PYTHON3

(Introducción a la programación)

Autor: Jesús Díaz

Contenido

Introducción

Este tutorial tiene el propósito de mostrar una base muy general en la preparación previa a la programación de computadoras a través de Python.

En la primera parte mostraremos una muy breve historia de la computación con el objetivo de ver su origen y evolución desde un marco puramente técnico.

La segunda parte es muy importante pues la filosofía de la informática nos será de gran utilidad para guiarnos, orientarnos hacía un objetivo u otro, ya que la informática desde sus inicios ha progresado de forma vertiginosa.

Decir que Python es un candidato con futuro, pues se utiliza en la programación científica, en la industria, programación de sistema y de red, creación de aplicaciones web, gráficas y un largo etc.

Espero que se comprenda como comenzar a programar en Python y que pueda aplicar lo aprendido para programar sus propios proyectos.

Capítulo 1.

Breve Historia de la computación

Como no podría ser de otra manera el cálculo surge de la necesidad del hombre en esencia para contar...

La mecanización aritmética

El ábaco es considerado como la primera “máquina” capaz de realizar cálculos, la época de origen es desconocida. Se cree que el origen se encuentra en China.

El quipu era una herramienta q utilizaban los incas para llevar el registro y la contabilidad, el más antiguo data del año 2500 a.C.

El **ábaco de Napier** es un ábaco inventado por John Napier … los productos se reducen a operaciones de suma y los cocientes a restas; al igual que con las tablas de logaritmos, inventadas por él mismo se transforman las potencias en productos y las raíces en divisiones.

En 1642, **Blaise Pascal** inventó para él la *roue pascaline*, «rueda de pascal» o Pascalina, considerada como una de las calculadoras más antiguas. Inicialmente solo permitía realizar adiciones, pero en el curso de los diez años siguientes añadió mejoras, siendo finalmente capaz de hacer restas. Las máquinas, trabajosamente confeccionadas una a una y a mano, llegó a fabricar cincuenta, de las que subsisten nueve.

Gottfried Wilhelm Leibniz se propuso la tarea de mejorar las máquinas de cálculo construidas hasta entonces: la de **Blaise Pascal** y la de **Samuel Morland**. a Calculadora Universal de Leibniz en 1694 no sólo sumaba y restaba, sino que también podía multiplicar y dividir. Leibniz ideó un dispositivo capaz de realizar múltiples sumas y restas: Este original dispositivo recibe el nombre de **rueda escalada de Leibniz.**

George Boole en 1835 comienza a estudiar por sí mismo matemáticas, estudiando a Newton, Laplace y Lagrange principalmente. Desde Platón no hemos parado de buscar las formas de desarrollo de un argumento lógico. La genialidad de Boole le llevo a argumentar que había una analogía cercana entre los símbolos algebraicos y los símbolos que representan las interacciones lógicas. También demostró que se podían separar los símbolos de calidad de aquellos símbolos de operaciones. Desarrolló un “proceso de análisis” que permitió a la gente cambiar procesos de pensamiento a pequeños pasos individuales. Cada paso implica hacer alguna proposición, que puede ser verdadera o falsa. Las respuestas de estos pasos se pueden combinar usando uno de los tres operadores: Y, O, o NO. Lo más simple de la idea es que se puedan coger dos propuestas al mismo tiempo y unirlas con un operador. Añadiendo muchos pasos, el álgebra booleana puede formar árboles de resolución complejos que presentan resultados lógicos a partir de una serie de aportaciones no relacionadas previamente.

Charles Babbage (1822) comenzó a trabajar en la máquina que llamó la Máquina de Diferencias (para calcular polinomios). Diseñó una maquina más sofisticada que llamó Máquina Analítica. Conceptos de sus diseños son usados en computadoras modernas.

El **telar de Jacquard** es un telar mecánico inventado por **Joseph Marie Jacquard** en 1801. El artilugio utilizaba tarjetas metálicas perforadas para conseguir tejer patrones en la tela, permitiendo

que hasta los usuarios más inexpertos pudieran elaborar complejos diseños. (Estas tarjetas metálicas perforadas fueron sustituidas por cintas magnéticas, los principios de la grabación magnética fueron obra del inglés **Oberlin Smith** en 1878)

En 1890, el estadounidense **Herman Hollerith** utilizó la tarjeta perforada para recopilar y analizar la información del censo de Estados Unidos y potenció el uso de esta herramienta hasta los años 50 del siglo XX. En 1924 construyó la empresa IBM.

En 1944 la invención de la computadora electromecánica Mark I, financiada por IBM. (se considera la primera realización de la Máquina Analítica de Babbage.

La primera computadora totalmente electrónica fue desarrollada por **John Atanasoff** y con ayuda de **Clifford Berry**, construyó un prototipo en 1939 y completó el primer modelo funcional en 1942. ENIAC la computadora construida en 1946 por **J.P. Eckert** y **J.W.Manchly**, contenía más de 18000 válvulas de vacío y 1500 relés y casi llenaba una habitación de tamaño 6x12 metros (la programación se llevaba a cabo conectando manualmente cables... y obviamente con propósitos militares) Posteriormente en 1951 construyó la UNIVAC Eckert-Mauchly.

El concepto de programa almacenado (pueden ser modificadas por la propia computadora mientras está calculando).

Generaciones de la computación:

1 generación ---> Válvulas de vacío, surgieron las primeras bases.

2 generación ---> IBM 7090 (1959 y 1965) usaban transistores en lugar de válvulas de vacío. Se desarrollan las necesidades industriales.

3 generación ---> Circuitos integrados (multiprogramación) IBM System/360, aparece la portabilidad.

4 generación ---> Chips de silicio. Un chip típico es equivalente a miles de transistores. (tamaño aproximado <1mm), aparecen los lenguajes de alto nivel.

Robert Noyce cofundador de la empresa Intel introdujo el microprocesador 4004 en 1971.
Los microprocesadores como Intel 1001 hicieron posible la computadora popular APPLE II en 1977 desarrollada por **Steven Jobs** y **Steve Wozniak**. A continuación, en 1981 se introdujeron las primeras computadoras personales (Pcs) de IBM.
(¿Qué pasó con el progreso en la industria automovilística?)

A finales de los setenta, los avances tecnológicos en la integración de circuitos conllevaban el desarrollo de las microcomputadoras, o también llamadas computadoras personales, porque sólo las utiliza una persona a la vez y las primeras redes funcionaban por conexiones telefónicas clásicas.

Las grandes computadoras de propósito general se utilizan en muchos negocios, universidades, hospitales y agencias gubernamentales para desarrollar sofisticados cálculos científicos y financieros. A estas grandes computadoras o grandes sistemas se las conoce como MAINFRAMES.

Acerca de Internet

Parece que el público confunde internet con la World Wide Web, veamos:

Internet (Red física), sus orígenes se remontan a 1969, cuando se estableció la primera conexión de computadoras, conocida como ARPANET, entre tres universidades en California (Estados Unidos).

En 1980, **Tim Berners-Lee**, un contratista independiente en la Organización Europea para la Investigación Nuclear (CERN por sus siglas en inglés), Suiza, desarrollo ENQUIRE, como una base de datos personal de gente y modelos de software, pero también como una forma de interactuar con el hipertexto;

En 1990 **Berners-Lee** había desarrollado todas las herramientas necesarias para trabajar la Web: el Protocolo de transferencia de hipertexto (HTTP por sus siglas en inglés) 0.9, el Lenguaje de Marcado de Hipertexto (HTML por sus siglas en inglés), el primer navegador web (llamado World Wide Web, que fue también un editor de páginas web), el primer servidor de aplicaciones HTTP (luego conocido como CERN httpd), el primer servidor web (http://info.cern.ch) y las primeras páginas web que describían el proyecto mismo.

En resumen, la World Wide Web aparece a principios de los 90, gracias al lenguaje HTML y el navegador Netscape 2.0.

Green Computing también conocido como Green IT o traducido al español como Tecnologías Verdes se refiere al uso eficiente de los recursos computacionales minimizando el **impacto ambiental**, maximizando su viabilidad económica y asegurando deberes sociales.
El término de green computing comenzó a utilizarse después de que la Agencia de Protección Ambiental (EPA, por sus siglas en inglés) de los Estados Unidos desarrollara el programa de Estrella de Energía en el año de 1992. (recordar que las primeras conexiones tenían una velocidad de solo decenas de Kbps)

Teoría de la Programación

Filosofía de la informática

Como en cada lenguaje humano, lo primero que se debe aprender es su construcción.
Una construcción parecida a traducir nuestro lenguaje natural (castellano, inglés, etc) en un lenguaje formal (como las matemáticas y la lógica).

En el idioma castellano por ejemplo tenemos nombres, verbos y adjetivos con los que crear la *estructura básica del lenguaje*.

Lo mismo ocurre con la máquina, con funciones, sentencias y operadores que controlan la manera en que lee el lenguaje la computadora.

¿ Qué significa el término informática ?
Tratamiento automático de la información (**Informática** = *Información* + *automática*).

¿ Pero qué es una computadora ?
Una computadora solo entiende cadenas o grupos de dígitos con dos posibles estados 0 o 1, es decir, una computadora habla en lenguaje binario.

Decir, que los microprocesadores emplean un lenguaje llamado ensamblador.

