada *docstring* (cadena de documentación), que es lo que imprime la función `help` de Python:

```

>>> help(func)
Help on function func in module __main__:
func(pass_argument)
    Example function definition with a simple argument
    :param pass_argument: string to print
    :type pass_argument: str

```

Denominamos *activar*, *invocar* o *llamar* a un función a la acción de usarla, ya que al declararla lo único que hacemos es asociar un nombre al fragmento de código que conforma la función. Para llamar (invocar) a la función (ejecutar su código) se escribiría el nombre de la función a la que se quiere llamar seguido de los valores que se quieran pasar como parámetros entre paréntesis.

```

>>> str_argument = "Pass argument string"
>>> func(str_argument) # calling a function
Pass argument string

```

Definición y uso de funciones con varios argumentos

No todas las funciones tienen un solo argumento. Se define ahora una función que imprime dos valores pasados como parámetros.

```

1 def mi_funcion(param1, param2):
2     """Esta funcion imprime los dos valores pasados como parametros"""
3     print(param1)
4     print(param2)

```

```

>>> mi_funcion("parámetro 1", "parámetro 2")
parámetro 1
parámetro 2

```

Error

El número de valores que se pasan como parámetro al llamar a la función tiene que coincidir con el número de parámetros que la función acepta según la declaración de la función. En caso contrario Python emitirá la correspondiente excepción.

```

>>> mi_funcion("parámetro 1", "parámetro 2", "otro parámetro")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: mi_funcion() takes 2 positional arguments but 3 were given

```

Al llamar una función, se le deben pasar sus argumentos en el mismo orden en el que los espera. Pero esto puede evitarse, haciendo uso del paso de argumentos como *keywords*.

Definición y uso de funciones sin argumentos

Lo único que se debe tener presente en este caso es que es obligatorio utilizar paréntesis a continuación del identificador, tanto para al definir la función ...

```

1 def hello_world():
2     """
3     Example function definition without arguments
4     :rtype: None

```

```

5     """
6     print("You are in Hellow World")

```

... como al invocarla.

```

>>> my_var = hello_world() # calling a function
You are in Hellow World

```

Note

A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar **procedimientos**. En Python no existen los procedimientos como tales, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor `None` (nada). ¿Y para qué sirve una función que no devuelve nada? Bueno, puede, por ejemplo, mostrar mensajes o resultados por pantalla.

```

>>> print("Return value function: " + str(type(my_var)))
Return value function: <class 'NoneType'>

```

Definición de funciones con devolución de valores

En el siguiente ejemplo se incorpora en el código del cuerpo de la función la palabra reservada `return`, para devolver el valor de la función.

```

1 def sum(a, b):
2     """
3     sum of two numbers
4     :param a: first operand
5     :type a: int or float
6     :param b: second operand
7     :type b: int or float
8     :return: suma
9     :rtype: int or float
10    """
11    c = a + b
12    return c

```

A continuación se invoca la función en línea imprimiendo el resultado.

```

1 >>> x = 10
2 >>> y = 50
3 >>> print("Result of addition " + str(x) + " + " + str(y) + " = " + str(sum(x, y)))
4 Result of addition 10 + 50 = 60

```

También se podrían pasar varios valores que retornar a `return`.

Note

Sin embargo esto no quiere decir que las funciones Python puedan devolver varios valores, lo que ocurre en realidad es que Python crea una tupla al vuelo cuyos elementos son los valores a retornar, y esta única variable es la que se devuelve:

```

1 def f(x, y):
2     return x * 2, y * 2

```

El resultado de la ejecución de este programa sería:

```

>>> a, b = f(1, 2)
>>> print(a, b)
2 4

```

Funciones con argumentos por defecto

Los valores opcionales o por defecto o por omisión de los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto:

```
>>> def imprimir(texto, veces = 1):
    print(veces * texto)
```

La llamada a la función imprimir sin especificación del parámetro muestra por pantalla el siguiente resultado:

```
>>> imprimir("PyQGIS")
PyQGIS
```

Especificando 3 veces el número de repeticiones obtenemos el siguiente resultado:

```
>>> imprimir("PyQGIS", 3)
PyQGISPyQGISPyQGIS
```

Funciones con argumentos de longitud variable

Finalmente, también es posible definir una función con un número variable de parámetros.

```
1 def print_variable_argument(convencional_argument,
2                             *variable_argument):
3     """
4     Example function with variable lenght argument
5     :param convencional_argument: conventional argument
6     :param variable_argument: variable argument
7     """
8     str_msg = "Convencional argument: "
9     str_msg += str(convencional_argument)
10    str_msg += ", Variable arguments: "
11    for var in variable_argument :
12        str_msg += str(var) + ", "
13    print(str_msg)
```

A continuación se muestran los resultados de la llamada a la función definida sin y con argumentos de longitud variable.

```
>>> print_variable_argument(60)
Convencional argument: 60, Variable arguments:
>>> print_variable_argument(100,90,40,60)
Convencional argument: 100, Variable arguments: 90, 40, 60,
```

Paso de variables por referencia o por valor

En el cuerpo de las funciones es posible definir y usar variables. Vamos a estudiar con detenimiento algunas propiedades de las variables definidas en el cuerpo de una función y en qué se diferencian de las variables que definimos fuera de cualquier función, es decir, en el denominado programa principal.

