UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN

Manual básico del lenguaje de programación Python: aplicaciones en las áreas químico-biológicas y farmacéuticas para no informáticos.

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADA EN FARMACIA

P R E S E N T A :

MARÍA JOSÉ FERNÁNDEZ HERNÁNDEZ

DIRECTOR DE TESIS:
Dr. SALVADOR FONSECA CORONADO

CUAUTITLÁN IZCALLI, ESTADO DE MÉXICO 2021

VNIVERSIDAD NACIONAL AVÓNOMA DE MEXICO

FACULTAD DE ESTUDIOS SUPERIORES CUAUTITLÁN
SECRETARÍA GENERAL
DEPARTAMENTO DE TITULACIÓN

ASUNTO: VOTO APROBATORIO

M. en C. JORGE ALFREDO CUÉLLAR ORDAZ
DIRECTOR DE LA FES CUAUTITLAN
PRESENTE

ATN: I.A. LAURA MARGARITA CORTAZAR FIGUEROA
Jefa del Departamento de Titulación
de la FES Cuautitlán.

Con base en el Reglamento General de Exámenes, y la Dirección de la Facultad, nos permitimos comunicar a usted que revisamos la: **Opción de titulación**

Manual básico del lenguaje de programación Python: aplicaciones en las áreas químico-biológicas y farmacéuticas para no informáticos.

Que presenta la pasante: **María José Fernández Hernández**
Con número de cuenta: **417075776** para obtener el Título de: **Licenciada en Farmacia**

Considerando que dicho trabajo reúne los requisitos necesarios para ser discutido en el **EXAMEN PROFESIONAL** correspondiente, otorgamos nuestro **VOTO APROBATORIO**.

ATENTAMENTE
"POR MI RAZA HABLARÁ EL ESPÍRITU"
Cuautitlán Izcalli, Méx. a 04 de Junio de 2021.

PROFESORES QUE INTEGRAN EL JURADO

	NOMBRE	FIRMA
PRESIDENTE	Dra. Sandra Diaz Barriga Arceo	
VOCAL	Dr. Salvador Fonseca Coronado	
SECRETARIO	Dr. Victor Hugo Vázquez Valadez	
1er. SUPLENTE	Dra. Gabriela Rodríguez Patiño	
2do. SUPLENTE	L.B.D. Larisa Andrea González Salcedo	

NOTA: los sinodales suplentes están obligados a presentarse el día y hora del Examen Profesional.

LMCF/javg

Dedicatorias

A MI PAPILLA

Porque a pesar de todas nuestras dificultades siempre vas a ser mi papá y siempre te voy a amar. Ojalá pudieras vivir por siempre para estar junto a mi dándome lata. Te amo mucho y espero que estés orgulloso de mi y de todos los pequeños o grandes logros que pueda tener. No pude pedir otro papá en el mundo.

A INESITA

Eres la mujer a la que más amo en esta vida, quiero que seas fuerte y sigas creciendo para que sigamos teniendo momentos tan hermosos como hasta ahora. No cambiaría nada entre tú y yo, siempre vas a ser mi bebé, siempre voy a estar ahí para ti, aunque esté en otra galaxia.

A MI WILLY

Has sido parte de mi vida desde hace muchos años y ahora lo serás hasta el final. Eres de mis más grandes pilares y lo sabes, tengo infinidad de cosas que agradecerte y estoy feliz de poder empezar con esto. Te amo mucho y también es gracias a ti que he podido llegar hasta donde estoy, muero de ganas de poder saber qué nos espera juntos.

A MI ABU

Eres el ser más dulce que he conocido, me hace muy feliz poder pasar los fines de semana a tu lado, recibiendo tus consejos y escuchando las anécdotas de tu vida. Siempre vas a poder contar conmigo, así como yo he contado contigo. Nunca podré terminar de agradecerte por cuidar a mi hermanita. Quédate con nosotros mucho tiempo más para que puedas vernos a todos crecer. Te amo.

A MIS ABUELOS ROÑAS Y CONEJO Y A MI MADRINA

Aunque ya no estemos juntos seguirán siendo mis abuelos y mi Mayi. Su apoyo llegó en un momento crucial en mi vida y de no ser por ustedes me hubiese costado mucho más trabajo llegar hasta donde estoy ahora. Este logro también es suyo. Los extraño, ojalá pueda verlos otra vez.

A MI MADRE

Aunque hace más de 10 años que no estás aquí y sé que nunca vas a leer esto, quiero darte las gracias por todo lo que pasó, de no haber sido así, no sé dónde estaría ahora. Un beso hasta el cielo.

Agradecimientos

A la FESC y la UNAM por brindarme la oportunidad y los recursos necesarios para desarrollarme en una carrera tan hermosa y diversa como lo es la Licenciatura en Farmacia, siempre me voy a esforzar por poner en alto el nombre de esta grandiosa universidad.

A todos los profesores involucrados en mi formación, en especial a las profesoras Rebollar, Gaby Patiño, Piñón, Raquel, Lupita Álvarez, Paty Zúñiga, Tais y a los profesores Roberto, Oropeza, Garduño, Rodolfo y Juanjo por dejarme enseñanzas académicas y no académicas.

A mis amigos Rafa (y a su bolsa de dulces), Alejandro, Dona, Fer, Arllet, Fátima, Yoseline, Esme y Tania por haber formado parte de esta etapa en algún o en diversos puntos de esta, siempre van a formar parte de mí. Rafita, sabes que eres el más especial para mí.

A mi amigo Oscar, porque, aunque estés lejos siempre has estado ahí para mí y eso tiene mucho valor, te quiero.

Al todo el equipo del LQM: Pablito, Juan y Caro (aunque ya no estén ahí), Maritza, Pablón, Serda, Eli, Tlotzin y sobre todo a Vic Valadez y Dr. Ángeles por abrirme las puertas del laboratorio y haber sido de gran apoyo para mí durante mi desarrollo en la facultad. Si me dieran a elegir una etapa de la universidad a la cual volver, sin duda elegiría estar con ustedes.

Al team Crash Course en especial a Chava, Paúl y Gil por recibirme en su equipo, compartir momentos tan agradables, brindarme conocimientos tan valiosos e introducirme a este mundo moderno de la ciencia. Me encantaría compartir proyectos con ustedes.

A mi familia sanguínea y política por escucharme y apoyarme de todas las formas posibles para lograr mis metas, empezando por culminar mi carrera. Los quiero mucho a todos. Gracias a Adri, Calla, Mafer, Tita, Güera, Mayi, Carmelita, Champi, Nicky, Sofi, Richie, a mi suegra, Yasna, Lili, Tatán y Ale.

A mis sinodales la Dra. Sandra Díaz y Larisa González por su tiempo y apoyo en la corrección de este trabajo.

“Las palabras nunca alcanzan cuando lo que hay que decir desborda el alma”.

Índice general

1. Resumen

El manejo de los lenguajes de la programación es cada vez más demandado por los campos laborales de todas las áreas profesionales. Para las ciencias químico-biológicas y de la salud las habilidades de programación son indispensables debido a los avances de la ciencia misma, por lo que, su implementación deberá ser paulatina y desde niveles iniciales como lo es una licenciatura (o antes) donde se incluya una puntualización a sus debidas áreas de conocimiento con ejemplos que tienen una aplicación real y de utilidad.

Por lo anterior, el objetivo de este trabajo fue Integrar un manual interactivo enfocado en el aprendizaje del lenguaje Python 3, con conceptos básicos y contenido teórico-práctico, con un enfoque hacia la resolución de problemas en las áreas químico-biológicas y farmacéuticas. Este manual puede ser implementado dentro de los planes de estudio de diversas carreras, así como también puede ser usado por cuenta propia y fomentar el autoaprendizaje y adquisición de herramientas útiles para la actualidad. Python es uno de los lenguajes que ha ido ascendiendo en su nivel de importancia en todas las áreas anteriores, siendo uno de los más valorados hoy en día. Además, el manual aquí presentado se elaboró para ser empleado como usuario del sistema operativo de Windows.

El resultado final consta de 19 capítulos que abarcan desde terminología básica hasta la ejemplificación de creación de un programa que concentra, almacena y maneja la información recopilada de diversos medios acerca de principios activos y excipientes con su respectiva información fundamental para formulación de medicamentos veterinarios y humanos puede ser de utilidad para un inicio integral en el aprendizaje del lenguaje de programación de Python 3.

2. Introducción

La programación es en la actualidad indispensable para los profesionales de la salud (médicos, inmunólogos, farmacéuticos, biólogos, veterinarios, biotecnólogos, etc.). El aprendizaje de lenguajes de programación y su implementación en las áreas de las ciencias químico-biológicas y farmacéuticas ha sido paulatino y no está muy difundido a nivel licenciatura; el uso de estas herramientas tiene un alto impacto en la optimización de recursos y tiempo y en el manejo de los grandes volúmenes datos que generan los equipos de alto rendimiento en la actualidad (Schubert *et al.,* 2016).