Reflexionemos:
Supongamos en no existe ningún tipo de computadora, ni moderna ni clásica, nada.

¿Cómo construir una computadora?

La idea de estas preguntas es situarnos en el mundo de la computación, pensar que vamos a programar para algo que ya está programado electrónicamente y en lenguajes binario y ensamblador. Por tanto, este supuesto inicio a la programación no es más que una extensión complementaria de un sistema operativo bien definido. El símil sería si deseamos introducirnos a la matemática de las variedades, pues estas son una extensión del álgebra lineal, estructuras algebraicas, geometría diferencial, topología, …

Por eso la complejidad de introducirse en la informática en este sentido sin aclarar cuales son sus bases, de donde viene, cuáles son sus objetivos, nos conduciría a un camino oscuro.

¿Cómo construir un lenguaje que entienda la máquina?

Observación: “¿El software sigue al hardware? “

¿Cómo empezamos, que “escribimos”, …?

¿Qué necesitamos para programar?

Podemos pensar en definir lo que es un programa o algoritmo o código lógico pues son términos semejantes. ¿Pensar en qué forma específica? Pues en la forma que está diseñada a procesar la computadora que es aplicar la lógica de proposiciones, sus operadores, sus conectores...

El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas, además del dominio del lenguaje a utilizar, algoritmos especializados y lógica formal.

(¿qué contradictorio cierto ?, lo complicado es todo menos programar...)

Un **algoritmo** es una secuencia no ambigua, finita y ordenada de instrucciones que han de seguirse para resolver un problema. Un programa normalmente implementa (traduce a un lenguaje de programación concreto) uno o más algoritmos. Un algoritmo puede expresarse de distintas maneras: en forma gráfica, como un diagrama de flujo, en forma de código como en pseudocódigo o un lenguaje de programación, en forma explicativa.

Es bien sabido, que siempre se ha dicho que por sí mismo un ordenador no sabe hacer nada: solo es capaz de ejecutar órdenes. ¿Pero... acaso es tan poco ejecutar ordenes? O tal vez es nuevamente el interés del hombre la manipulación de la enseñanza.
¿Hoy día, en 2018, diríamos lo mismo ?, ¿Cómo ?, ¿Qué Google no sabe hacer nada?

A la pregunta: ¿Una computadora tiene emociones?
La respuesta: ¿Se ha programado alguna para que las tenga?

¿Para qué se usa un determinado lenguaje de programación?
¿Carencias de otros lenguajes? ¿Cuál es nuestro objetivo?
Hoy día se sigue trabajando en cobol, yo mismo programe en el año 2009 en Cobol para una entidad financiera, entonces si hoy convive un lenguaje de programación del año 1960 con digamos GO (desarrollado por Google) del año 2009. ¿Cuál lenguaje de programación escogemos?
¿Un lenguaje estructurado como C, Cobol, Pascal?
¿Un lenguaje orientado a objetos (POO)?
¿Lenguaje como JavaScript del lado del cliente?
¿Lenguaje como Php o Asp del lado Servidor?

Lo primero es seleccionar una plataforma (esta define un entorno de trabajo específico), es decir un hardware y su sistema operativo, por analogía nos haríamos las mismas preguntas anteriores respecto a seleccionar una plataforma u otra.

¿Qué son los programas?
Los programas de computadora son simulaciones digitales de modelos conceptuales y físicos.
Un estudio psicológico serio descubrió que un humano medio solo puede comprender simultáneamente siete +- dos informaciones.

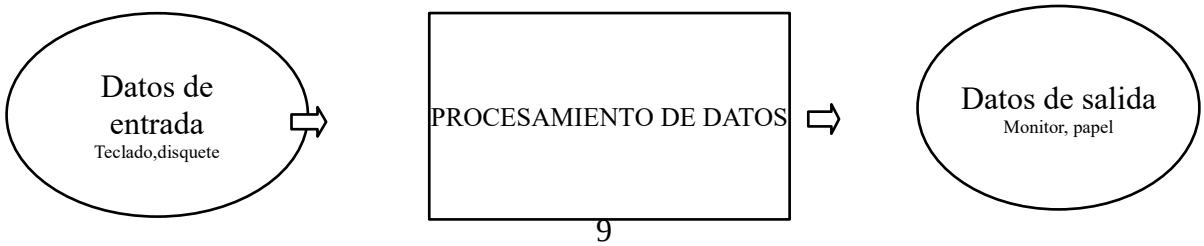
Antes de definir lo que es un programa, recordamos que es imprescindible motivar al alumno a escribir, ejecutar el mayor número posible de estos programas. Una vez hechos los ejemplos de clase debe diseñar sus propios programas, resolver y entender cada programa, que le conducirá a la claridad, legibilidad, modularidad y eficiencia requerida en el diseño de un programa.

Concepto y definición de programa

Programa (Software en inglés) es un conjunto o secuencia de instrucciones que una computadora puede interpretar y ejecutar.
Un programa está formado por algoritmos y estructura de datos.

Lenguaje de programación es el idioma utilizado para controlar el comportamiento de una computadora. Consiste en un conjunto de símbolos y reglas sintácticas y semánticas que definen su estructura y el significado de sus elementos y expresiones.

Ejecución o funcionamiento de un programa.



Nota: Cuando yo empecé a programar en clipper, había la dificultad de que programabas con una ram de 256 Kb y la consecuencia no poder tener abiertos más de 10 ficheros.

¡De tal modo que si pasabas esos límites daba error, demasiados ficheros abiertos!
Uno de estos límites era el número máximo de subrutinas que podía contener un programa, que eran 32. Se podía ajustar estas cosas en config.sys: files = 20, buffers = 15 (Kb)

Por tanto, la problemática esencial del programa era el tamaño del código ejecutable y la cantidad de recursos, así como el tiempo de ejecución.

Otras limitaciones en los inicios de la programación de computadoras eran los compiladores, pues el compilador para un programa cobol por ejemplo no funcionaba de igual forma en distintos computadores, pues cada computadora tenía sus especificaciones propias de su fabricante.

De este punto surge la necesidad de pensar cómo el procesador va a ejecutar nuestras sentencias para garantizar que funcionen adecuadamente y por otro lado en el menor tiempo posible.
(claro hoy en día los procesadores son tan rápidos que un programa mal diseñado también puede ir muy rápido)

Modos de operación de una cpu:

Un programa se puede ejecutar de los siguientes modos:

Modo interactivo y modo de procesamiento por lotes.
Sistemas de tiempo compartido (red por módems)

Ensambladores, compiladores e intérpretes

Por la forma como se ejecutan Hay lenguajes compilados e interpretados. En seguida veremos los distintos tipos de lenguajes.

¿Qué es un compilador?
Una computadora dispone de un procesador y este dijimos que solo sabe hablar en ceros y unos.

Un **ensamblador** traduce una instrucción de lenguaje simbólico a otra instrucción en lenguaje de máquina, la traducción es de 1 a 1.

Un **compilador** traduce un programa fuente en un lenguaje de máquina. Una instrucción de alto nivel puede ser el equivalente varias instrucciones a nivel de máquina, la traducción es de 1 a muchos.

Un compilador realiza la traducción una sola vez y genera un programa permanente, que pueda ejecutarse tantas veces como se desee.

El compilador genera el fichero para que el sistema operativo pueda traducirlo finalmente a ceros y unos.

Un **intérprete** (compilador Jit , just in time) analiza las instrucciones una a una o en pequeños grupos del programa, las transforma al lenguaje de la máquina y las ejecuta, después de lo cual las instrucciones traducidas se pierden.

Un compilador es un ejemplo de **programa**, pues un compilador o intérprete es en suma un programa que acepta un programa de alto nivel como datos de **entrada** y genera el correspondiente programa en lenguaje maquina como **salida**.

Se creo el standard ANSI que consiste en la definición de un conjunto de librerías que acompañan al compilador y de las funciones contenidas en ellas, de esta forma los programas que utilicen estas bibliotecas para interactuar con el sistema operativo obtendrán un comportamiento equivalente en otro sistema. (ANSI → Instituto Nacional Americano de Estándares)

Tipos de lenguaje de programación

De propósito general:
- Bajo nivel → lenguaje binario
- Medio nivel → lenguaje ensamblador, lenguaje C
- Alto nivel → lenguaje C, Pascal, Java, Python, ...

De propósito especial:

Simulación → GPSS, CSMP, SIMSCRIPT II.5, Simula (descendiente de ALGOL, introduce el concepto de clases), …

Inteligencia Artificial → Lisp, Prolog, Haskell (cálculo lambda y lógica combinatoria), ...

Observación:

Smalltalk
Es el primer lenguaje orientado a objetos en el año 1972. Utiliza una máquina virtual, idea que Java retoma con su JIT y presenta el concepto de **MVC**. Modelo Vista Controlador (MVC) es un estilo de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

La programación en la actualidad

¿Programo para la web o para el escritorio? ¿Realizo desarrollo nativo o multiplataforma?

En la programación web (sistemas distribuidos), al inicio de la Word wide web los servidores de internet procesaban el 100% de la información y al cliente le llegaba el HTML puro y duro. Actualmente el servidor envía datos a través de JSON al cliente y este se encargará de producir el HTML. Hoy creamos programas con unos pocos clics, pues el desarrollo web está en dominio de la gestión de contenidos (CMS) donde el trabajo principal es personalizar y no programar.