En el paso *por referencia* lo que se pasa como argumento es una referencia o puntero a la variable, es decir, la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido en si. En el paso *por valor*, por el contrario, lo que se pasa como argumento es el valor que contenía la variable.

Si se quiere modificar el valor de uno de los argumentos y que estos cambios se reflejen fuera del ámbito de la función, se tendría que pasar el parámetro por referencia.

En el caso de Python los valores mutables se comportan como paso por referencia, y los inmutables como paso por valor:

```
1 def f(x, y):
2     x = x + 3
3     y.append(23)
4     print("Valor var x dentro de func:", x, "Valor var y dentro de func:", y)
```

```
>>> x = 22
>>> y = [22]
>>> f(x, y)
Valor var x dentro de func: 25 Valor var y dentro de func: [22, 23]
>>> print("Valor variable x fuera de la función:", x, "Valor variable y fuera de la función:")
Valor var x dentro de func: 22 Valor var y dentro de func: [22, 23]
```

Vemos a continuación más ejemplos de argumentos por referencia vs. pasos por valor

```
1 def fun(a):
2     a = a + 9000
3     print("var value inside the function: ", a)
```

```
>>> a = 1
>>> fun(a)
var value inside the function: 9001
>>> print("var value outside the function: ", a)
var value outside the function: 1
```

```
1 def pass_ref(list1):
2     """
3     Example 2 pass by reference versus pass by value
4     :param a: lista a imprimir
5     :type a: list
6     """
7     list1.extend([23,89])
8     print("List inside the function: ", list1)
```

```
>>> list1 = [12, 67, 90]
>>> print("List before pass: ", list1)
List before pass: [12, 67, 90]
>>> pass_ref(list1)
List inside the function: [12, 67, 90, 23, 89]
>>> print("List outside the function: ", list1)
List outside the function: [12, 67, 90, 23, 89]
```

Ámbito de las variables: variables locales y globales

```
1 def func():
2     """
3     Example 1 scope of variables
4     :param a: number to print
5     :type a: int
6     """
7     a = 1
8     print("Inside the function the value of a is acting as local variable", a)
```

```
>>> a = 9000
>>> func()
Inside the function the value of a is acting as local variable 1
>>> print("Outside the function the value of a is acting as global variable", a)
Outside the function the value of a is acting as global variable 9000
```

La sentencia `global` se utiliza para declarar variables globales.

```
1 def func():
2     """
3     Example 2 scope of variables
4     :param a: number to print
5     :type a: int
```

```

6     """
7     global a
8     a = a + 7
9     print("2.- Variable a is now global", a)

```

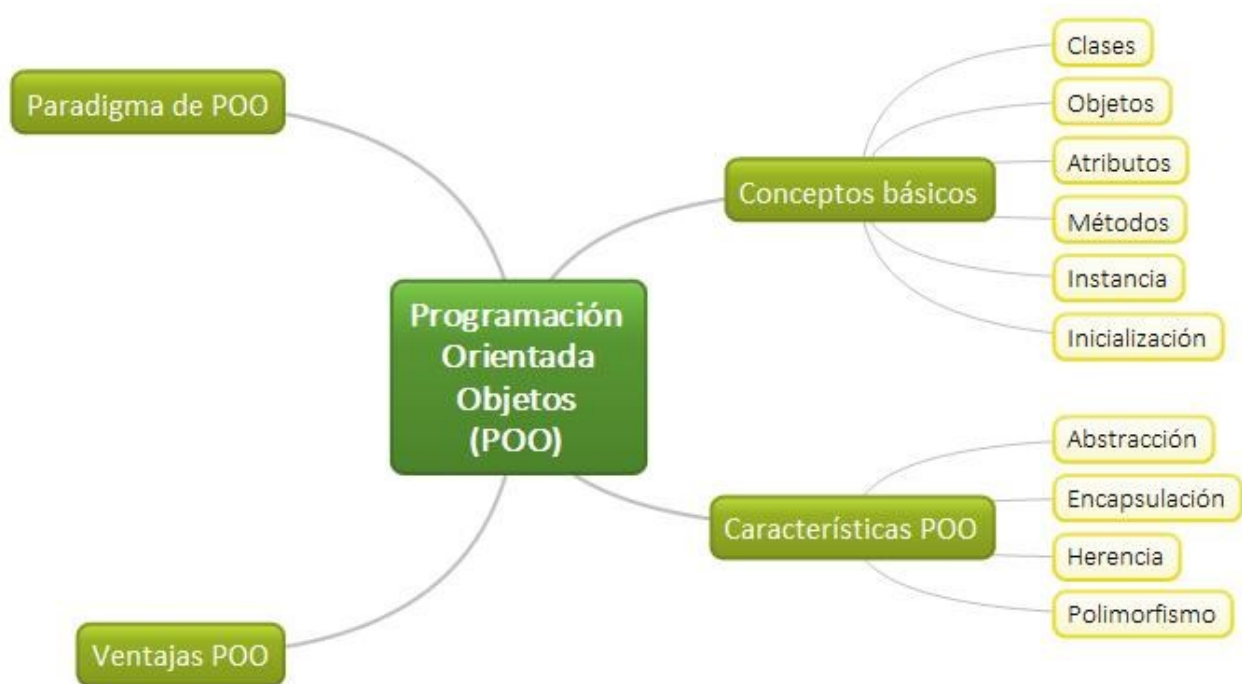
```

>>> a = 9000
>>> print("1.- Variable a before pass: ", a)
1.- Variable a before pass: 9000
>>> func()
2.- Variable a is now global 9007
>>> print("3.- Accesing the value a outside the function", a)
3.- Accesing the value a outside the function 9007