Python es un lenguaje de programación de muy alto nivel, de uso comercial y académico generalizado. Presenta una infinidad de ventajas: es de código abierto, simplificado, rápido, flexible, elegante, ordenado, limpio, portable y posee una comunidad grande para la interacción y resolución de dudas. Python puede interactuar con código optimizado escrito en diferentes lenguajes de programación, y junto con el proyecto Numerical Python numpy, constituye una buena elección para la programación científica (Bassi, 2017).

Python y sus diversos paquetes de recursos como Biopython son herramientas de código abierto disponibles gratuitamente para los principales sistemas operativos (Cock, *et al.,* 2009). La informática ha revolucionado las ciencias biológicas durante las últimas décadas, de modo que prácticamente toda la investigación contemporánea en biología molecular, bioquímica y otras biociencias utiliza programas informáticos (Brea, 2019). Los avances computacionales se han producido en muchos frentes, impulsados por desarrollos fundamentales en hardware, software y algoritmos, así como análisis y manejo de datos de volúmenes muy grandes (Cañeda, Ramos y Guerrero, 2015). Estos avances han influido e incluso generado, una extensa variedad de campos de la biociencia, incluida la evolución molecular y la bioinformática dentro de la cual los estudios de genoma, proteoma, transcriptoma y metaboloma tienen un gran auge, dando como resultado que gran parte de la biología posgenómica se está integrando cada vez más como una disciplina de biología computacional. La capacidad de diseñar y escribir programas de computadora, mejor

conocido como pensamiento computacional, es una de las habilidades más indispensables que puede desarrollar un investigador moderno (Zapata-Ros, 2015).

Las ciencias farmacéuticas se sustentan en la actualidad en el uso de los lenguajes de programación para sus actividades de investigación y logística, por ejemplo: los químicos computacionales desarrollan diversos algoritmos y modelos de simulación para predecir propiedades moleculares y evaluar la probabilidad de que una molécula sea estable en su tránsito farmacológico y farmacodinámico; el desarrollo de estos modelos computacionales permiten optimizar los estudios y perfilar a los mejores candidatos. El uso de los lenguajes de programación puede ser llevado a otras instancias como el control de calidad, diseños de experimentos, optimización de formulaciones, control de inventarios, farmacovigilancia, productividad, movilidad de producto, etc. (Mitra, 2020).

El objetivo de esta tesis es presentar las bases y conceptos fundamentales para la implementación de flujos de trabajo basados en Python que logre poner al alcance de los profesionales de las áreas químico-biológicas y farmacéuticas, que no necesariamente están relacionados con el área informática, una fuente de información, lógica, coherente y con ejemplos de aplicaciones para su utilización en las áreas químico-biológicas y farmacéuticas.

El trabajo presentará las estrategias de instalación de las suites y programas relacionados, los comandos básicos y las expresiones regulares de mayor uso y aplicación y diversos ejemplos de aplicación en análisis de secuencias genómicas (Polimorfismos, SARS-CoV-2, patrones de herencia, entre otros), así como de aplicación de la resolución de procesos durante el desarrollo farmacéutico; en este último caso se presentará el desarrollo de un software de utilidad en la agilización para la búsqueda de información de excipientes y activos farmacéuticos durante la fase de pre-formulación.

La pre-formulación consiste en la caracterización fisicoquímica del principio activo y de los excipientes (Del Río, 2002) la cual se realiza primero por revisión bibliográfica y,

posteriormente, pueden realizarse pruebas específicas marcadas en bibliografía tal como las farmacopeas internacionales y pruebas directas dentro de una formulación para conocer el comportamiento real. La obtención de información acerca de estos activos o excipientes en ocasiones es complicada, debido a la dispersión de esta; no siempre se encuentra fácilmente disponible (ya sea en la web o en lugares físicos como las bibliotecas universitarias) debido a sus altos costos y, la información disponible, muchas veces puede ser insuficiente. Con el desarrollo de un software que logre agrupar la mayor cantidad de información y se presente de manera amigable y gratuita a los usuarios, se busca tener un impacto positivo en el sector farmacéutico desde niveles de enseñanza hasta la industria e investigación.

3. Objetivos

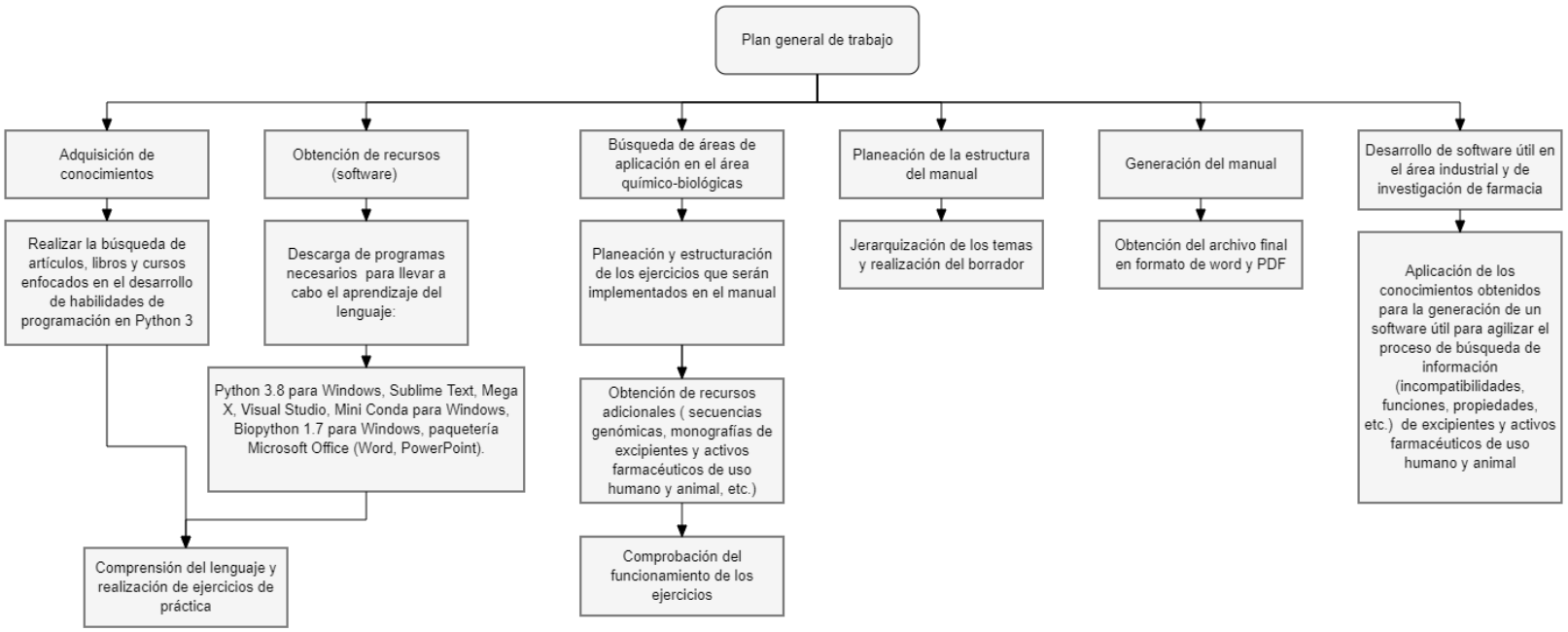
3.1 Objetivo general

Demostrar la importancia de la implementación del lenguaje de programación Python 3 en el área de las ciencias químico-biológicas y farmacéuticas para la resolución de problemas emergentes, por medio del desarrollo de un manual interactivo con ejemplos ilustrativos y un software auxiliar en el área de desarrollo farmacéutico humano y veterinario.

3.2 Objetivos específicos

- Identificar las áreas de oportunidad para la implementación de herramientas y recursos informáticos para la resolución de problemas biológicos y farmacéuticos, generadas a partir del uso del lenguaje de programación Python 3.
- Recopilar todos los recursos informáticos y de software (secuencias genómicas diversas, monografías de excipientes, activos farmacéuticos de uso humano y veterinario, etc.) para la implementación de los ejemplos.
- Integrar un manual interactivo enfocado en el aprendizaje del lenguaje Python 3, con conceptos básicos y contenido teórico-práctico, con un enfoque hacia la resolución de problemas en las áreas químico-biológicas y farmacéuticas.
- Diseñar, estructurar y poner en función un software de utilidad en la agilización para la búsqueda de información de excipientes y activos farmacéuticos de uso humano y veterinario basado en el lenguaje de programación Python 3.