Tipología de los lenguajes de programación (Paradigmas)

Paradigma o modelo de programación
Un paradigma es una representación, mediante un modelo teórico coherente, de una visión particular del mundo y permite al desarrollador describir algoritmos.

Programación imperativa o por procedimientos o estructurada son equivalentes.
C, FORTRAN, COBOL y Pascal implementan el **paradigma imperativo**, divide el problema en partes más pequeñas, que serán realizadas por subprogramas (subrutinas, funciones, procedimientos), que se llaman unas a otras para ser ejecutadas (recursividad).

Eiffel, Java y Smalltalk implementan el **paradigma orientado a objetos**, los humanos gestionamos la complejidad a través de la abstracción. No pensamos que nuestro coche es una lista de decenas de miles de partes insondables. Pensamos que es un objeto bien definido con un comportamiento propio único. Los programas algorítmicos tradicionales se pueden descomponer en los objetos con los que trata el proceso. Estructurar el código en bloques reutilizables.

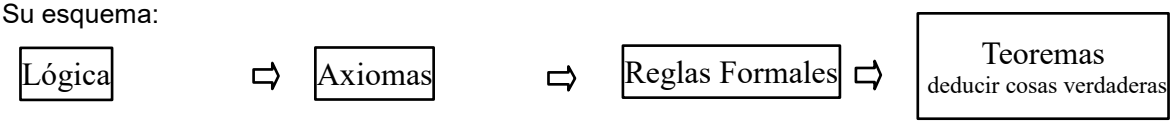
POO mediante clases
Consiste en la escritura de clases (C++, Java, etc)

POO mediante prototipos
Permite definir un nombre de clase, padres y agregar métodos (Lua, JavaScript).
En la programación mediante prototipo no existe la noción de instanciación. Los objetos no son más que contenedores de métodos estáticos.

El **paradigma funcional**, permite centrarse en la reflexión acerca de los datos, y no en los algoritmos. Es un paradigma de programación declarativa basado en el uso de funciones matemáticas, en contraste con la programación imperativa, que enfatiza los cambios de estado mediante la mutación de variables. La programación funcional tiene sus raíces en el cálculo lambda, un sistema formar desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión.

Ejemplos de lenguajes funcionales son:
Lisp, Scheme, Haskell, Objective Caml, Erlang, Rust, Scala, F#, R (estadística), J y K (análisis financiero).

Su esquema:



Prolog implementa el **paradigma lógico**, donde los datos resultan mucho más importantes que el algoritmo. Se asemeja a la teoría de grafos y es utilizada en la inteligencia artificial.

AspectJ implementa el **paradigma orientado a aspectos**, encapsular los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias entre cada uno de los módulos. Un programa se convierte en un entrecruzado de diversas preocupaciones. Python dispone de los módulos: aop, aspyct, aspects, Sprint Python, etc.

Programación concurrente
Para permitir realizar varias tareas simultáneas.

Programación por contrato
Permite desarrollar en función de una serie de precondiciones y postcondiciones. (Eiffel)
Python también permite usar este paradigma con su módulo dedicado pycontract.

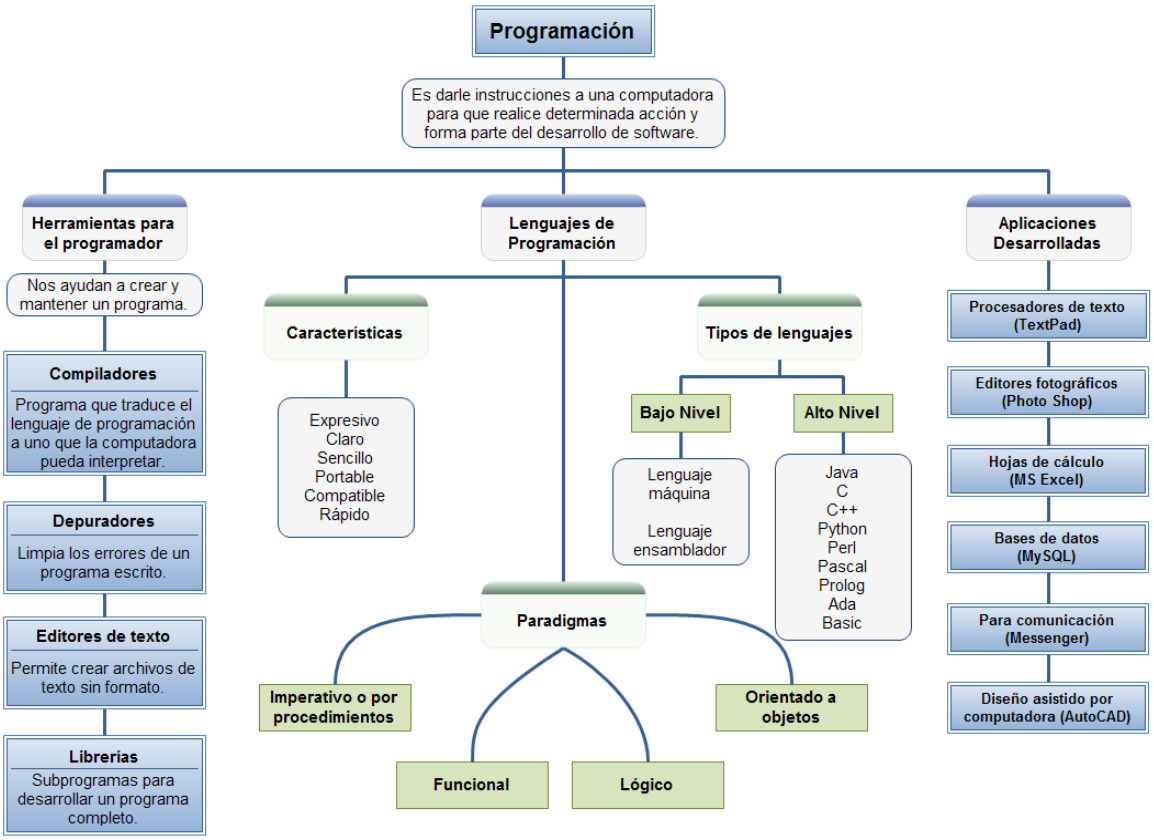
Entorno de Desarrollo Integrado

Entorno de Desarrollo Integrado (en inglés Integrated Development Environment 'IDE'): Es un programa compuesto por un conjunto de herramientas para un programador. Estos IDE son programas que sirven para programar, ya sea en un lenguaje de programación o en varios lenguajes. Los IDE que permiten crear programas en muchos lenguajes de programación permiten usar un solo programa para trabajar en varios lenguajes de programación, es decir no limitan al programador.

Ejemplos de IDES:

- Gambas (lenguaje derivado de BASIC),
- Eclipse (lenguaje Java),
- Kdevelop (varios lenguajes),
- Netbeans (varios lenguajes: java, php, C/C++)
- Visual Studio (varios lenguajes: C, C++, C#, Visual basic, ASP, Javascript)
- RAD Studio (Delphi, C++)

Esquema resumen:



Capítulo 2.

El lenguaje de programación Python

¿Qué es Python?

Python es un lenguaje de propósito general (open source), de alto nivel, interpretado y que admite la aplicación de diferentes paradigmas de programación, como son, por ejemplo, la programación procedural, imperativa y la orientación a objetos principalmente.

Python se emplea en la programación científica, la programación de sistemas o de aplicaciones web. También para desarrollar aplicaciones de escritorio con interfaz gráfica de usuario o incluso desarrollar juegos.

Puede ser utilizado en diversas plataformas y sistemas operativos, como Windows, Mac os x y Linux. También puede funcionar en smartphones, Nokia desarrolló un intérprete de este lenguaje para su sistema operativo Symbian.

Python se inspira en los lenguajes ABC inspirados a su vez en ALGOL y pensados para suceder a Basic, Pascal y Awk. En ALGOL un programa se describe por una serie de algoritmos, incluye recursividad. Éxito universitario, pero no industrial.

Ventajas

Es libre y gratuito. Es un lenguaje muy versátil, puesto que trabaja con numerosos paradigmas y algoritmos. Hacer cosas como controlar un robot, impresoras 3D o servomotores, trabajar con matemáticas científicas (como propone **MATLAB**) o incluso crear una interfaz gráfica o un pequeño sitio web. Ofrece excelentes librerías científicas y al ser libres estas el investigador puede distribuir su proyecto libremente.
Python es una emergencia de soluciones tales como Django o Twisted.
Python es la única alternativa libre, potente y diversa para el cálculo científico.
C/C++ y Python pueden trabajar de manera conjunta, interactuar con otros lenguajes.
Rpy2 es una interfaz que permite que podamos comunicar información entre R y Python y que podamos acceder a funcionalidad de R desde Python.
También interactuar con bases de datos, directorios LDAP, servicios web, archivos de datos (xml comprimidos, imágenes, vídeos o sonido, xml, cvs, pdf. También puede interactuar con el sistema de archivos, con la red, Internet... Extensiones de Mozilla pueden realizarse con Python.
Python puede alojarse en Apache y el desarrollo es más rápido que en Php.
Gestión de paquetes para Unix, uso de protocolos en Python para Cisco, también Cloud computing, redes sociales Bit.ly, Evite … API python-youtube para escribir un cliente YouTube, Spotify, ZeOmega...