```

Programación orientada a objetos en Python

Python es un lenguaje popular de *scripting*, pero también soporta el paradigma de Programación Orientada a Objetos (POO). En esta capítulo se facilita una descripción teórica de conceptos básicos y características principales de la POO y de los detalles de su implementación en el lenguaje de programación Python. Estos conocimientos serán aplicados posteriormente en la descripción de APIs de QGIS.



Paradigma de POO

Un paradigma de programación representa un enfoque particular o filosofía para diseñar soluciones de software y Python implementa varios paradigmas: imperativo, procedural, funcional y POO. En el paradigma de la POO los conceptos del mundo real relevantes para resolver un determinado problema se modelan a través de clases y objetos y sus interacciones, trasladando el mundo real al mundo informático.

La idea fundamental de la orientación a objetos y de los lenguajes que implementan este paradigma de programación, es combinar (encapsular) en una sola unidad tanto los datos como las funciones que operan (manipulan) sobre los datos. Esta característica permite modelar los objetos del mundo real de un modo mucho más eficiente que con funciones y datos. Esta unidad de programación se denomina **objeto**.

Conceptos básicos de la POO

A continuación se definen las ideas fundamentales más básicas que todo aquel que trabaja en POO debe comprender y manejar constantemente.

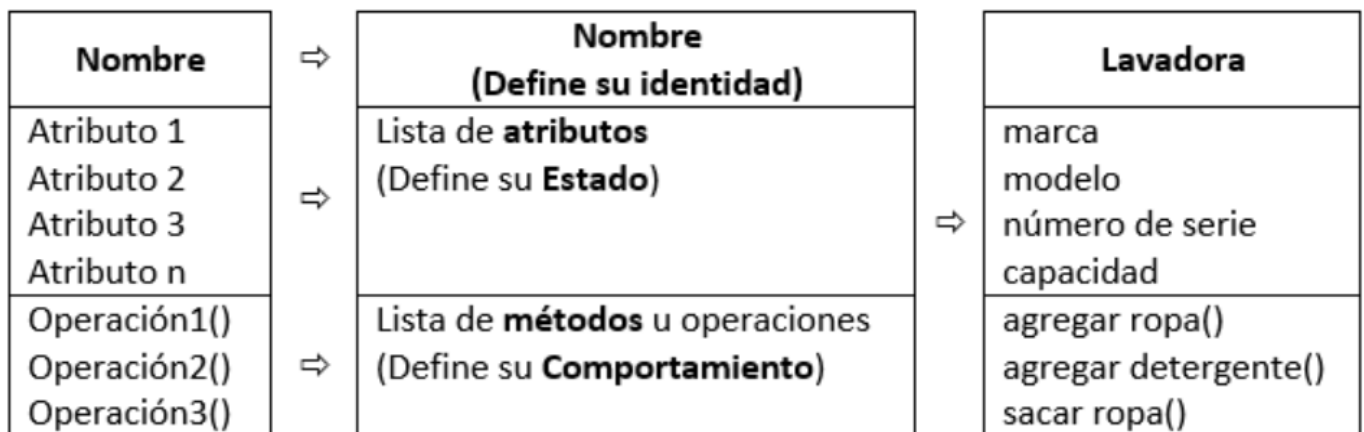
Una **clase** es simplemente una abstracción que hacemos de nuestra experiencia sensible. El ser humano tiende a agrupar seres o cosas con características similares (*objetos*) en grupos o categorías (*clases*). Las clases son abstracciones que agrupan entidades con un *estado* y unas *funcionalidades* similares. El nivel de detalle con el que se definen depende de la capacidad de abstracción de quien analiza el objeto.

- El **estado** se define a través de variables llamadas **atributos** (o características).
- La **funcionalidad** se modela a través de funciones a las que se les conoce con el nombre de **métodos** (o acciones) del objeto, que representan su **comportamiento**. En su implementación, los métodos son segmentos de código en forma de funciones.

Important

Una **clase** se puede considerar como un tipo de dato definido por el usuario que permite definir y representar colecciones de objetos y proveen un modelo o *plantilla* genérica para su creación. Dicho de otra forma, una clase es un molde que se utiliza para crear objetos de un tipo específico.

Ejemplo de una clase:



Note

En notación del Lenguaje Unificado de Modelado (UML, *Unified Modeling Language*) una clase se representa en un rectángulo con tres compartimentos: el nombre de la clase, sus atributos y sus métodos.

- Notación de atributo: `nombreAtributo:tipoDato = valorPorDefecto`
- Notación de método: `nombreMetodo(listaParametros::tipoDato):tipoDatoDevuelto`

Al contrario de lo que sucede en la programación estructurada, donde variables y funciones están separadas, en la POO los objetos integran datos y algoritmos.