4. Metodología



5. Resultados (Manual)

ÍNDICE DEL MANUAL

Introducción

Este manual tiene como objetivo el acercar a profesionales o estudiantes en formación de las áreas químico-biológicas y de la salud a un área que es cada vez más necesaria y común en todos los trabajos: la informática. Existen diversos lenguajes de programación de utilidad para la industria, la medicina y los laboratorios. Aunque los lenguajes de programación pueden adaptarse a los propósitos que se desee, algunos son más fáciles de comprender y emplear que otros, además, hay lenguajes que poseen más recursos y herramientas adicionales en ciertas áreas, lo que agilizará los proyectos. El desarrollo de los ejemplos y ejercicios de este manual permitirá al usuario integrarse al mundo de la programación en Python con un enfoque dinámico: con aplicaciones a la vida cotidiana como realizar sumas u ordenar datos, hasta ideas útiles para desarrollo de programas asociados con experimentos genéticos o la administración de una farmacia. La dificultad de los temas se ordenó de forma ascendente. Sin embargo, algunos temas son complementarios entre sí y otros no presentarán el mismo grado de utilidad a lo largo de los ejercicios (lo cual no significa que no posean importancia durante el aprendizaje del lenguaje). Dentro del manual se encuentran referencias bibliográficas y recursos adicionales como sitios web donde el usuario puede encontrar apoyo a cualquier dificultad presentada durante el desarrollo de los ejercicios.

¿Por qué programar con Python?

A diferencia de otros lenguajes que manejan objetos, Python permitirá programar de una manera clásica (lo que significa que el usuario crea un programa donde él mismo maneja los datos que entran, como serán procesados, y la respuesta que puede obtenerse) donde el propio lenguaje ofrece una guía de estilo de Programación Orientada a los Objetos (POO/OPP) los cuales son considerados, dentro de Python, como cualquier cosa: una lista es un objeto, una variable es un objeto, una función lo es también y muchas otras cosas como los paquetes y los módulos son considerados un objeto en este lenguaje (Bassi, 2017). Aunque esto facilita comenzar a programar, puede limitar al programador a no aprovechar todas las posibilidades que ofrece Python. Los objetos pueden ser manipulados de múltiples maneras y en conjunto (como manipular todos los datos de una lista al mismo tiempo)

debido a que Python posee métodos para agrupar a los objetos (como clases y/o subclases) y facilitar el trabajo. Lo mismo es cierto para otros tipos de datos que se incluyen en Python. Se puede usar una clase para definir un tipo de datos u objetos. Aunque los tipos de datos incluidos en Python son múltiples y variados, su capacidad para incluir todas las necesidades de modelado de información es limitada (Lee, 2019).

Uno de los objetivos de la programación es representar el mundo real (Zapata-Ros, 2015): se puede usar un diccionario para representar una tabla de traducción entre nucleótidos y aminoácidos, una cadena de texto para representar una secuencia de ADN o una tupla para representar las coordenadas espaciales de un átomo en una proteína. Pero ¿qué tipo de datos se utilizan para representar un estado metabólico de una célula?, ¿los diferentes dominios en una proteína?, ¿el resultado de una corrida en BLAST?, ¿qué pasa con un ecosistema?, ¿se puede emplear para un hospital o farmacia?

Existe la necesidad de ser capaces de definir nuestros propios tipos de datos y objetos, para poder modelar cualquier sistema, ya sea biológico o de cualquier otro tipo. Aunque las funciones son útiles para agrupar el código en módulos, no están diseñadas para cumplir esta función. Las funciones no pueden almacenar estados de los objetos, ya que, los valores de las variables solo pueden estar vigentes mientras se ejecuta la función. Otros lenguajes tienen sus tipos de datos personalizados, pero no tienen la misma flexibilidad que los objetos de lenguajes basados en OOP como Java, C ++ o Python (Brea, 2019). Solo los objetos tienen suficiente capacidad de ser modelados para cualquier tipo de sistema y sus posibles relaciones con otros (Bassi, 2017).

Algunas ventajas sobre programar con Python:

Open source (código o fuente abiertos)

Python tiene una licencia de código abierto liberal que lo hace de libre uso y distribución, incluso para uso comercial.

Simplificado y rápido

Este lenguaje simplifica mucho la programación. Hace que adaptarse a un modo de lenguaje de programación sea mucho más sencillo, ya que, Python propone un patrón fácil de digerir. Es un gran lenguaje para crear programas sencillos basados en una lista de comandos o guion (scripting), si se requiere algo rápido en el sentido de la ejecución del lenguaje, con unas cuantas líneas ya está resuelto el problema. Además, para simplificar el trabajo, Python ya incluye librerías de paquetes de datos que están inmediatamente disponibles para el uso libre del usuario y sin necesidad de descargarlos previamente (Bassi, 2017). Con Python se puede en pocas líneas: leer archivos XML (eXtensible Markup Language) que está siempre presente en sitios web, extraer archivos comprimidos (zip), generar listas de correos, manejar archivos generales, leer datos, adquirir datos desde servidores de la web, acceder a enlaces y mucho más.

Elegante y flexible

El lenguaje brinda muchas herramientas, si se necesitan varios tipos de datos, no hace falta que se especifiquen o declaren todos los tipos de estos al crear objetos como listas. Es un lenguaje tan flexible que hay menos detalles pequeños en los que se debe de preocupar el usuario a diferencia de otros lenguajes de programación.

Programación sana y productiva

Programar en Python se convierte en un estilo muy sano de programar: es sencillo de aprender, direccionado a las reglas perfectas (sugeridas) que son fáciles de llevar a cabo, incrementa nuestro nivel de conocimientos en programación, el uso de las líneas resulta cómodo, etc. (Bassi, 2017). Además, es un lenguaje que fue hecho con la productividad en mente, es decir, Python hace ser al usuario más productivo, lo que nos permite entregar en los tiempos especificados un trabajo.

Disponibilidad de módulos de terceros

Además de los módulos y paquetes desarrollados especialmente para programar en Python existe la posibilidad de crear recursos por cuenta propia y subirlos a la web. Además, se puede hacer uso de dichos recursos creados por la comunidad Python, los cuales tienen una amplia diversidad de usos y de temas, por ejemplo: 2/3D plotting, generación de archivos PDF, análisis bioinformático, animación, desarrollo de videojuegos, desarrollo e implementación de software auxiliar para administración de almacenes farmacéuticos, hospitales, inmunología y mucho más.

Ordenado y limpio

El orden que mantiene Python es de lo que más agrada a usuarios, es muy legible, cualquier otro programador lo puede leer y trabajar sobre el programa escrito en Python. Los módulos están bien organizados, a diferencia de otros lenguajes y diversas características más.

Portable

Es un lenguaje muy portable (puede usarse ya sea en Mac, Linux o Windows) en comparación con otros lenguajes. La filosofía de baterías incluidas es poseer las librerías que más se necesitan al día a día de programación y al estar éstas dentro del interprete, no se tiene la necesidad de instalarlas adicionalmente como en otros lenguajes (Brea, 2019).

Comunidad

Algo muy importante para el desarrollo de un lenguaje es la comunidad. La misma comunidad de Python cuida el lenguaje, ofrece cursos y lecturas complementarias y, casi todas las actualizaciones se hacen de manera democrática (Lee, 2019). Para poder visitar las diversas comunidades latinas se pueden hacer introduciendo los siguientes enlaces en el buscador/navegador:

Comunidad Python Colombia:
https://www.python.org.co/

Comunidad Python Chile:

https://pythonchile.cl/comunidad/

Comunidad Python Argentina:

http://www.python.org.ar/

Comunidad Python Brasil (portugués):

https://python.org.br/

Comunidad Python Perú:

https://www.meetup.com/es/pythonperu/

Y existen foros grandes de la comunidad en otros idiomas como:

- Python Forum (Inglés).
- Python-Forum.de (Alemán).
- /r/learnpython (Inglés).

Otra comunidad donde es posible obtener ayuda de manera eficiente y rápida (además de que hay muchas propuestas interesantes a los problemas) es en Stack Overflow. En este sitio no existen únicamente errores relacionados con Python, sino con cualquier tema relacionado con la programación. Es posible obtener respuesta a dudas o problemas en cuestión de minutos. Los enlaces son el siguiente:

- https://stackoverflow.com/ (Inglés)
- https://es.stackoverflow.com/ (Español)

Python en las ciencias químico-biológicas

Python es muy útil para los profesionales informáticos, diseñadores, etc., pero ¿tiene alguna utilidad en la ciencias químico-biológicas? Para los profesionales de la salud tales como médicos, inmunólogos, farmacéuticos, biólogos, bioquímicos, veterinarios, biotecnólogos, etc., también puede resultar muy útil el aprender a programar e implementar estas herramientas en tareas que pueden ser desde coloquiales hasta para situaciones específicas fuera de lo común.

La informática ha revolucionado las ciencias biológicas durante las últimas décadas, de modo que, prácticamente, toda la investigación contemporánea en biología molecular, bioquímica y otras biociencias utiliza programas informáticos (Bassi, 2017). Además, poseer conocimiento en programación (sobre todo en Python y otros lenguajes más) puede ser de mucha utilidad para las empresas hoy en día. Los avances computacionales se han producido en muchos frentes, impulsados por desarrollos fundamentales en hardware, software y algoritmos, así como análisis y manejo de datos de volúmenes muy grandes (big data). Estos avances han influido e incluso generado, una extensa variedad de campos de la biociencia, incluida la evolución molecular y la bioinformática; estudios experimentales de genoma, proteoma, transcriptoma y metaboloma; genómica estructural y simulaciones atomísticas de conjuntos moleculares a escala celular tan grandes como ribosomas y virus intactos. En resumen, gran parte de la biología posgenómica se está convirtiendo cada vez más en una forma de biología computacional. La capacidad de diseñar y escribir programas de computadora es una de las habilidades más indispensables que puede desarrollar un investigador moderno.