Inconvenientes

Es algo “lento” a la hora de realizar cálculos en comparación de un método óptimo de C.
Su poca difusión respecto a otros lenguajes como C o C++ y Java. (no hay suficientes desarrolladores Python)

Breve historia de PYTHON

Su creador

La primera versión pública es la 0.9.0, publicada en un foro de Usenet en febrero de 1991 y su autor es Guido Van Rossum que trabajaba en el equipo del sistema operativo Amoeba y en sus ratos libres comienza a desarrollar Python. El origen del nombre fue en honor a la serie televisiva Monty Python Flyng Circus, de la cual era fan. La licencia evoluciona hacia GPL en la versión 1.6.1.

Desarrollo y promoción de Python
Se lleva a cabo a través de una organización, sin ánimo de lucro, llamada Python Software Foundation, que fue creada en marzo de 2001. Python tiene un claro carácter open source.

Principales características de Python

A los programas en Python se les denomina scripts.
Saber que para Python todo es un objeto y que los tipos son clases como cualquier otra y así podemos sobrecargar los tipos y se dice que Python es fuertemente tipado.
Python es un lenguaje de alto nivel - permite también trabajar a bajo nivel, puesto que gestiona recurso de forma automática.
Los paradigmas imperativo y procedural están también presentes.

Es posible trabajar utilizando únicamente el paradigma imperativo, si bien lo que se manipula son realmente objetos, útil para realizar scripts cortos.
Python tiene vocación de ser un lenguaje universal y de los denominados multiparadigma.

Integración con otros lenguajes

Extensiones C
Python es un lenguaje que dispone de varias implementaciones, la más común de ellas es Python, escrita en C. Podemos integrar programas escritos en C en nuestro Python realizando bindings, es decir, extensiones de C y recíprocamente gracias a la existencia de boost::Python.

También podemos integrar programas Java, la implementación está escrita en Java llamada Python, así incrustar java en nuestro Python o cargar un archivo JAR. Raramente se incrusta Python en Java.

Existen otros módulos de Python que permiten importar código realizado en otros lenguajes (Fortran,Lisp,Scheme...)

Permite realizar scripts de sistema como los lenguajes Shell (sh,csh,ksh,zsh,bash...)
Docstring herramienta por parte de Python para documentar el código.

Instalar Python

En **Windows** python.org/download/ …
Preinstalado en **Mac** …
En **Linux**, aptitude o yum install python3.
En caso de necesidades concretas, descargar los archivos fuente y ./configure,make o make install.
Podemos mantener varias versiones de Python la 2 y 3, y crear un enlace simbólico para que por

ejemplo el comando Python apunte directamente a la versión 3.
ln -s /usr/bin/python3 /usr/bin/python

Instalar librerías externas

Pip es actualmente el gestor de paquetes de referencia para Python, viene contenido en la instalación de Python.
$ pip install <nombre_del_paquete> (busca en la web de Pypi), install -U <nombre_del_paquete> actualiza el paquete.

Entorno virtual

En caso de no querer alterar nuestra instalación tenemos esta alternativa.

aptitude o yum install python-virtualenv

Para crear un entorno virtual:
$ virtualenv -p python3 nombre entorno

Para entrar en dicho entorno virtual:
$ cd nombre_entorno
$ source bin/activate

Instalar un IDE

En la web podemos encontrar Eclipse, PyDev, Aptana, Eric, Spyder, StacklessPython.
Entorno heterogéneo mediante Pyro. (Comunicarse entre diferentes lenguajes)

Consola estándar

Trabajar con la consola nos permite realizar operaciones tales como probar el código antes de copiarlo al archivo del código fuente.
Otra herramienta más interactiva es la **consola Python**.

Distribuir sus propias aplicaciones
Ver la herramienta distutils (py2exe, py2app), ver el concepto cx_Freeze.

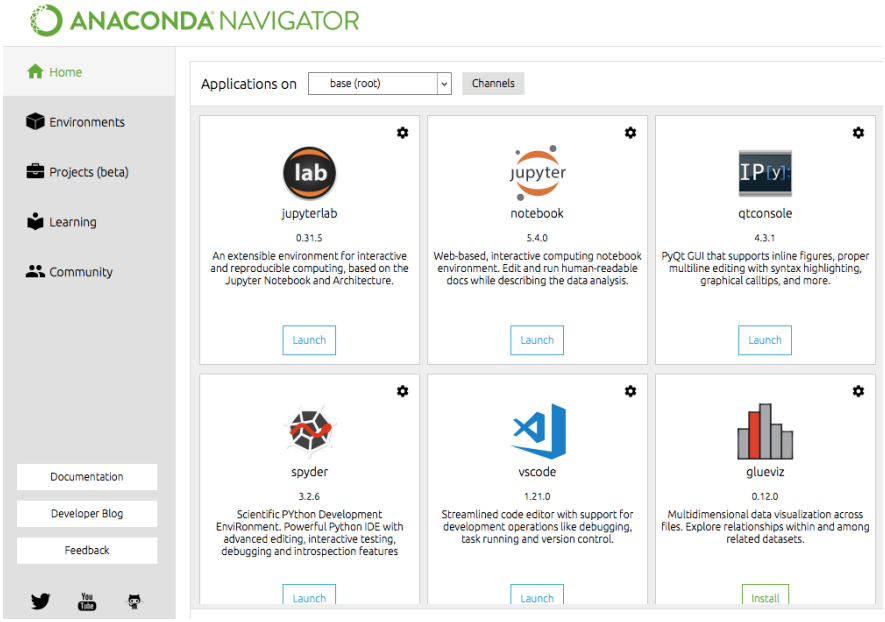
Anaconda

Anaconda Distribution se agrupa en 4 sectores o soluciones tecnológicas, **Anaconda Navigator**, **Anaconda Project**, Las **librerías de Ciencia de datos** y **Conda**. Todas estas se instalan de manera automática y en un procedimiento muy sencillo.
Libre, de código abierto, con una documentación bastante detallada y una gran comunidad.

- Multiplataforma (Linux, MacOS y Windows).
- Permite instalar y administrar paquetes, dependencias y entornos para la ciencia de datos con Python de una manera muy sencilla.
- Ayuda a desarrollar proyectos de ciencia de datos utilizando diversos IDE como Jupyter, JupyterLab, Spyder y Rstudio.

Instalación en Mac

https://www.anaconda.com/what-is-anaconda/
Anaconda 5.1 For MacOS Installer
Anaconda3-5.0.1-MacOSX-x86_64.pkg
Y una instalación standard siguiente, siguiente, ...



Editores genéricos

Editores genéricos para programar en diferentes lenguajes de programación:

Unix → Vim, Vi, Emacs, gedit, TextMate (de pago), TextWrangler, etc.

Multiplataforma: Notepad++, Komodo Edit, ...

Editor específico Python: Toolkit QT → Eric, Zope, PyQT, PyGTK, Django y wxPython.

Intérprete interactivo mejorado: Ipython.

Depuradores

Herramienta pdb:

Uso: import pdb
pdb.set_trace()
$python3x.x -m pdb nombreFichero.py

Prolifing

Medir el rendimiento de la ejecución de otro programa (estadísticas).

(cProfile, profile y hotshot)

$python3.x -m cProfile nombreFichero.py , (timeit → tiempo que tarda la ejecución)

Los fundamentos del lenguaje Python

El intérprete

Abrimos una consola o terminal : en mi portátil se tiene:
sh-3.2# python3.7
Python 3.7.0a4 (v3.7.0a4:07c9d8547c, Jan 8 2018, 19:27:01)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>

Pues esta última línea comienza por los caracteres >>> y nos muestra el cursor parpadeando. Este es el prompt del intérprete que nos permite interactuar directamente con él. El prompt nos indica que está listo para recibir comandos y ejecutarlos.

Ejemplo si tecleamos copyright y pulsamos enter nos devuelve la info sobre copyright de Python.

Ejemplo escribir el siguiente comando y pulsamos enter:

>>> print(“Hola mundo”) si no funciona, sustituir “ → ' depende versión Python

Primer programa en Python

Para nuestra comodidad nuestro código será ejecutado desde un fichero de texto generado con extensión .py, por ejemplo hola.py.

(Utilizaremos el intérprete para expresiones que nos puedan generar alguna duda sobre su funcionamiento.)

$ editor de texto hola.py, que nos genera tal archivo y escribimos print ('hola mundo'), guardamos y salimos de nuestro editor.

Seguidamente desde la consola ya podemos ejecutar nuestro nuevo archivo Python.

$ python3.7 hola.py

Código fuente y bytecode

El intérprete Python se encarga de generar unos ficheros binarios que son los que serán ejecutados. Este proceso se realiza de forma transparente, a partir de los ficheros fuente. Al código generado automáticamente se le llama bytecode y utiliza la extensión .pyc. Así pues, al invocar al intérprete de Python, este se encarga de leer el fichero fuente, generar el bytecode correspondiente y ejecutarlo. (así Python no vuelve a leer el fichero fuente, sino que lo ejecuta directamente)

En ocasiones necesitamos ejecutar nuestro código en máquinas que no disponen de este intérprete. Suelen darse en sistemas Windows. Para salvar este obstáculo contamos con programas como py2exe, que se encarga de ejecutar un binario para Windows (.exe) a partir de un fichero fuente escrito en Python.