Important

Un **objeto** es un ejemplar de una clase (instancia), una entidad con valores específicos de atributos y operaciones:

`miLavadora:Lavadora`

```

marca = «LG»
modelo = «F4J6TY0W»
numero de serie = «S321158»
capacidad = 8

```

La duración de un objeto en un programa siempre está limitada en el tiempo. Cada objeto es responsable de inicializarse y destruirse de forma correcta: los objetos son creados mediante un mecanismo denominado **instanciación** y dejan de existir cuando son destruidos.

Implementación de clases en Python

En Python las clases se definen mediante la palabra reservada `class` seguida del nombre de la clase, dos puntos (`:`) y a continuación, indentado, el cuerpo de la clase. Como en el caso de las funciones, si la primera línea del cuerpo es una cadena de texto, esta será la cadena de documentación de la clase o *docstring*.

```

class <class name>(<parent class name>):
    <method definition-1>
    <method definition-n>

```

Los **atributos** que se aplican a toda la clase son definidos al principio y se denominan atributos de clase (línea 3 del siguiente ejemplo). Todos los **métodos** incluidos en la definición de la clase pasan el objeto en cuestión como primer parámetro (línea 4). La palabra `self` es utilizada para este parámetro.

Note

El uso de `self` es actualmente por convención, no se trata de una palabra reservada de Python, pero es una de las convenciones más respetadas.

Ejemplo de implementación de una clase:

```

1 class MyClass:
2     """ Brief: a simple example class """ # docstring
3     i = 12345                             # class attribute
4     def method_1(self):                   # regular method
5         return 'Hola Mundo'

```

Cada objeto creado a partir de una clase se denomina *instancia* de la clase. Para **instanciar** una clase en Python:

```

x = MyClass() # creacion de un objeto de la clase MyClass

```

Una vez creado el objeto, se puede acceder a sus atributos y métodos mediante la sintaxis `objeto.atributo` (línea 1) y `objeto.metodo()` (línea 3):

```

1 >>> x.i # acceso a atributos
2 12345
3 >>> x.method_1() # acceso a metodos
4 Hola Mundo

```

El método `__init__`, con una doble barra baja `__` al principio y final del nombre, se ejecuta justo después crear un objeto (instancia de una clase). El método `__init__` sirve, como sugiere su nombre, para realizar cualquier proceso de **inicialización** que sea necesario. Es el equivalente al *constructor* de otros lenguajes orientados a objetos como C++.

Ejemplo de clase con el método constructor `__init__`:

```

class NumeroComplejo:
    """
    Resumen: Números complejos
    """
    def __init__(self,
                 parte_real,
                 parte_imaginaria):

```

```

"""
Resumen: Método constructor de la clase
:param parte_real: parte real del número
:type parte_real: real number
:param parte_imaginaria: parte imaginaria del número
:type parte_imaginaria: real number
"""
print("¡Clase inicializada!")
self.r = parte_real
self.i = parte_imaginaria

```

El primer parámetro de `__init__` y del resto de métodos de la clase es siempre `self` que sirve para referirse al objeto actual.

En este caso, para crear un objeto se escribiría el nombre de la clase seguido de cualquier parámetro que sea necesario entre paréntesis. Estos parámetros son los que se pasarán al método `__init__`

```

>>> x = NumeroComplejo(3.0, -4.5)
¡Clase inicializada!

```

Se puede acceder a la parte real e imaginaria como en el caso anterior:

```

>>> print(x.r, x.i)
3.0 -4.5

```

Note

Python pasa el primer argumento `self` (la referencia al objeto que se crea) automáticamente.

Características de la POO

Hay cuatro conceptos que son básicos en el paradigma de POO que lo llevan a ser un estilo de desarrollo que permite crear código reutilizable: la abstracción, el encapsulamiento, la herencia y el polimorfismo.

Abstracción

Anteriormente definimos que la clase es una representación abstracta de un objeto, pero ¿qué es *abstracción*?

Important

La **abstracción** es el proceso mental de extracción de las características esenciales de algo, ignorando los detalles superfluos. Es un mecanismo, quizá innato, por el que tendemos a hacer simple aquello que por su naturaleza es complejo.

Es decir, cuando vemos un objeto, solo nos fijamos en aquellas propiedades y comportamiento que nos son útiles para el fin que perseguimos, eliminando aquellos otros que, de momento, son irrelevantes o nos distraen del problema.

Desde otro enfoque, la abstracción surge del reconocimiento de las similitudes entre ciertos objetos, situaciones o procesos del mundo real, y en la decisión de concentrarse en esas similitudes e ignorar por el momento las diferencias.

Por ejemplo, en una persona podrían ser características esenciales el DNI, el nombre, la edad, la talla o el peso. Sin embargo, la marca de sus calcetines es una característica superflua.

Encapsulación

Important

La **encapsulación** es el proceso mediante el cual se ocultan las estructuras de datos y los detalles de implementación, permitiendo considerar a los objetos como «*cajas negras*», evitando que otros objetos accedan a detalles que no les interesan. Esta cualidad hace que la POO sea muy apta para la reutilización de programas.

Un ejemplo de encapsulación sería una medicina prescrita por el médico en forma de cápsula. Sabemos sus beneficios pero no como funciona por dentro.