Python se ha convertido en un lenguaje de programación popular en las biociencias. En gran parte porque se considera un lenguaje fácilmente accesible por todas sus características; es expresivo y adecuado para la programación orientada a objetos, así como para otros paradigmas modernos; y las numerosas bibliotecas disponibles y los conjuntos de herramientas de terceros amplían la funcionalidad del lenguaje central a prácticamente todos los dominios biológicos (análisis de secuencia y estructura, filogenómica, sistemas de gestión del flujo de trabajo, etc.) (Brea, 2019).

Algunos ejemplos interesantes de Python en las ciencias biológicas pueden ser:

- En la industria farmacéutica para el uso de química computacional como apoyo en el diseño y descubrimiento de nuevos fármacos.
- Para el manejo de una farmacia y su stock en línea.

- Un marco de inmunoinformática de código abierto que ofrece un acceso fácil y unificado a métodos para la predicción de epítopos y otras aplicaciones inmunoinformáticas como FRED2 (Schubert *et al.,* 2016).

Capítulo I. Descarga e instalación de programas y archivos esenciales

Descarga de Python 3

Python viene por defecto en sistemas operativos como Mac OS X y Linux, pero para Windows es necesario descargarlo desde el sitio web oficial: https://www.python.org/ donde se verá la siguiente interfaz:

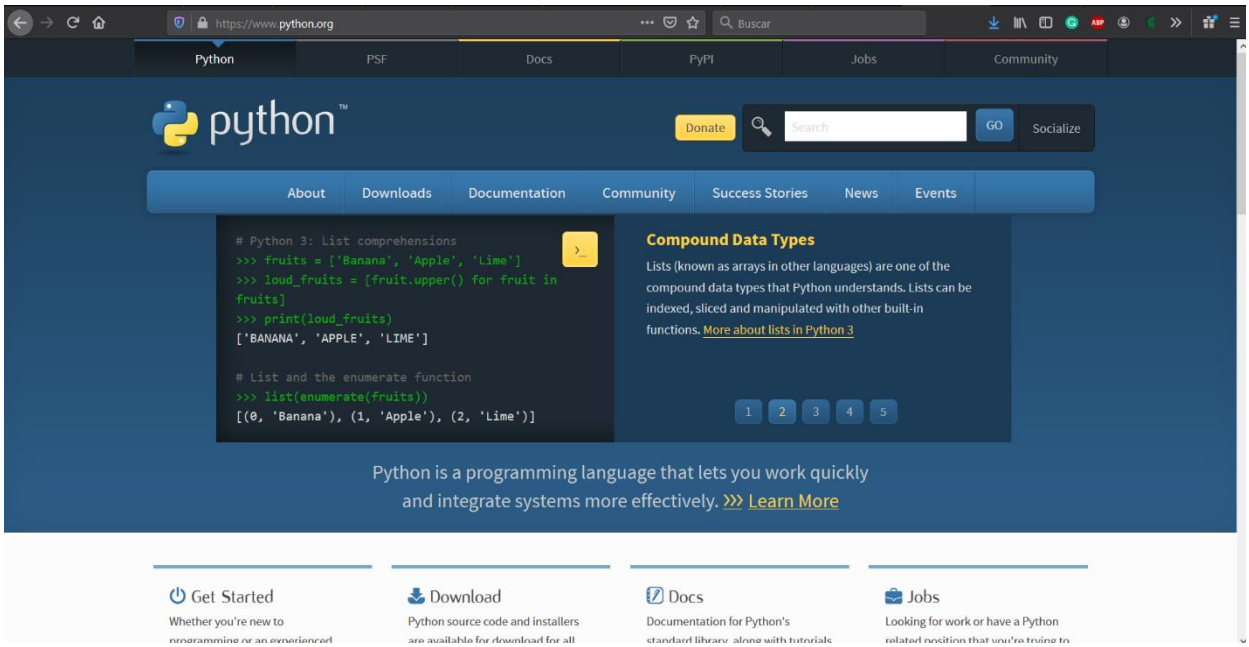


ILUSTRACIÓN 1. PÁGINA OFICIAL DE PYTHON

A continuación, colocar el puntero sobre la pestaña de descargas, dando clic en donde dice "Windows":

ILUSTRACIÓN 2. ZONA DE DESCARGAS DE PYTHON

Aparecerá una lista que contiene las diferentes versiones de Python 2 o Python 3 que están disponibles. Actualmente la versión más estable y empleada de Python 2 es la versión 2.7 y la versión más estable de Python 3 es la versión 3.6. Se está trabajando en versiones de prueba Python 3.9 y se encuentra en desarrollo la versión 3.10. Este manual se desarrolló empleando Python 3.6 y 3.8.3 y se recomienda emplearla para seguir los ejercicios que aquí se plantean. Sin embargo, es posible adaptarlos a versiones como la 2.7 u otras versiones de Python 3 si se realizan ciertas modificaciones a los códigos sugeridos en este manual. Se debe considerar que si se poseen versiones de Windows XP o inferiores pueden no estar diponibles ciertas versiones de Python, lo cual se indica debajo de cada una de las versiones listadas.

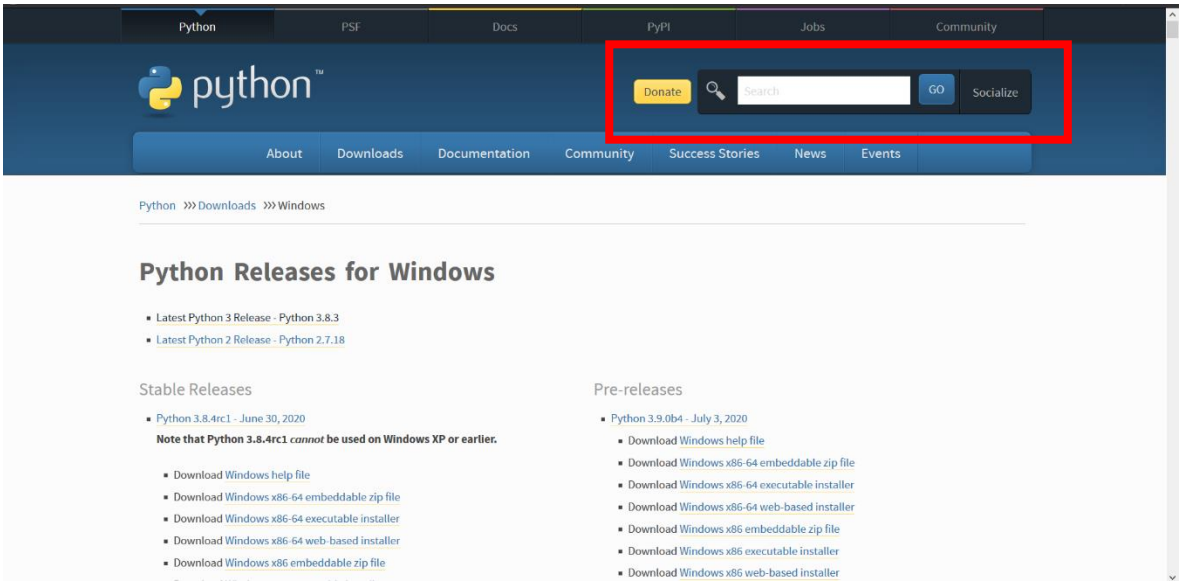


ILUSTRACIÓN 3. LISTA DE VERSIONES DISPONIBLES DE PYTHON PARA WINDOWS

En la Ilustración 3 se puede observar un ícono de búsqueda, en el que se colocará la versión de Python que se desee instalar. Al elegir la versión y dar clic sobre ésta, se desplegará información útil como la fecha de la última actualización, mejoras añadidas, corrección de errores, así como las Propuestas de Mejora de Python (PEP, por sus siglas en inglés: *Python Enhancement Proposals*) los cuales se pueden consultar para tener mejor información acerca de cómo utilizar la versión que se va a descargar o que se posee.

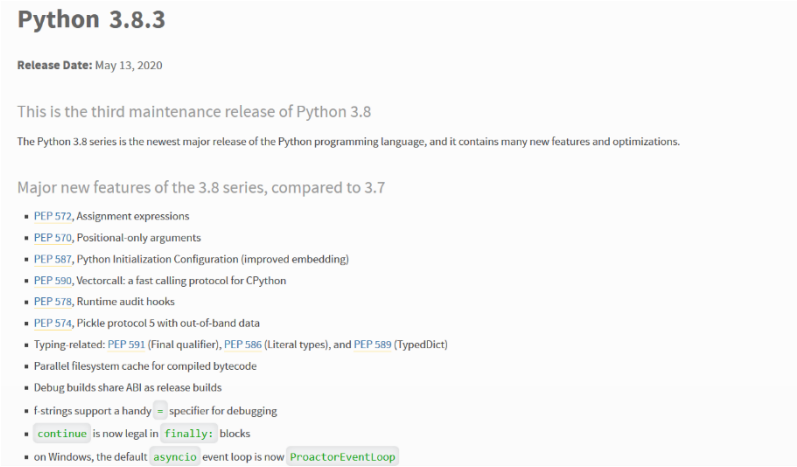


ILUSTRACIÓN 4.INFORMACIÓN SOBRE LA VERSIÓN MÁS ACTUAL DE PYTHON PARA WINDOWS

Se deberá desplazar el cursor sobre la barra lateral para llegar a la zona de descargas. Se elige el archivo de acuerdo con el sistema operativo disponible en nuestro ordenador, que (en este caso) es Windows de 64 bits: Windows x86-64 executable installer. Al dar clic aparecerá una ventana donde se selecciona “Guardar archivo” o una opción similar en su navegador.