Capítulo 3.

Algoritmos en Python

Sintaxis en Python

Errores comunes

Razones por las que sus scripts pueden no funcionar correctamente:

Permisos de archivo. (Linux/Mac)

Formatos del editor de código (donde no mantiene concordancia con el sistema, Unicode, ASCII, binario, etc.)

Path no incluye el path a Python

Ejecutar el script desde el propio intérprete (consola).

Falta algún archivo externo propio a Python, o en nuestro script hacemos una llamada a una función que requiere importar una librería en particular.

El punto y coma

Puede servir para separar varias instrucciones diferentes que se escribirían en la misma línea:

```
>>> a =1;a*=5;print(a)   (se utiliza mucho)
```

El comentario

```
# esto es un comentario (porque al texto le precede un carácter de almohadilla)
```

Una instrucción en varias líneas

Por razones de legibilidad usaremos el símbolo \ para dividir líneas:

```
unknown_relation_ranks, avg_precision = \
            self.get_unknown_relation_ranks_and_avg_prec(item_distances, unknown_relations,
             known_relations)
```

ejemplo trivial

```
>>> my_str='Ejem\
…    plo'
>>> my_str
```

en nuestro fichero:

```
#ejemplo trivial
my_str="Ejem\
plo"
print(my_str)     #observamos que aquí si es necesario invocar a la instrucción print
```

Palabras clave

Permiten estructurar los algoritmos y no se pueden modificar.
Contiene 33 palabras clave, por nombrar algunas : def, del, for, from, if, return, try, else, …

Palabras reservadas

Son nombres que se corresponden con funciones, clases o módulos usuales. No se recomienda utilizar estos nombres para declarar variables, ya que Python no lo prohíbe.

Observación: En python 3, print es una simple palabra reservada y en Python 2 es una palabra clave.

Símbolos

Indentación

Una de las diferencias más destacables es el uso de la indentación. Diferentes niveles de indentación son utilizados para marcar las sentencias que corresponden al mismo bloque. A diferencia de lenguajes como C/C++, Php y Java, cada sentencia no necesita un punto y coma (;). En Python basta con que cada sentencia vaya en una línea diferente. Tampoco se hace uso de las llaves ({}) para indicar el principio y fin de bloque. Simplemente se utilizan los dos puntos (:) para marcar el comienzo de bloque y el cambio de indentación se encarga de indicar el final.

Por este motivo la indentación es estricta en la codificación de un programa.

Por motivos de legibilidad e interpretación del flujo de datos se recomienda su uso.
La indentación es un desfase hacia la derecha de una o varias líneas de código. Típicamente se aplica al escribir un bloque de código de una cierta condición.

Ejemplo: Hacemos la identación con 4 espacios (mejor no usar tabulaciones)

```
if total_compra > 100:
    tasa_descuento = 10
    importe_descuento = total_compra * tasa_descuento / 100
    importe_a_pagar = total_compra – importe_descuento
```

En Python no existen las llaves. Delimitamos el bloque mediante dos puntos al final de la línea.

```
for capital,estado in pares:
    capitales.append(capital)
    estados.append(estado)
```

Los paréntesis

Sirven para escribir algoritmos, definir n-tuplas, generadores, invocar a una función o para instanciar un objeto. Lo que se encuentra entre paréntesis define su propio significado (una coma para separar tuplas, palabras clave para generadores).

Uso aritmético:

```
a = (1+2) / 3
```

He aquí tuplas donde la coma es el elemento esencial que caracteriza a una n-tupla:

```
a = (1, 2)
```

Ejemplo de llamada a una función:

```
 f((1, 2))
```

Los corchetes

Sirven para definir una lista de valores cuando se utilizan solos:

```
>>> j = [1, 2, 3]
```

En cambio, ligados a una variable, definen una palabra clave o un índice (o intervalo si se indican dos puntos):
>>> j[1]
2
o franja
>>> j[1:]
[2,3]

Un índice es un número entero que permite ubicar un elemento dentro de una colección ordenada.

Una clave es un objeto cualquiera que sirve para encontrar un elemento en una colección que asocia un valor a una clave, como es el caso de los diccionarios.

Las llaves permiten definir un conjunto o un diccionario, en función del uso de los dos puntos.
>>> diccionario = {'clave': 'valor1', 'clave2': 'valor2'}
>>> conjunto = {1, 2, 3}

Un **conjunto** lo es en el sentido matemático del término (permiten realizar la unión y la intersección).

El **diccionario** es una colección que asocia un valor con una clave. Cada clave es única.

Es posible también recorrer diccionarios
>>> d = {chr(i): chr(i+32) for i in range(65, 91)}

y conjuntos
>>> e = {i**2 for i in range(10)}

Los dos puntos sirven para delimitar la separación entre una clave y un valor en un diccionario:
>>> diccionario = {'clave': 'valor1', 'clave2': 'valor2'}

El punto sirve para acceder a un objeto y también como indicador decimal para los números:
>>> type(42)
<class 'int'>

>>> type(42.0)
<class 'float'>

El cero es opcional, aunque se recomienda escribirlo por motivos de legibilidad.

Como hemos visto el punto nos permite hacer dos cosas distintas.
En general el primer punto es siempre el separador entre la parte entera y la parte decimal.

Ahora (acceso a objeto):
>>> 1..real
1.0
el segundo punto indica acceso al objeto y permite utilizar el atributo real del objeto entero 1.

>>> (1).real
1
así con un solo punto para acceder al objeto hay que poner paréntesis.

Operadores

El significado de operador en Python está asociado no solo al propio operador sino a los objetos sobre los que se les aplica.

Operadores aritméticos

Operador	Descripción	Ejemplo	Resultado
+	Suma	result = 5 + 6	11
-	Resta	result = 5 – 1	4
*	Multiplicación	result = 2 * 6	12
**	Exponente	result = 2**3	8
/	División	result = 12 / 6	2
//	División Entera	result = 4.5//2	2.0
%	Módulo	result = 7 % 2	1

```
contador += 1          # es equivalente a contador = contador + 1
porc = 5               # asigna número entero a variable
total *= porc / 100    # es equivalente a total = total * porc/100
valor = -5             # el signo "-" también se usa para los nº negativos
```

Operadores binarios

Los operadores binarios emplean en sus operaciones la representación binaria de los datos. Los operadores binarios son:

Operador binario	Descripción
>>	Desplazamiento (*shift*) izquierdo
<<	Desplazamiento (*shift*) derecho
&	and binario
^	xor (or exclusivo) binario
|	or binario
~	not binario

Son también operadores:

(), [], {}, ""	Tuplas, listas, diccionarios y cadenas

Para profundizar, dar un significado particular a los operadores ver capítulo Modelo de objetos, Tipo de datos y algoritmos aplicados.
Ejemplo → normalizar operadores de modo que si:
a * b funciona , **b * a** funcione también sean cuales sean los tipos de **a** y de **b**.

Instrucciones

Definiciones

Variable
Una variable es una palabra que empieza por una letra minúscula o mayúscula y que contiene únicamente letras, cifras y el carácter de subrayado.

Por convención, las variables, las funciones no contienen más que letras minúsculas-cifras. Si están compuestas por varias palabras, se separan mediante caracteres de subrayado:

```
>>> mi_variable_util
>>> mi_funcion_util_42()
```
Por el contrario, los nombres de las clases se escriben con su primera letra mayúscula.
```
>>> MiClaseUtil42()
```

Declarar una variable

Para declarar una variable, se utiliza el operador de asignación (=) y situando en la izquierda el nombre de la variable (contenedor) y en la derecha su valor (contenido):

```
>>> ejemplo = 42
>>> type(ejemplo)
<class 'int'>
```

y de esta forma el contenido, aquí 42, establece el tipo de la variable ejemplo.

Asignación múltiple:

```
>>> a, b = 1,2
>>> a,b
(1, 2)
```

Como Python es un lenguaje tipado dinámicamente, esto significa que no es necesario ninguna declaración previa. Es posible utilizar el mismo nombre de variable más adelante para describir una variable con un tipo distinto.

Recordar que en Python todo es un objeto. Aquí, ejemplo es un objeto de tipo entero.
Varios punteros pueden apuntar al mismo objeto en memoria, la **palabra clave is** (comparación entre objetos) nos indica si dos punteros apuntan al mismo objeto o no. (true/false)

```
>>> ejemplo1 = ejemplo
>>> ejemplo is ejemplo1
True
```

Función

Para definir una función es necesario anteponer a su firma la palabra clave **def** y a continuación, escribir un **bloque** que contenga su código. Este bloque está delimitado mediante el carácter de dos puntos y al menos una línea de código indentada; el final de la indentación indica el final del bloque: (reglas de la gramática)

```
>>> def say_hello(to):
...     print("Hello %s!" % to)
```

Muy importante, decir que, una función no es más que una variable de tipo función:

```
>>> type(say_hello)
<class 'function'>
```

Puede, por tanto, utilizarse como una variable:

```
>>> say_hello1 = say_hello
>>> type(say_hello1)
<class 'function'>
```

Una función puede definirse en cualquier lugar del código, pero solo estará visible en el bloque en curso o en el que esté incluida, tras la definición de la función:

```
def print_add(a, b):
    def add(a,b):
        return a+b
    print(add(a, b))

>>> print_add(5, 6)
11
```

La función **add** no está definida fuera de la función:

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

Cuidado con no mezclar espacios y tabuladores, Python es estricto con el número de espacios utilizados en la indentación.