De modo predeterminado, el conjunto de atributos y métodos se encuentran encapsulados o contenidos dentro de una misma clase, de manera que son **miembros** de dicha clase. Esos métodos y atributos pueden ser utilizados por otras clases sólo si la clase que los encapsula les brinda los permisos necesarios para ello. De esta forma, se impide el acceso a determinados métodos y atributos de los objetos estableciendo así qué puede utilizarse desde fuera de la clase.

Esto se consigue en otros lenguajes de programación como Java utilizando modificadores de acceso que definen si cualquiera puede acceder a esa función o variable (`public`) o si está restringido el acceso a la propia clase (`private`).

En Python no existen los modificadores de acceso, y lo que se suele hacer es que el acceso a una variable o función viene determinado por su nombre: si el nombre comienza con dos guiones bajos (`__`) (y no termina también con dos guiones bajos) se trata de una variable o función *privada*, en caso contrario es *pública*.

Note

Los métodos cuyo nombre comienza y termina con dos guiones bajos `__` son métodos especiales que Python llama automáticamente bajo ciertas circunstancias.

En el siguiente ejemplo, definimos un método público (línea 2) y un método privado (línea 4) ...

```
1 class Encapsula:
2     def public_method(self):
3         print("Has accedido a un método público")
4     def __private_method(self):
5         print("Has accedido a un método privado")
```

... tras instanciar la clase Encapsula y crear un objeto ...

```
>>> encapsula_object = Encapsula()
```

... sólo se imprimiría la cadena correspondiente al método `publico()`, ...

```
>>> encapsula_object.public_method()
Has accedido a un método público
```

Error

... mientras que al intentar llamar al método `__privado()` Python lanzará una excepción:

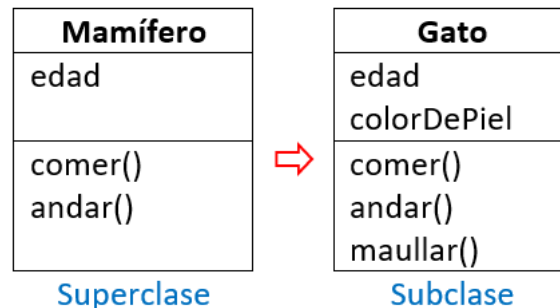
```
>>> encapsula_object.__private_method()
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: 'Encapsula' object has no attribute '__private_method'
```

La *abstracción* y el *encapsulamiento* son conceptos complementarios: la primera se centra en el comportamiento observable de un objeto, mientras el encapsulamiento se centra en la implementación que da lugar a este comportamiento.

Herencia

En un lenguaje orientado a objetos cuando una clase (**subclase** o **clase hija**) **hereda** de otra clase existente (**superclase** o **clase padre**) se consigue que la subclase contenga todos los atributos y métodos que tenía la superclase. A este procedimiento también se le denomina “*extender una clase*”.

De esta forma, las clases hijas heredan las características de sus clases antecesoras; los atributos y métodos declarados en la clase padre son accesibles en la clase hija, como si se hubieran declarado localmente, permitiendo reutilizar código creando nuevas clases a partir de las existentes, previamente construidas y depuradas.



En Python, para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis () después del nombre de la clase.

Vamos a implementar el ejemplo de la figura en lenguaje Python. En primer lugar definimos la superclase **Mamifero**

```
...
class Mamifero:
    def comer(self):
        print("Puedo comer")
    def andar(self):
        print('Puedo andar')
```

... y a continuación, la subclase **Gato** que hereda de la clase **Mamifero**, extendiendo su funcionalidad a través de un nuevo método (**maullar**):

```
class Gato(Mamifero):
    def maullar(self):
        print('Miau')
```

El paso siguiente es crear un nuevo objeto de la clase **Gato**:

```
>>> objeto_gato = Gato()
```

Y finalmente, se accederemos a los métodos de la superclase (**comer** y **andar**) y al método de la subclase (**maullar**) desde la instancia del objeto de la clase **Gato**. La ejecución de este código producirá la impresión en pantalla de los siguientes mensajes:

```
>>> objeto_gato.comer()
Puedo comer
>>> objeto_gato.andar()
Puedo andar
>>> objeto_gato.maullar()
Miau
```

Herencia múltiple

En Python, a diferencia de otros lenguajes como Java o C#, se permite la **herencia múltiple**, es decir, una clase puede heredar de varias clases a la vez. Basta con enumerar las clases de las que se hereda separándolas por comas. Se muestra a continuación un sencillo ejemplo de uso de la herencia múltiple:

En primer lugar se definen las clases **A** (línea 1) y **B** (línea 5) y a continuación, la clase **C** (línea 9) que hereda de la clase **A** y **B**.

```
1 class A():
2     def suma(self, a, b):
3         c = a+b
4         return c
```

```

5 class B():
6     def resta(self, a, b):
7         c = a-b
8         return c
9 class C(A, B):
10    pass

```

A continuación, se crea un objeto de la clase **C**:

```
>>> c_obj = C()
```

Finalmente, se accede al método de suma de la clase **A** y sustracción de la clase **B** desde el objeto de la clase **C**, produciendo los siguientes resultados:

```

>>> print("Suma es ", c_obj.suma(12,4))
Suma es 16
>>> print("La resta es ", c_obj.resta(45,5))
La resta es 40