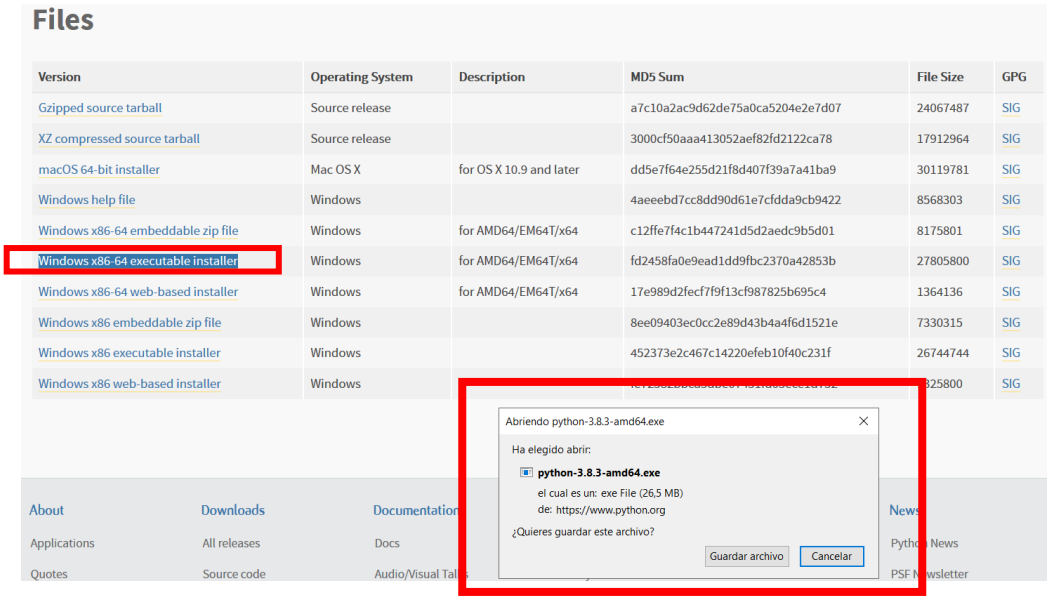


ILUSTRACIÓN 5.DESCARGA DE PYTHON EN EL NAVEGADOR

Para el navegador Mozilla Firefox y Opera, la zona de descargas se muestra en la parte superior derecha como una flecha con dirección hacia abajo y al culminar la descarga se

torna de color azul. Dar clic en el ícono de descargas (Ilustración 6) y seleccionar el archivo de Python que se acaba de descargar.

ILUSTRACIÓN 6. ÍCONO DE DESCARGAS EN MOZILLA FIREFOX

Aparecerá la siguiente ventana con la cual se llevará a cabo una instalación semiautomática de Python. Es importante marcar la casilla de "Añadir al PATH" y seleccionar la opción de "Instalar ahora".

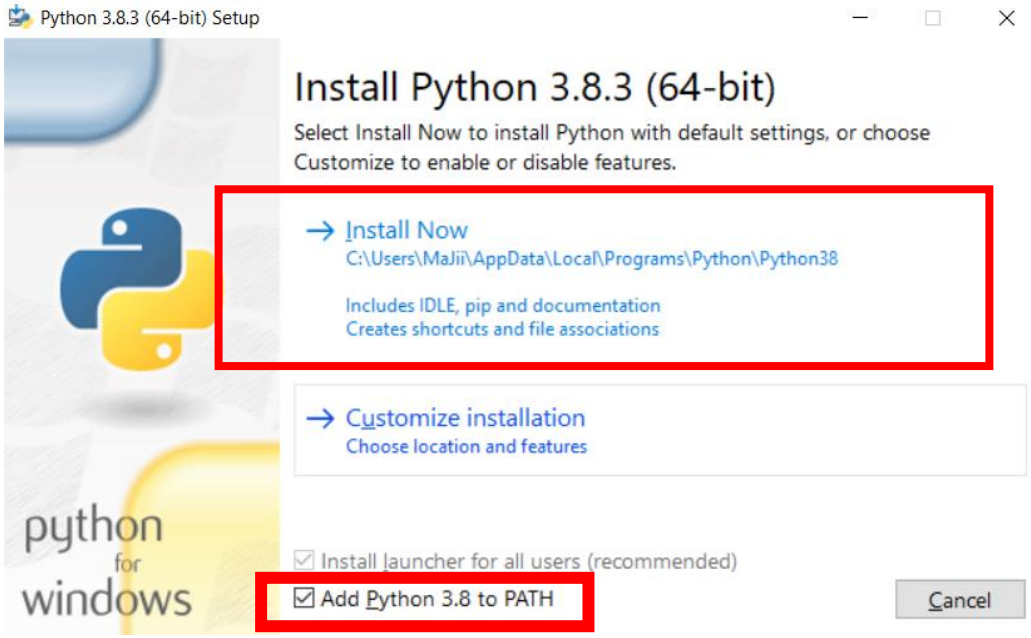


ILUSTRACIÓN 7. COMIENZO DE INSTALACIÓN DE PYTHON Y ADICIÓN AL PATH

Comenzará la instalación de Python.

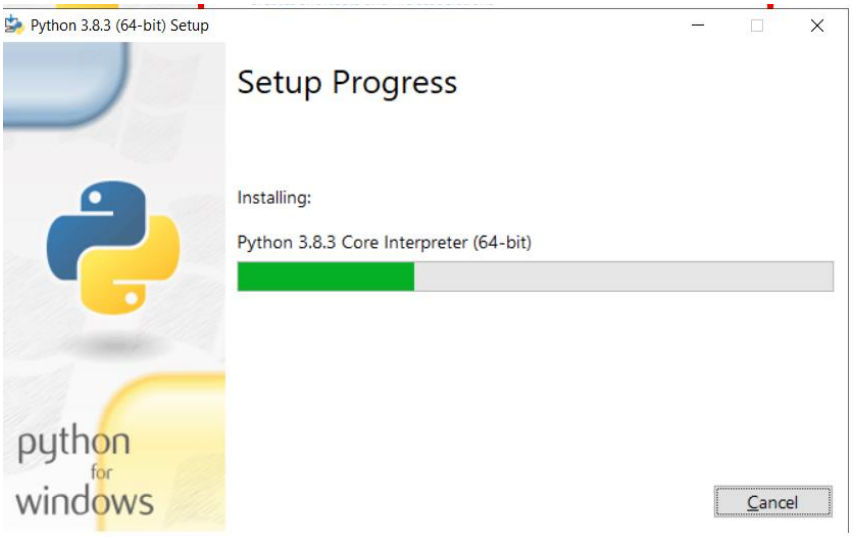


ILUSTRACIÓN 8. PROCESO DE INSTALACIÓN EN CURSO

Al finalizar, se otorgará la opción de deshabilitar la opción de limitación de caracteres en el PATH, esta selección es opcional. Una vez finalizada la selección, dar clic al botón de “Cerrar”. Se ha finalizado con la instalación de Python. Posteriormente se verificará esto con ayuda del símbolo del sistema y se corroborará una vez más dentro del programa “Sublime Text 3”.