Definir una función en el interior de otra función es muy potente y se utiliza para crear decoradores. Un **decorador** es una función que recibe como parámetro una función y que devuelve una función.

Para definir un método, se trata exactamente del mismo proceso. Un **método** es, simplemente, una función definida en el bloque de una clase.

En efecto, la primera utilidad de la clase es la **encapsulación**, es decir, el hecho de contener sus métodos y sus atributos.

Para resumir, una función es una variable de tipo función que se declara de forma particular, pues contiene un bloque de código. Un **atributo** es una variable en una **clase** y un método es una función encapsulada en una clase.

Funciones lambda

Una función anónima es un tipo de función, simple de escribir y por tanto no es necesario declararla en una variable.

Las funciones lambda son una forma de escribir una función anónima que utiliza una sintaxis análoga a la que se conoce en matemáticas.

```
>>> lambda x: x**2
<function <lambda> at 0x101862e18>
```

En Python, es la única forma de escribir una función anónima. Es útil en la programación funcional. En efecto, una función puede estar directamente escrita en la llamada a una función sin necesidad de definirla previamente.

```
>>> list(map(lambda x: x**2, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Si bien las funciones lambda se utilizan con el objetivo de crear funciones anónimas, es posible darles un nombre:

```
>>> f = lambda x: x**2
>>> f(5)
25
>>> list(map(f, range(10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Esta funcionalidad es muy rica, ejemplo:

```
>>> g = lambda x, y: x*y**2
>>> g(4, 2)
16
```

Como conclusión podemos decir que la filosofía de una función lambda es describir una relación entre parámetros y una expresión que los utiliza, de **forma algebraica**.

Clase

La definimos con la palabra clave **class**, y seguido el nombre de la clase, a continuación una lista (ordenada) de sus padres, y a continuación un bloque:

```
>>> class MiClase:
...     pass
```

Una clase puede definirse, también, en cualquier lugar, incluso dentro de otra clase o en una función.

```
>>> class A:
...     class B:
...         pass
```

Por útimo, recordaremos que cualquier variable definida en la clase es un atributo y cualquier función definida en una clase es un método.

```
>>> class MiClase:
...     atributo = 42
...     def metodo(self):
...         pass
```

Instrucción vacía

Para definir una función vacía o una clase vacía (sin instrucciones), es necesario indicarlo con la palabra clave **pass**.

```
>>> def f():
...     pass

>>> class A:
...     pass
```

Cabe destacar que se recomienda encarecidamente documentar siempre las funciones y clases. El docstring es muy recomendado.

Borrado

La palabra clave **del** nos permite eliminar cualquier variable, simplemente:

```
>>> a=5
>>> del a

>>> b=list(range(10))
>>> del b[5] #índice
>>> b
[0, 1, 2, 3, 4, 6, 7, 8, 9]

b=list(range(10))
>>> del b[2:7] #franja  (2:7 intervalo, :1 paso por defecto)
>>> b
[0, 1, 7, 8, 9]

b=list(range(10))
>>> del b[3:7:2] #franja  (2:7 intervalo, :2 paso)
>>> b
[0, 1, 2, 4, 6, 7, 8, 9]
```

```
>>> c = {'a': 'A', 'b': 'B', 'c': 'C'}
>>> del c['b']  # busca la clave b y la elimina
>>> c
{'a': 'A', 'c': 'C'}
```

Devolver el resultado de la función

Una función (o un método) devuelve siempre un valor, y un único valor. Por defecto, se devuelve **None**: (nada)

```
>>> def f():
...     pass
...
>>> print(f())
None
```

El uso de la instrucción **return** permite especificar explícitamente el valor de retorno:

```
>>> def uno():
...     return 1

>>> uno()
1

>>> type(uno())
<class 'int'>

def g():
...     return 1,2,3
...
>>> g()
(1, 2, 3)
```

En realidad, se devuelve un único valor, y se trata de una n-tupla. En efecto, escribir return 1,2,3 equivale exactamente a escribir return (1, 2, 3), la coma es el separador característico de una n-tupla. En la gramática de Python, se interpreta a una secuencia de datos separados por comas como parte de una n-tupla.

Instrucciones condicionales

Definición

Un **bloque condicional** es un bloque de código que se ejecuta si y solamente si la instrucción de control que lo contiene ve cumplida su condición y también puede ubicarse dentro de otro bloque condicional.

Condición

Una condición es la evaluación de una expresión que transforma de forma determinista dicha expresión por uno de los dos valores **True** o **False**.

Las condiciones utilizan con frecuencia operadores de comparación, aunque no es la única forma de crearlas.

Un operador de comparación devuelve simplemente un valor booleano, que es asimismo un objeto.

La **palabra clave is** permite realizar una comparación sobre la identidad del objeto, y no sobre su valor.

Instrucción if

La palabra clave if ejecuta las instrucciones solamente si se verifica una condición:

```
>>> def evaluacion(num):
...     r = 'positivo'
...     if num < 0:
...         r = 'negativo'
...     return r
...
>>> evaluacion(5)
'positivo'
>>> evaluacion(-5)
'negativo'
```

observamos que la escritura de una condición no requiere paréntesis.

Es posible utilizar pass en el bloque condicional aunque a priori no es útil:

```
>>> if True:
...     pass
...
```

Instrucción elif

La instrucción **elif** se utiliza únicamente si el conjunto de instrucciones anteriores se han evaluado a **False**.

Una instrucción **elif** se escribe, obligatoriamente, tras una instrucción **if** y puede haber tantas como sea necesario.

```
>>> if False:
...     print(1)
... elif True:
...     print(2)
... elif True:
...     print(3)
...
2
```

La primera expresión no es verdadera, de modo que el primer bloque no se ejecuta y se sigue leyendo el código. La segunda expresión es verdadera, de modo que se ejecuta el segundo bloque y se detiene la lectura de las siguientes instrucciones condicionales y por tanto sale del bloque condicional.

Instrucción else

Para distinguir cuándo se respeta una condición y cuándo no, es posible realizarlo con dos bloques diferentes:

```
>>> def evaluacion(num):
...     if num < 0:
...         r = 'negativo'
...     else:
...         r = 'positivo'
...     return r
...
>>> evaluacion(-7)
'negativo'
```

Las instrucciones **if** y **else** permiten tratar todos los casos posibles, aunque **elif** aporta una simplificación del algoritmo. He aquí el algoritmo de la sección que describe la instrucción **elif** realizado sin dicha instrucción:

```
>>> if False:
...     print(1)
... else:
...     if True:
...         print(2)
...     else:
...         if True:
...             print(3)
...
2
```

Se aprecia inmediatamente el interés de la instrucción **elif**. En este caso, la tercera instrucción tampoco se valida.

La instrucción else también puede utilizarse de forma complementaria a **elif**, aunque se sitúa, obligatoriamente, en último lugar:

```
>>> def evaluacion(num):
...     if num < 0:
...         return 'negativo'
...     elif num > 0:
...         return 'positivo'
...     else:
...         return 'nulo'
...
>>> evaluacion(0)
'nulo'
```

La instrucción switch no existe en Python, no es amigo de tener varios elementos diferentes para resolver problemáticas idénticas.

La palabra clave **in** se utiliza para verificar si un valor se encuentra en una secuencia:

```
>>> a = 3
>>> a in (2, 3, 5, 7, 11, 13)
True

>>> a == 2 or a == 3 or a == 5 or a == 7 or a == 11 or a == 13
True
```

Cuando la secuencia es muy larga o contiene elementos complejos, las ventajas son evidentes.

Interrupciones del bloque condicional

Cuando se utiliza la instrucción **return** en un bloque condicional, el algoritmo se detiene automáticamente y devuelve el valor solicitado o **None**, sea cual sea la situación.

Rendimiento

Cuando alguna condición se lee a menudo y está compuesta por varias partes con una complejidad similar, resulta importante poner en primer lugar la sección que es más probable que devuelva falso con el objetivo de detener lo antes posible el proceso de evaluación. Si alguna de las partes es más compleja, debería situarse en último lugar, de modo que no tenga que evaluarse salvo si las demás condiciones son verdaderas.

Además, para una condición o parte de una condición compleja que deba calcularse y utilizarse en varias ocasiones, resulta conveniente evaluar de manera previa las instrucciones condicionales de modo que no sea preciso realizar el cálculo varias veces.

Iteraciones

Instrucción for

La instrucción **for** permite repetir una secuencia de instrucciones sobre un conjunto de datos que se le pasa como parámetro.
La palabra clave **in** verifica la pertenencia de un elemento a una secuencia. La combinación de esta con la palabra clave **for** permite iterar sobre el conjunto de elementos de la secuencia.

```
>>> for x in (5, 7, 11, 13):
...     print('%d es un número primo' % x)
...
5 es un número primo
7 es un número primo
11 es un número primo
13 es un número primo
```

Existen muchas formas de iterar mediante estas dos palabras clave, aunque están íntimamente ligadas a los tipos de datos. Existen palabras clave que permiten anticipar la salida de un bucle, las cuales se presentan más abajo.