```

Herencia multinivel

También está permitida la **herencia multinivel**. Para ello se define en primer lugar la clase **A** (línea 1). A continuación, se define la clase **B** (línea 5) que hereda de la clase **A** y finalmente, se define la clase **C** (línea 7) que hereda de la clase **B**.

```

1 class A():
2     def sum1(self, a, b):
3         c = a+b
4         return c
5 class B(A):
6     pass
7 class C(B):
8     pass

```

A continuación, se instancia la clase **C**, creando un objeto de esta clase:

```
>>> c_obj = C()
```

Finalmente, se accede al método `sum1` definido en la clase **A** (línea 2), desde el objeto de la clase **C**:

```

>>> print("Suma es ", c_obj.sum1(12,4))
Suma es 16

```

Polimorfismo

Important

Por **polimorfismo** se entiende aquella cualidad que poseen los objetos para responder de distinto modo ante el mismo mensaje. Esto significa que dos clases que tengan un método con el mismo nombre y que respondan al mismo tipo de mensaje (es decir, que reciban los mismos parámetros), ejecutarán acciones distintas. Por otro lado, **sobrecarga** hace referencia a un conjunto de métodos definidos en una misma clase con el mismo nombre pero con diferente número de parámetros y/o tipos de estos.

Para ilustrar la implementación de esta característica en Python se crean en primer lugar dos clases `Perro` y `Pajaro`, con un método con el mismo nombre `avanzar` que imprimen en pantalla dos acciones distintas:

```

1 class Perro():
2     def avanzar(self):
3         print('Corriendo')
4 class Pajaro():
5     def avanzar(self):
6         print('Volando')

```

A continuación se crea una nueva función que recibe como parámetro un objeto de ambas clases y realiza la llamada al método `avanzar` definido anteriormente en ambas clases:

```
1 def mover(animal):  
2     animal.avanzar()
```

El siguiente paso consistirá en crear dos nuevos objetos de cada una de las clases:

```
>>> objeto_perro = Perro()  
>>> objeto_pajaro = Pajaro()
```

Y, finalmente se realiza la llamada a la función `mover` pasando como argumento `objeto_perro` y `objeto_pajaro`, respectivamente. La respuesta al mismo tipo de mensaje ejecuta dos acciones distintas:

```
>>> mover(objeto_perro)  
Corriendo  
>>> mover(objeto_pajaro)  
Volando
```

Ventajas de la POO

Se enumeran a continuación las principales ventajas de este paradigma de programación:

- **Reutilidad.** Si diseñamos adecuadamente las clases, podrán ser utilizadas en distintas partes del programa y en numerosos proyectos.
- Reducción de código redundante, lo que permite un código conciso y sin repeticiones (mecanismo de Herencia). La codificación utilizando clases es fácil de *leer, entender, extender y mantener*.
- **Fiabilidad.** Al dividir el problema en partes más pequeñas podemos probarlas de manera independiente y aislar mucho más fácilmente los posibles errores que puedan surgir. Gracias a la modularidad, cada componente o módulo de un desarrollo tiene independencia de los demás componentes.
- Se ocultan los detalles de implementación al usuario de la clase.
- Correspondencia directa con el mundo real debido a la filosofía del paradigma.

Excepciones

Las excepciones son errores detectados por Python durante la ejecución del programa y no son necesariamente errores fatales. Muchas excepciones no son manejadas por los programas; es posible escribir programas que manejen excepciones seleccionadas. Si la excepción no se captura el flujo de ejecución del programa se interrumpe y se muestra la información asociada a la excepción en la consola.

En Python se utiliza una construcción `try-except` para capturar y tratar las excepciones. El bloque `try` (intentar) define el fragmento de código en el que se sospecha que podría producirse una excepción. El bloque `except` (excepción) permite indicar el tratamiento que se llevará a cabo de producirse dicha excepción.

Codifica en un bloque `else` solamente lo que quieras ejecutar si no hubiesen excepciones ocasionadas por el código del bloque `try`. Esto es útil si tienes algún código que no quieres ejecutar si una excepción es encontrada y no quieres analizar si hay excepciones en ese código.

Algunas veces, se buscará que ocurra algo independientemente de lo que suceda con la excepción. En aplicaciones reales es útil para liberar recursos externo como archivos o conexiones a bases de datos. El bloque `finally` de una cláusula `try` se ejecutará siempre, se produzca o no la excepción.

A continuación se muestra un ejemplo de uso de `else` y `finally` en el contexto del manejo de excepciones:

```
1 def divide(x, y):  
2     try:  
3         result = x / y  
4     except ZeroDivisionError:  
5         print("Denominador cero!!!")  
6     else:  
7         print("El resultado es", result)  
8     finally:  
9         print("Ejecución cláusula finally")
```

Si la llamada a la función se realiza con los siguientes argumentos `divide(2,1)`, se obtendrá el siguiente resultado:

```
El resultado es 2.0
Ejecución cláusula finally
```

Error

Si se introduce como denominador el valor cero, `divide(2,0)`, obtendremos el siguiente resultado:

```
Denominador cero!!!
Ejecución cláusula finally
```

Finalmente, si introducimos una excepción no controlada como es el caso de la división de dos cadenas de texto, `divide("2","1")`, obtendremos el siguiente error durante la ejecución:

```
Ejecución cláusula finally
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

En el siguiente ejemplo se ha sustituido el bloque de código de la excepción para mostrar el error.