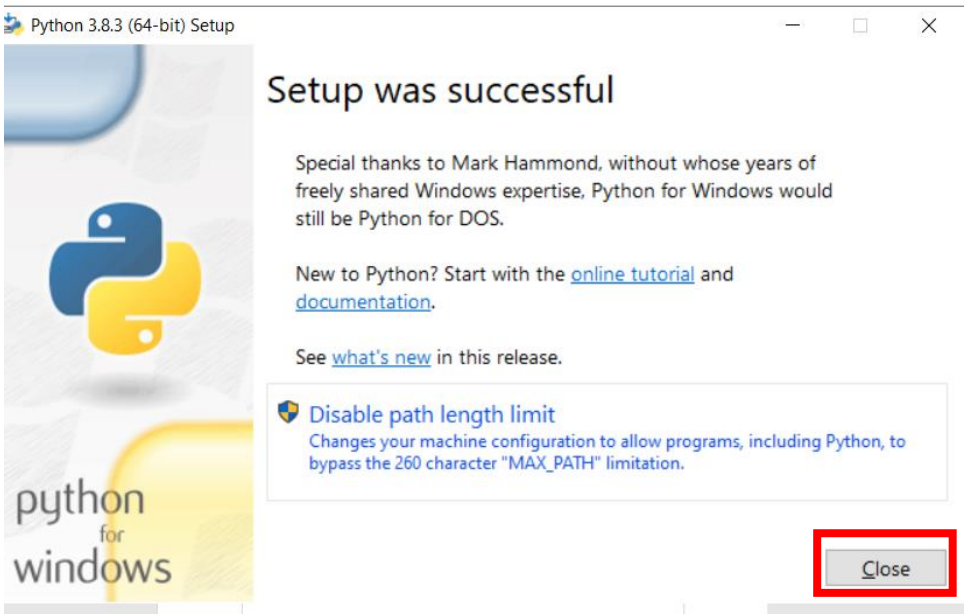


ILUSTRACIÓN 9. FINALIZACIÓN DE PROCESO DE INSTALACIÓN DE PYTHON

Para comprobar que Python se instaló con éxito con el símbolo del sistema (el cual se puede buscar dentro del menú de programas y la barra de búsqueda de Windows), sólo se deberá ejecutar y teclear “python”, teclear enter y observar lo siguiente:

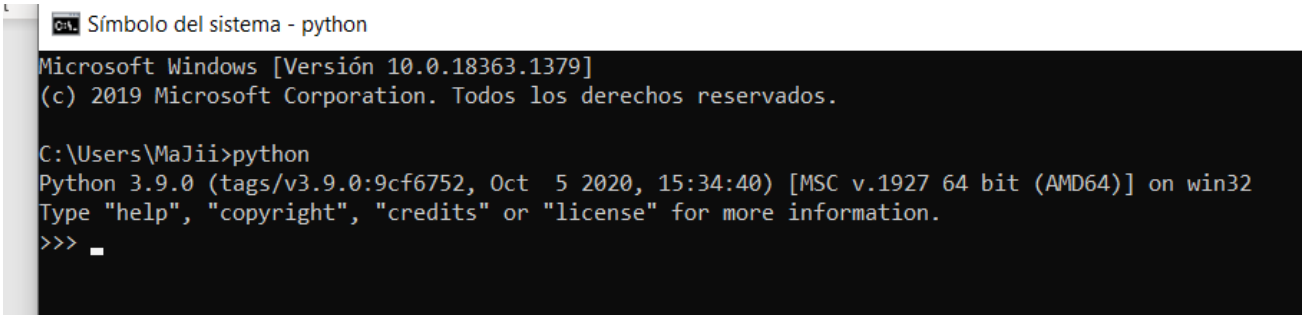


ILUSTRACIÓN 10. COMPROBACIÓN DE INSTALACIÓN DE PYTHON

Descarga de Sublime Text 3

Sublime Text es un editor de texto y editor de código fuente, está escrito en C++ y Python. Se puede descargar y evaluar de forma gratuita. Sin embargo, no es software libre o de código abierto y se debe obtener una licencia para su uso continuado. Aunque la versión de evaluación es plenamente funcional y no tiene fecha de caducidad. Para acceder a la descarga del programa debe de seguir el siguiente enlace: https://www.sublimetext.com/3 donde se podrá observar la siguiente interfaz:

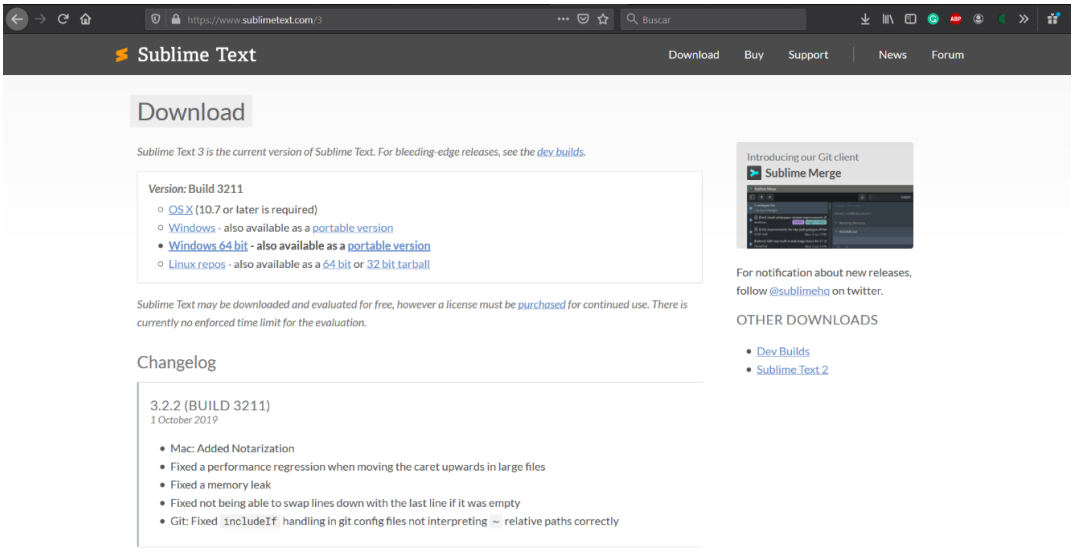


ILUSTRACIÓN 11. ZONA DE DESCARGAS DE SUBLIME TEXT 3

En el primer apartado que se observa, donde dice “Download”, dar clic sobre el enlace de Windows (recordar verificar si el Windows es de 32 o de 64 bits para elegir la versión adecuada). En este caso seleccionar la opción de Windows 64 bit y aparecerá la siguiente ventana. Repetir la acción de “Guardar archivo” al igual que con la descarga de Python.

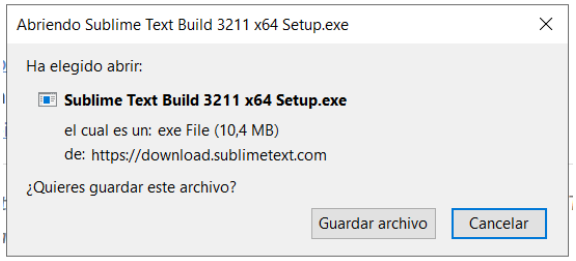


ILUSTRACIÓN 12. PROCESO DE GUARDADO EN EL EQUIPO

Proceder a abrir el archivo desde el gestor de descargas en el navegador o la carpeta asignada por el usuario para descargas de archivos nuevos. Ejecutar el instalador del programa, el cual se mostrará como lo muestra la **Ilustración 13.** Dar clic al botón de “Instalar”.

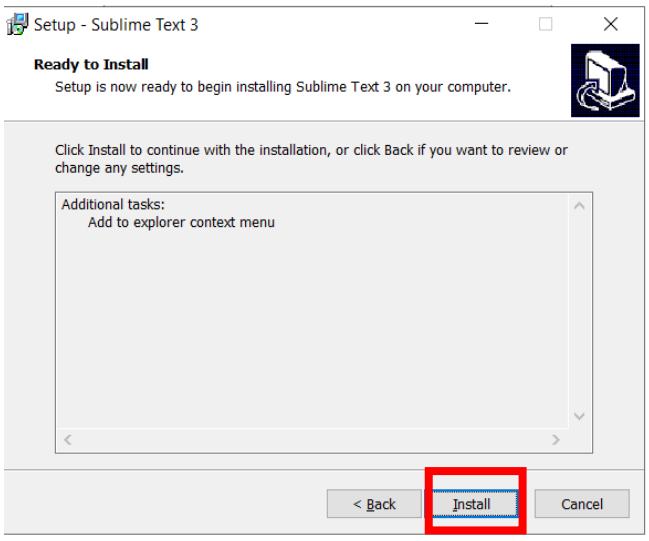


ILUSTRACIÓN 13. COMIENZO DE INSTALACIÓN DE SUBLIME TEXT 3

Comenzará la instalación del programa y esperar. Se mostrará la ventana siguiente:

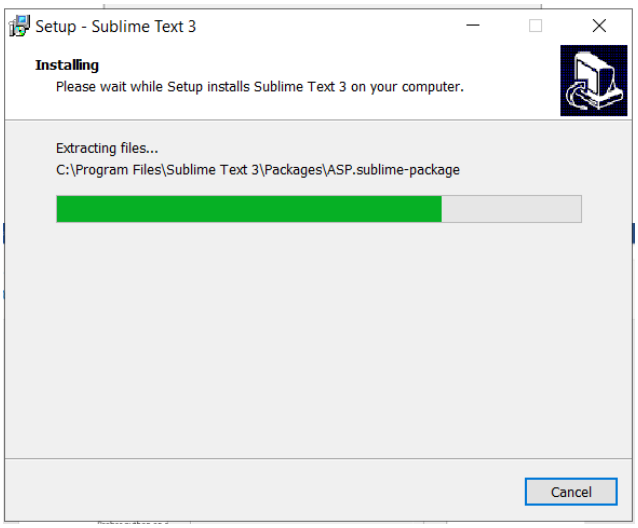


ILUSTRACIÓN 14. PROCESO DE LA INSTALACIÓN

Para finalizar la instalación, simplemente dar clic en “Finalizar”.