Instrucción while

La instrucción while sirve para repetir una serie de instrucciones mientras la condición se evalúe como verdadera. En un bucle siempre debemos controlar su salida, ya sea mediante otra condición, instrucción, etc.

Intentaremos siempre generalmente utilizar el bucle con **for** pues de antemano conocemos el número de iteraciones que ejecutarán, ya que **while** necesita un caso bien definido y no es predecible en qué momento la condición se volverá falsa.

Interrupciones de bucle

La instrucción **break** permite terminar la iteración inmediatamente.
Ejemplo: Queremos obtener la potencia de 2 inmediatamente superior a un millón. Se parte de 1 y se multiplica por dos hasta que se cumpla la condición.

```
>>> def f():
...     a = 1
...     while True:
...         a *= 2
...         if a > 1000000:
...             break
...     return a
...
>>> f()
1048576
```

Instrucción return

Cuando se obtiene un resultado, en lugar de utilizar break, es posible devolver el resultado inmediatamente. Es otra forma de terminar una iteración.

```
>>> def f():
...     a = 1
...     while True:
...         a *= 2
...         if a > 1000000:
...             return a
...
>>> f()
1048576
```

Instrucción continue

Permite, simplemente, interrumpir una iteración para pasar a la siguiente.

```
>>> def positivo(l):
...     for a in l:    # espera una lista
...         if a < 0:
...             continue
...         print(a)
…
>>> positivo([1,2])
1
2
```

Instrucción else para bucle for

Es una especificidad de Python. La palabra clave **else** también tiene significado cuando se asocia con la palabra clave **for**.

```
>>> def es_valido(l):
...     for a in l:
...         if a < 0 or a > 20:
...             r = False
...             break
...     else:
...         r = True
...     return r
...
>>>
>>> es_valido([1,2,3])
True
```

La palabra clave else funciona también en combinación con la palabra clave while del mismo modo y con el mismo significado. Ejemplo:

```
>>> n = 2
>>> while n < 10:
...     x = 2
...     while x < n:
...         if n % x == 0:
...             print('%i vale %i * %i' % (n, x, n/x))
...             break
...         x += 1
...     else:
...         print('%i es un número primo' % n)
...     n += 1
...
```

```
2 es un número primo
3 es un número primo
4 vale 2 * 2
5 es un número primo
6 vale 2 * 3
7 es un número primo
8 vale 2 * 4
9 vale 3 * 3
```

La semántica de **else** se define por oposición a **break**.

Generadores

La diferencia entre una secuencia de valores y un generador de valores es que la primera se calcula íntegramente antes de utilizarse, lo cual exige la ocupación en memoria de la lista íntegra y espera su cálculo antes de poder utilizarla. Por el contrario, el generador se contenta con calcular un valor a continuación del otro, devolviéndolos con cada llamada y esperando que se devuelva el control para calcular el valor siguiente.

La principal dificultad de un generador consiste en devolver un valor al algoritmo que lo invoca (el que utiliza el generador), y devolver el control a continuación. Existe solución fácil mediante el uso de la instrucción **yield**:

```
>>> def g(num):
...     for i in range(num):
...         print('Generator %d' % i)
...         yield i
...
>>> g(4)
<generator object g at 0x10195dde0>
>>> gen = g(2)
```

Este código se ejecutará cuando se utilice en un bucle:

```
>>> for i in gen:
...     print('Uso %d' % i)
...
Generator 0
Uso 0
Generator 1
Uso 1
```

Este tipo de generador se llama generador finito, pues realiza el bucle n veces. Dicho generador finaliza obligatoriamente con un **return**, pues un generador es una función y toda función termina de esta manera.

Existe una función **next** que permite invocar al valor siguiente de un generador:

```
>>> def test():
...     yield 1
...     yield 2
...     yield 3
...
>>>
>>>
>>> gen = test()
>>> next(gen)
1
>>> next(gen)
2
```

```
>>> next(gen)
3
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Un generador devuelve una excepción de tipo **Stopiteration** cuando no existen más valores para devolver.

```
# Ejemplo función generadora
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

```
>>> a = my_gen()
>>> next(a)
```

No es posible volver a poner un generador a cero o ir hacia atrás, puesto que no se almacena ningún valor. Por el contrario, es posible crear un nuevo generador para volver a comenzar con él:

```
>>> gen = test()
```

En un generador infinito, el usuario debe gestionar la condición de parada. Ejemplo:

```
>>> def uno():
...     while True:
...         yield 1
...
>>>
```

Y un código que gestiona la detención de la iteración.

```
>>> for a in uno():
...     break
...
```

Yield from

Se utiliza para evitar escribir bucles dentro de otros bucles, ejemplo:

```
>>> def cadena(*iters):
...     for it in iters:
...         yield from it
...
```

Construcciones funcionales

Generadores

Es posible construir un generador en una línea, utilizando las palabras clave **for** e **in** y utilizando paréntesis para delimitarlo:

```
>>> gen = (a**2 for a in range(1000))
```

Para construir un generador infinito:

```
>>> gen = (a**2 for a in generator_infinito())
```

Recorrido de listas

De nuevo, se utilizan las palabras clave for e in, aunque en lugar de utilizar paréntesis se utilizan corchetes:

```
>>> lista = [a**2 for a in range(12)]
>>> print(lista)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

Recorrido de conjuntos

También usamos for e in aunque ahora utilizando llaves:

```
>>> conjunto = {a**2 for a in range(12)}
>>> print(conjunto)
{0, 1, 64, 121, 4, 36, 100, 9, 16, 49, 81, 25}
```

Recorrido de diccionarios

De nuevo for e in junto a llaves, y la presencia de los **dos puntos** indica la diferencia respecto al recorrido de conjuntos.

```
>>> diccionario = {a: a**2 for a in range(12)}
>>> print(diccionario)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81, 10: 100, 11: 121}
```

Gestión de excepciones

Una excepción es un error que puede tener carácter lógico, o más generales del código o despistes, un intento de conexión a un servidor que fracasa ...
Para tener el control, hay que prever los posibles errores que se pueden producir de cara a poder gestionarlos y adaptar la excepción a la situación:

```
>>> def media(*args):
...     return sum(args)/len(args)
...
>>> media()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in media
ZeroDivisionError: division by zero
```

Ajustamos mejor el mensaje de error:

```
>>> def media(*args):
...     if len(args) == 0:
...         raise TypeError('media expected at least 1 arguments, got 0')
...     return sum(args)/len(args)
...
>>> media()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in media
TypeError: media expected at least 1 arguments, got 0
```

Esto se realiza mediante la palabra clave **raise** seguida de una instancia que herede de la clase base Exception.

La información que nos da es el nombre de la excepción, su descripción y la pila de llamadas que contiene las anidaciones del programa y que permite identificar las secciones del código afectadas. Normalmente lo que se hace es capturar el error y llamar algún procedimiento adaptado.

Elevar una excepción

Imaginemos un error al intentar establecer la conexión con un servidor remoto que fracasa, podemos prever el error y expresarlo así: Intenta conectarte al servidor. Si no lo consigues, conéctate a un servidor auxiliar. “Si tampoco lo consigues, envía un correo electrónico al administrador y devuelve un mensaje de error sencillo, conciso y educado al usuario”.

El código que hace el trabajo informa cuando encuentra un problema. Se le denomina **código crítico**.
El hecho de elevar una excepción y de permitir su captura por otro permite articular lo que se conoce como **reparto de responsabilidades**.

Aserciones

La instrucción assert resulta útil para permitir generar excepciones si las condiciones no se cumplen, lo cual resulta perfectamente útil para realizar un control.
Las aserciones pueden utilizarse simplemente para comprobar una expresión:

```
>>> a=1
>>> assert a%2 == 1
>>> a=2
>>> assert a%2 == 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Mientras todo vaya bien, no ocurre nada, pero en caso contrario se genera una excepción de tipo AssertionError cuando una expresión se evalúa como incorrecta.
En este caso es posible pasar a la palabra clave una segunda expresión que se evalúa y sustituye si aparece un problema, que permite precisar un poco mejor el tipo de problema encontrado:

```
>>> a=1
>>> assert a%2 == 1, 'variable a incorrecta'
>>> a=2
>>> assert a%2 == 1, 'variable a incorrecta'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: variable a incorrecta
```

Manera de deshabilitar este modo de depuración es el siguiente:

$ python3 -O nombre_archivo.py (se ignoran las instrucciones **assert**)

Capturar una excepción

Veamos una función que genera un error:

```
>>> def test(num):
...     if num <0:
...         raise ValueError('El número es negativo')
...     return num
...
```

Cuando se produce una excepción, se genera una traza y se envía hasta el origen incluyendo la pila de llamadas:

```
>>> test(-1)

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in test
ValueError: El número es negativo
```

Veamos cómo capturar una excepción:

```
>>> def f(num):
...     try:
...         return test(num)
...     except:
...         return 0
...     finally:
...         print('siempre se ejecuta')
...
>>>
>>> f(1)
siempre se ejecuta
1
>>> f(-1)
siempre se ejecuta
0
```

Vemos que o bien el **bloque try** se ejecuta con éxito, o bien se interrumpe y las instrucciones del **bloque except** se ejecutan a continuación. El bloque de instrucciones incluido en **finally** se ejecuta antes de devolver el resultado, como si se hubiera copiado en ambas secciones del código justo antes del retorno. (evitar la duplicación de código). El bloque finally solamente puede utilizarse con try para gestionar una salida.