```
1 def divide(x, y):
2     try:
3         result = x / y
4     except Exception as detail:
5         print(detail, type(detail))
```

En este caso la llamada a la función con argumento denominador cero genera el siguiente resultado:

```
>>> divide(2,0)
division by zero <class 'ZeroDivisionError'>
```

Para el caso de la llamada a la función con dos cadenas de texto, producirá en este caso los siguientes resultados y no detendrá la ejecución del programa:

```
>>> divide("2","1")
unsupported operand type(s) for /: 'str' and 'str' <class 'TypeError'>
```

Tip

Las excepciones son también objetos regulares de Python que heredan de la clase `BaseException`. La jerarquía de excepciones puede consultarse en el [siguiente enlace](#).: El tipo de error producido, `TypeError`, hereda de la clase `BaseException` `Exception` `StandardError`

También se pueden implementar tipos de error personalizados. En el siguiente ejemplo se muestra un bloque de código en el que se intenta conectar con una base de datos SQLite, mostrando un error si la conexión no se lleva a cabo:

```
1 try:
2     connection_object = sqlite.connect(path)
3     str_msg = "It has established a connection with the database " + path
4     self.iface.messageBar().pushMessage(c.CONST_APPLICATION_NAME,
5                                         str_msg,
6                                         level = QgsMessageBar.INFO)
7     return connection_object
8 except sqlite.OperationalError, Msg:
9     str_msg_error_db = str(Msg)
10    str_msg = "Can't connect to Database: " + path + ". Error:" + str_msg_error_db
11    self.iface.messageBar().pushMessage(c.CONST_APPLICATION_NAME,
12                                         str_msg,
```

```

13                                     QgsMessageBar.CRITICAL,
14                                     10)
15     return

```

Módulos y paquetes

Módulos

Para facilitar el mantenimiento y la lectura los programas demasiado largos pueden dividirse en módulos, agrupando elementos relacionados. Los módulos son entidades que permiten una organización y división lógica de nuestro código. Los ficheros son su contrapartida física: cada archivo Python almacenado en disco equivale a un **módulo**.

Para importar un módulo se utiliza la palabra clave `import` seguida del nombre del módulo, que consiste en el nombre del archivo menos la extensión.

El `import` no solo hace que se tenga disponible todo lo definido dentro del módulo, sino que también ejecuta el código del módulo.

Es necesario preceder el nombre de los objetos que se importan de un módulo con el nombre del módulo al que pertenecen. Sin embargo es posible utilizar la construcción `from-import` para ahorrarnos el tener que indicar el nombre del módulo antes del objeto que nos interesa. De esta forma se importa el objeto o los objetos que indiquemos al espacio de nombres actual.

Tip

Aunque se considera una *mala práctica*, también es posible importar todos los nombres del módulo al espacio de nombres actual usando el caracter `*`:

```
>>> from time import *
```

A la hora de importar un módulo Python recorre todos los directorios indicados en la variable de entorno `PYTHONPATH` en busca de un archivo con el nombre adecuado. El valor de la variable `PYTHONPATH` se puede consultar desde Python mediante `sys.path`

```

>>> import sys
>>> sys.path

```

Paquetes

Mientras los módulos se corresponden a nivel físico con los archivos, los paquetes se representan mediante directorios. Para hacer que Python trate a un directorio como un paquete es necesario crear un archivo `__init__.py` en dicha carpeta.

Escritura y lectura de ficheros de texto

Formateo de la salida

Antes de describir las sentencias para el manejo de ficheros de texto, se aportan unas definiciones previas sobre el formateo de salida.

La sentencia `print`, o más bien las cadenas que imprime, permiten también utilizar técnicas avanzadas de formateo, de forma similar al `sprintf` de C. Veamos varios ejemplos bastante simples.

Lo que hace la primera línea es introducir los valores a la derecha del símbolo `%` (la cadena «mundo») en las posiciones indicadas por los especificadores de conversión de la cadena a la izquierda del símbolo `%`, tras convertirlos al tipo adecuado.

```

>>> print("Hola %s" % "mundo")
Hola mundo

```

En la segunda línea, vemos cómo se puede pasar más de un valor a sustituir, por medio de una tupla. En este ejemplo sólo tenemos un especificador de conversión: %s.

```
>>> print("%s %s" % ("Hola", "mundo"))
Hola mundo
```

```
>>> print("%d %f %s" % (2, 3.14, "Hi"))
```

```
>>> print("La distancia total recorrida es de %f metros" % 9.10)
```

Los especificadores más sencillos están formados por el símbolo % seguido de una letra que indica el tipo con el que formatear el valor:

Especificador	Formato
%s	Cadena
%d	Entero
%f	Real

Se puede introducir un número entre el % y el carácter que indica el tipo al que formatear, indicando el número mínimo de caracteres que queremos que ocupe la cadena. Si el tamaño de la cadena resultante es menor que este número, se añadirán espacios a la izquierda de la cadena. En el caso de que el número sea negativo, ocurrirá exactamente lo mismo, sólo que los espacios se añadirán a la derecha de la cadena.