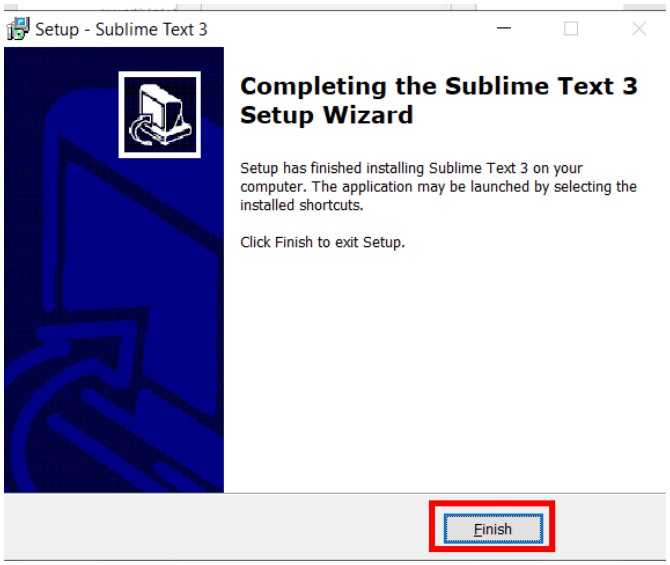


ILUSTRACIÓN 15. FINALIZACIÓN DE LA INSTALACIÓN DE SUBLIME TEXT 3

Una vez instalado el programa se puede crear un acceso directo al escritorio o anclar el programa a la barra de tareas. Accediendo a éste se verá la interfaz que se muestra a continuación:

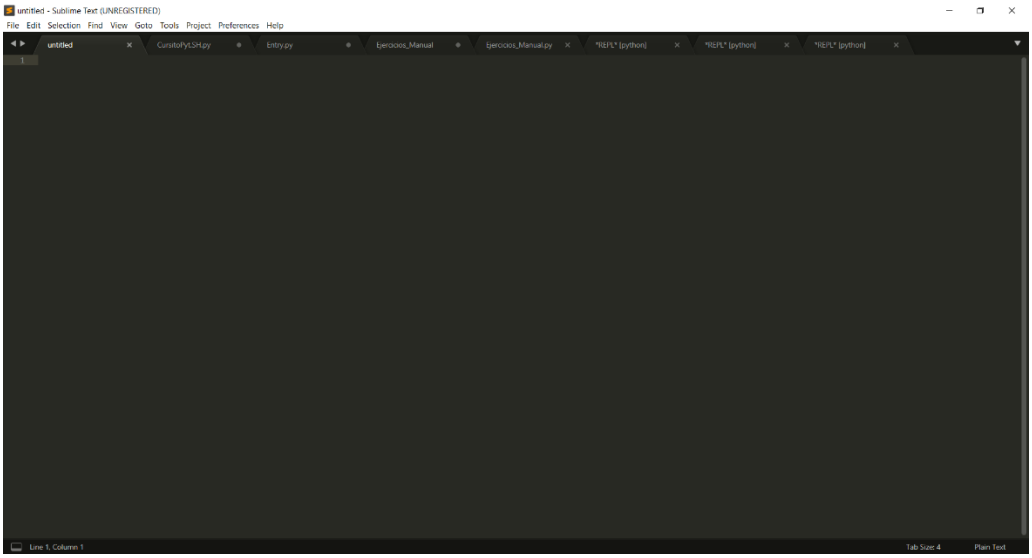


ILUSTRACIÓN 16. INTERFAZ DE TRABAJO DE SUBLIME TEXT 3

Se pueden modificar varias opciones de la ventana en la pestaña de “View”, por ejemplo, añadir la visualización de la barra lateral, mostrar la consola, entrar en modo de pantalla completa, etc.

Sublime REPL

Para poder ejecutar código Python (y de muchos otros lenguajes) en Sublime se debe instalar una herramienta llamada “Sublime REPL”. Para poder descargarlo, ir a la pestaña de “Herramientas” y dar clic en “Command Palette”.

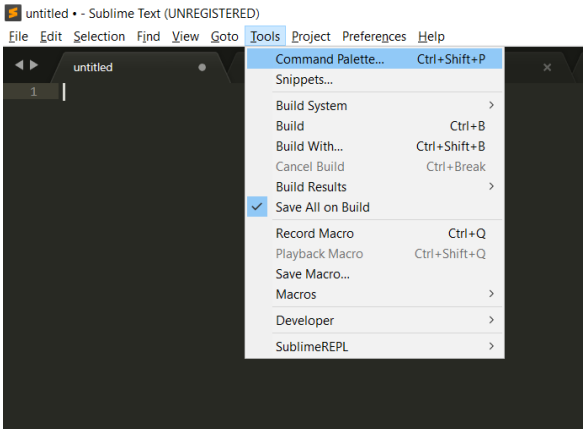


ILUSTRACIÓN 17. DESPLIEGUE DEL MENÚ TOOLS (HERRAMIENTAS) PARA ACCEDER A LA OPCIÓN COMMAND PALETTE

Después de dar clic aparecerá una especie de buscador dentro del programa como se ve a continuación:

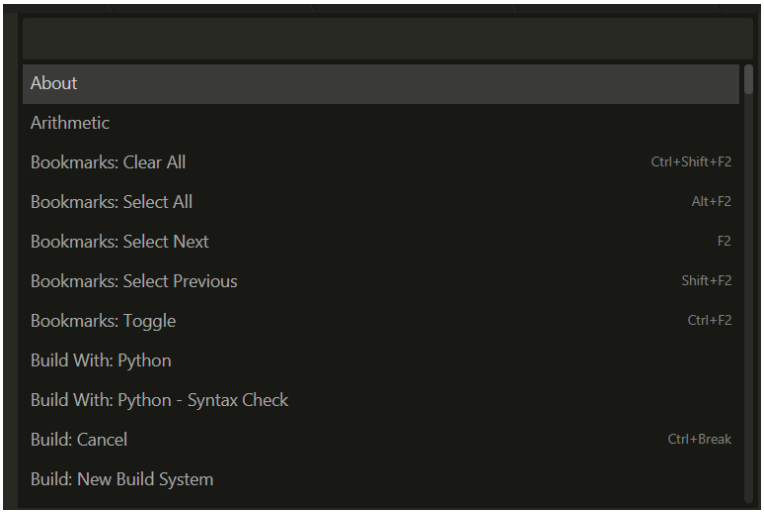


ILUSTRACIÓN 18. DESPLIEGUE DEL COMMAND PALETTE

Colocar en el buscador “Install Package” y seleccionar el “Package Control” dando clic sobre la opción.

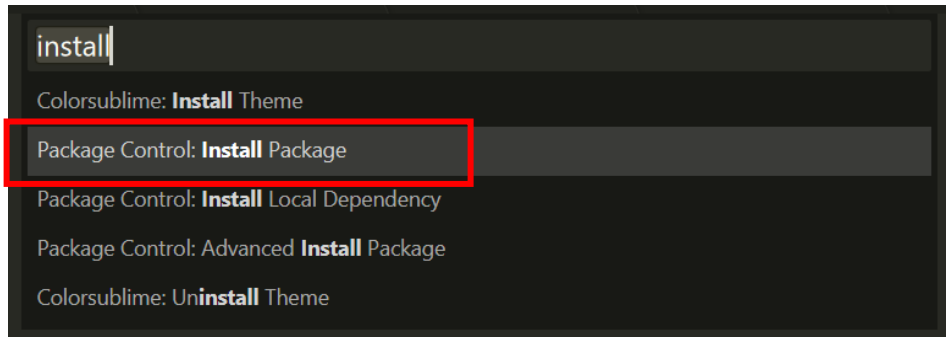


ILUSTRACIÓN 19. BÚSQUEDA DEL PAQUETE A INSTALAR

Después de dar clic aparecerá una ventana con el siguiente mensaje. Dar clic en “Aceptar”.