Procesamiento de errores personalizado

De cara a gestionar con detalle qué tipo de error se produce durante la ejecución de la secuencia de instrucciones contenida en el bloque **try**, es posible capturar distintos tipos de excepciones:

```
>>> try:
...     pass
... except TypeError:
...     """Procesamiento para este tipo de excepción"""
... except ValueError:
...     """Procesamiento para este tipo de excepción"""
```

```
... except:
...     """Procesamiento para los demás tipos de excepción"""
...
```

Esta solución funciona, aunque no distingue entre un error producido por una u otra de las instrucciones presentes el bloque **try**. Si se debe realizar alguna distinción, es necesario gestionar dos bloques **try** diferentes.

En ocasiones, para generar una excepción, es necesario recuperar el objeto de excepción:

```
>>> try:
...     pass
... except TypeError as e:
...     """Procesamiento para este tipo de excepción"""
... except ValueError as e:
...     """Procesamiento para este tipo de excepción"""
... except Exception as e:     # tipo exception, es decir, la excepción más general
...     """Procesamiento para los demás tipos de excepción"""
...
```

Para profundizar consultar la instrucción **with** que se utiliza con **as**:
Comentar solo su sintaxis propia:

```
with EXPR as VAR:
...     BLOCK
```

esto equivale a:

```
VAR = EXPR
VAR._enter_()
try:
        BLOCK
finally:
        VAR._exit_()
```

Ejemplo típico:

```
>>> with open('ejemplo.txt') as archivo:
…         content = archivo.read()
…
```

De este modo, el archivo siempre se cierra correctamente y sus datos se preservan.

Gestionar módulos

Para diseñar y utilizar nuestros propios módulos se utiliza la instrucción **import**.

Es posible importar todo un módulo:

```
>>> import os
```

Es posible utilizar cualquier función haciendo referencia al módulo.

```
>>> os.walk
<function walk at 0x1019611e0>
```

Es posible importar únicamente lo que vamos a necesitar:

```
>>> from os import walk
>>> walk
<function walk at 0x1019611e0>
```

También es posible utilizar la palabra clave as en los casos anteriores para dar un alias a un módulo, una función, una clase o una constante:

```
>>> from os import walk as w
>>> w
<function walk at 0x1019611e0>
```

Muy útil para los módulos estructurados en profundidad con nombres largos.

Visibilidad de la variable, global, local

La palabra clave global permite publicar una variable que proviene de un contexto local en un contexto global.

En cambio, si definimos normalmente una variable dentro de una función o lo que es lo mismo en su ámbito:

```
>>> def f():
...     a = 1
...
```

O se dice que la variable a está definida en el espacio de nombres local de la función f.
Existe un aislamiento estricto.

En cambio con la palabra clave global, declarándola en la función:

```
def f():
...     global b
...     b=3
...
>>> b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Todavía no existe pues no se ejecutó la función, hacemos la llamada y se comprueba que si accedemos al espacio de nombres de la función f:

```
>>> f()
>>> b
3
```

Existe **nonlocal**, que permite impactar no en el espacio de nombres global sino en el espacio de nombres local inmediatamente superior.

```
>>> def f():
...     a = 0
...     def g():
...         a = 1
...         return a
...     return g() + a
...
>>> a = 10
>>> f()
1
```

```
>>> def f():
...     a = 0
...     def g():
...         nonlocal a
...         a = 1
...         return a
...     return g() + a
...
>>> f()
2
```

Por lo que ambas variables a se comparten en ambos espacios de nombres locales de ambas funciones.

Ejemplo de la palabra clave **global**:

```
>>> def f():
...     global a
...     a = 5
...     print('a de f',a)
...     def g():
...         print('a de g',a)
...         return a
...     return g() + a
...
>>>
>>> f()
a de f 5
a de g 5
10
```

Funciones print, help, eval y exec

La función print imprime por pantalla datos como pueden ser cadenas, números, etc. Separa cada argumento empleando un espacio en blanco por defecto. Como argumentos mencionamos **sep** y **end** en el ejemplo:

```
>>> a='Azul'
>>> b='Verde'
>>> c='Rojo'
>>> print(a,b,c,sep=', ',end='.')
Azul, Verde, Rojo.>>>
```

La función **help** permite mostrar la ayuda de un objeto que se pasa como parámetro, sea una variable, una función, una clase o un módulo.

```
>>> help(print)
```

Help on built-in function print in module builtins:

```
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
```

    flush: whether to forcibly flush the stream.
(END)

Salimos de la ayuda pulsando tecla q.

La función **eval** evalúa una expresión y devuelve un valor. La función **exec** se contenta con ejecutar el código, pues ambas permiten interpretar el contenido de una cadena de caracteres como si se tratara de una línea de instrucciones.

```
>>> a=eval('"Hello World!"')
>>> print(a)
Hello World!
>>> b=exec('"Hello World!"')
>>> print(b)
None
```

En particular, para ejecutar un código contenido en una cadena de caracteres (o en un archivo concreto, o cualquier otro origen), se utiliza **exec**. Para evaluar una expresión y recuperar el valor obtenido en una variable, se utiliza **eval**.

Apéndice

Historia de la computación

La búsqueda de avances tecnológicos a través de las ciencias matemáticas, física, química, etc, ha venido motivada por la competencia económica, es decir, abaratamiento y descenso de costes.
El transistor (un amplificador pequeño y de baja potencia) reemplazó a la válvula de vacío, de mayor consumo y tamaño. Idea (transistor + programa almacenado = ordenador digital)

La miniaturización nace de las exigencias requeridas en los distintos proyectos de misiles y satélites instados por oficinas militares y espaciales.

En 1958, Jack St Clair Kilby concibió el modo de miniaturizar los circuitos fabricando las resistencias, los condensadores y los transistores en el mismo trozo de silicio que incluía las interconexiones en el propio sistema, sentando las bases del circuito integrado.

En 1961 se usaron por vez primera en un ordenador del ejército y en 1962 se incluyeron en la electrónica de los cohetes "minuteman". También se usaron en las naves espaciales del proyecto Apollo.

El chip, su producción es obviamente un largo proceso, pero destaco:
Mediante la fotolitografía, se trasladan a la superficie del semiconductor unos patrones geométricos que permiten definir los elementos constitutivos, sus interconexiones y el aislamiento eléctrico entre ellos.

https://youtu.be/Fxv3JoS1uY8

Eniac primer gran ordenador electrónico.
Ley de Moore.

Índice de términos

Cálculo lambda
El **cálculo lambda** es un sistema formal diseñado para investigar la definición de función, la noción de aplicación de funciones y la recursión.

Cargado
Normalmente lo realiza el propio entorno de ejecución. El archivo ejecutable se lanza en el Sistemas Operativo.

Compilación
El código fuente resultante en lenguaje C++ se compila mediante el software apropiado, obteniendo un código máquina llamado archivo objeto (cuya extensión suele ser .obj). Este código aún no es ejecutable ya que necesita incluir el código máquina relativo a las funciones y módulos que se utilizaban en nuestro código. Estas funciones están incluidas en archivos externos (librerías).

Django
Es un framework para aplicaciones web gratuito y open source, escrito en Python. Es un WEB framework - un conjunto de componentes que te ayudan a desarrollar sitios web más fácil y rápidamente.

Edición
El código se escribe en un editor de texto o en un editor de código preparado para esta acción. El archivo se suele guarda con extensión .cpp (también en cxx, c++ o cc).

Enlazado
El código objeto se une al código compilado de las librerías y módulos invocados por el código anterior. El resultado es un archivo ejecutable (extensión .exe en Windows)

Preprocesado
Antes de compilar el código, el preprocesador lee las instrucciones de preprocesador y las convierte al código fuente equivalente.

PEP-8
Estilo de codificación que debe utilizarse cuando se desarrolla con Python.
Http://www.python.org/dev/peps/pep-0008/

PEP-7
Recomendaciones sobre la redacción de código C de Python.

PEP-257
Explica qué es un docstring, su utilidad, … convenciones relativas.

Twisted
Es un framework de red para programación dirigida por eventos escrito en Python y licenciado bajo la licencia MIT.
Twisted proporciona soporte para varias arquitecturas (TCP, UDP, SSL/TLS, IP Multicast, Unix domain sockets), un gran número de protocolos (incluidos HTTP, XMPP, NNTP, IMAP, SSH, IRC, FTP), y mucho más.

Unidades fundamentales del tiempo de ejecución
Un **ciclo de instrucción** (también llamado ciclo de *fetch-and-execute* o ciclo de *fetch-decode-execute* en inglés) es el período que tarda la unidad central de proceso (CPU) en ejecutar una instrucción de lenguaje máquina.

Referencias y bibliografía

Python 3 Al descubierto
Arturo Fernández Montero

Python 3 Los fundamentos del lenguaje
Sébastien Chazallet (Eni ediciones)