```
>>> print("%10s mundo" % "Hola")
_____Hola mundo

>>> print("%-10s mundo" % "Hola")
Hola_____mundo
```

En el caso de los reales es posible indicar la precisión a utilizar precediendo la f de un punto seguido del número de decimales que queremos mostrar:

```
>>> from math import pi
>>> print("%.4f" % pi)
3.1416
```

La misma sintaxis se puede utilizar para indicar el número de caracteres de la cadena que queremos mostrar

```
>>> print("%.4s" % "hola mundo")
hola
```

Manejo de ficheros de texto

Los ficheros en Python son objetos de tipo `file` creados mediante la función `open` (abrir). Esta función toma como parámetros:

- una cadena con la ruta al fichero a abrir, que puede ser relativa o absoluta;
- una cadena opcional indicando el modo de acceso (si no se especifica se accede en modo lectura).
- un entero opcional para especificar un tamaño de *buffer* distinto del utilizado por defecto.

El modo de acceso puede ser cualquier combinación lógica de los siguientes modos:

Modo de acceso	Operador	Descripción
Lectura (read)	r	Abre el archivo en modo lectura. El archivo tiene que existir previamente, en caso contrario se lanzará una excepción de tipo <code>IOError</code>
Escritura (write)	w	Abre el archivo en modo escritura. Si el archivo no existe se crea. Si existe, sobrescribe el contenido.
Añadir (append)	a	Abre el archivo en modo escritura. Se diferencia del modo <code>w</code> en que en este caso no se sobrescribe el contenido del archivo, sino que se comienza a escribir al final del archivo.

```
>>> f = open("archivo.txt", "w")
```

Tras crear el objeto que representa un archivo mediante la método `open` se podrán realizar las operaciones de lectura/escritura pertinentes utilizando los métodos del objeto que se verán a continuación.

Important

Finalizada su utilización, se deberá cerrar el archivo con el método `close`.

Escritura de ficheros

Para la escritura de archivos se utilizan los métodos `write` y `writelines`. Mientras el primero funciona escribiendo en el archivo una cadena de texto que toma como parámetro, el segundo toma como parámetro una lista de cadenas de texto indicando las líneas que queremos escribir en el fichero.

```
obj_file_results = open("C:/TemporalC/archivo.txt", "w")
num_pto = 1
coord_x = 720487.27
coord_y = 4367654.23
crs_epsg_auth_id = "CRS-EPSG: 25830"
str_msg = "Coordinates point %d: (%.1f, %.1f) %s" %(num_pto, coord_x, coord_y, crs_epsg_auth_id)
obj_file_results.write(str_msg)
obj_file_results.close()
```

Lectura de ficheros

Para la lectura de archivos se utilizan los métodos `read`, `readline` y `readlines`.

El método `read` devuelve una cadena con el contenido del archivo o bien el contenido de los primeros `n` bytes, si se especifica el tamaño máximo a leer.

```
>>> completo = f.read()
>>> parte = f.read(512)
```

El método `readline` sirve para leer las líneas del fichero una por una. Es decir, cada vez que se llama a este método, se devuelve el contenido del archivo desde el puntero hasta que se encuentra un carácter de nueva línea, incluyendo este carácter.

```
1 while True:
2     linea = f.readline()
3     if not linea:
4         break
5     print(linea)
```

Por último, `readlines`, funciona leyendo todas las líneas del archivo y devolviendo una lista con las líneas leídas.

Mover el puntero de lectura/escritura

Hay situaciones en las que nos puede interesar mover el puntero de lectura/escritura a una posición determinada del archivo. Por ejemplo si queremos empezar a escribir en una posición determinada y no al final o al principio del archivo.

Para esto se utiliza el método `seek` que toma como parámetro un número positivo o negativo a utilizar como desplazamiento. También es posible utilizar un segundo parámetro para indicar desde dónde queremos que se haga el desplazamiento: 0 indicará que el desplazamiento se refiere al principio del fichero (comportamiento por defecto), 1 se refiere a la posición actual, y 2, al final del fichero.

Para determinar la posición en la que se encuentra actualmente el puntero se utiliza el método `tell()`, que devuelve un entero indicando la distancia en bytes desde el principio del fichero.

Extensión de la funcionalidad de QGIS mediante plugins



Capítulo en redacción con los siguientes contenidos previstos:

Introducción

Complementos del núcleo de QGIS

Complementos externos a QGIS

Inicialización de complementos

Complementos recomendados

Ideas finales

Descripción de las APIs de QGIS: API C++ y API Python



Capítulo en redacción con los siguientes contenidos previstos:

Introducción

La API de QGIS explicada a través de ejemplos

Consulta de la API QGIS en la consola de Python de QGIS

Configuración de QGIS y PyCharm para el desarrollo de plugins



Capítulo en redacción con los siguientes contenidos previstos:

Instalación de QGIS

Instalación de PyCharm

Configuración del entorno de desarrollo de QGIS

Primeros pasos con PyCharm

Estilo de codificación propuesto

Desarrollo de complementos en QGIS



Capítulo en redacción con los siguientes contenidos previstos:

Creación de un complemento básico de QGIS

Diseño y creación de la Interfaz de Usuario

Implementación de la funcionalidad del complemento

Indices and tables

- `genindex`
- `search`