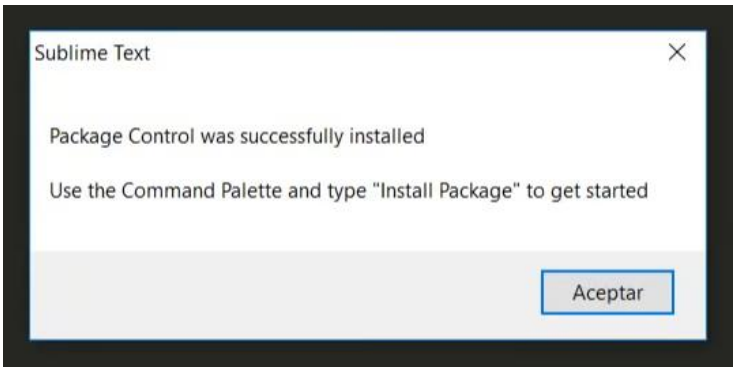


ILUSTRACIÓN 20. VENTANA RESULTANTE DE LA INSTALACIÓN

Se deberá atender a la instrucción de la ventana emergente: volver a abrir el palette, introducir la búsqueda “Install Package” e instalar el Package Control. Una vez instalado el paquete se abrirá automáticamente otra búsqueda, como se observa a continuación:

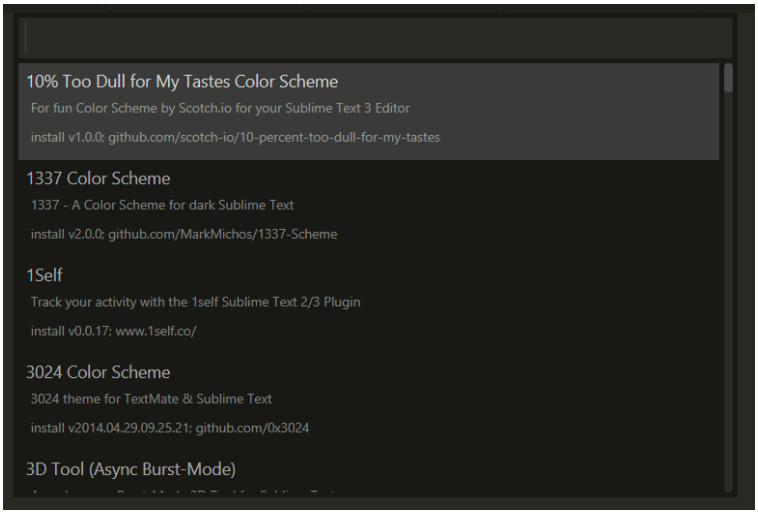


ILUSTRACIÓN 21. BÚSQUEDA DEL SEGUNDO PAQUETE A INSTALAR

Dentro de este nuevo buscador introducir la búsqueda de “SublimeREPL” y dar clic sobre la misma:

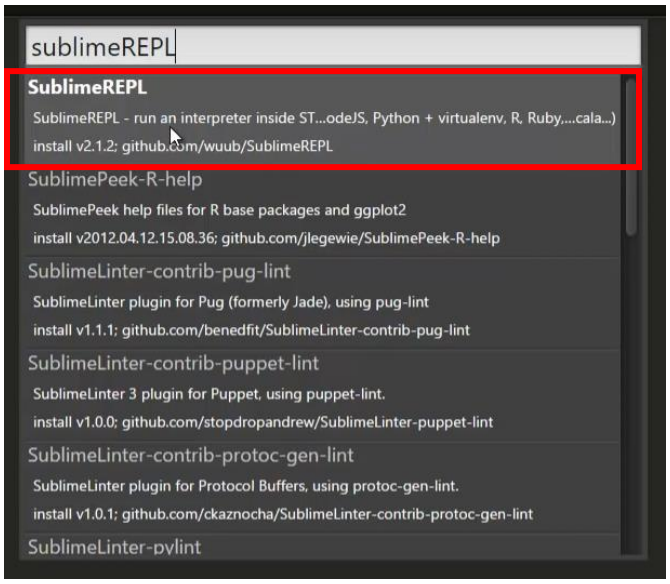


ILUSTRACIÓN 22. INSTALACIÓN DE SUBLIME REPL

Una vez instalado SublimeREPL podrá ser utilizado y visualizado en la pestaña de Herramientas. Se encuentra al final de la lista de la pestaña como se muestra en la siguiente Ilustración 23. Para trabajar REPL con Python se debe de posicionar el cursor sobre SublimeREPL, después buscar el lenguaje de Python y seleccionar la opción de Python (esto permite abrir una interfaz para trabajar directamente sobre ella).

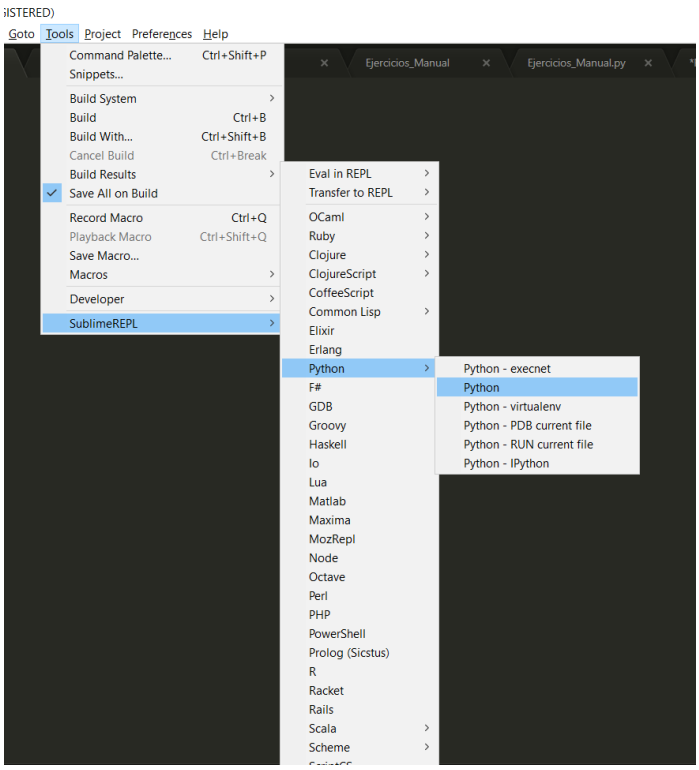


ILUSTRACIÓN 23. DESPLIEGUE DEL MENÚ TOOLS Y ACCESO A LA HERRAMIENTA SUBLIME REPL

Siguiendo los pasos anteriores se abrirá una pestaña nueva en Sublime con el nombre de "*REPL*[python]". Se informará acerca de la versión instalada con la cual se trabajará. En este caso deberá de ser Python 3.8.3 y aparecerá su respectiva fecha de última actualización.

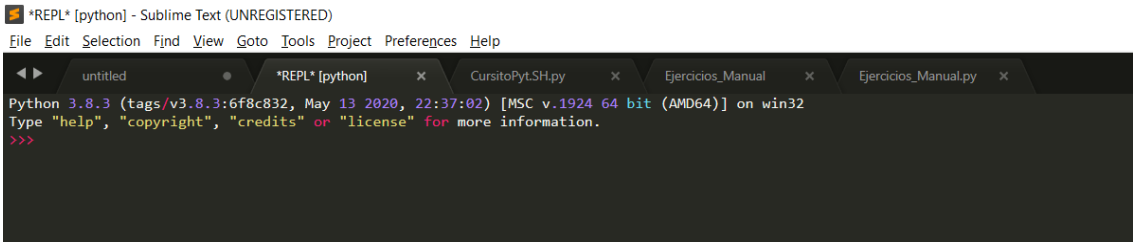


ILUSTRACIÓN 24. SUBLIME REPL Y DATOS ACERCA DE LA VERSIÓN DE PYTHON INSTALADA EN EL EQUIPO.

Para poder trabajar de manera más cómoda y tener la posibilidad de guardar y/o modificar los archivos que se necesiten crear, se puede trabajar desde el editor de texto. Deberá de abrirse la opción de herramientas del menú y colocar el cursor sobre la opción "Build System" y por último dar clic en la opción de "Python" o "Python 3".

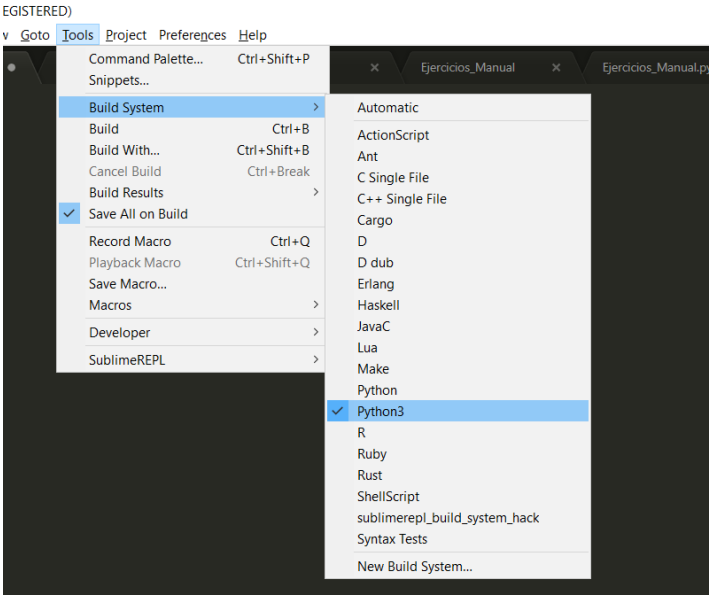


ILUSTRACIÓN 25. CONFIGURACIÓN ESPECÍFICA PARA LA ESCRITURA EN SUBLIME REPL

Se puede monitorear el trabajo si se abre la consola del programa. Si se introdujera un script y se quiere verificar su funcionalidad tendría que ser guardado con el comando ctrl + s y, posteriormente, abrir la consola ejecutando el comando ctrl + b. Si se quieren visualizar cambios realizados en el script deben de ser guardados antes de ejecutar el comando para actualizar la consola. Este comando brinda la aparición de una sección en la parte inferior de Sublime y se ve de la siguiente manera:

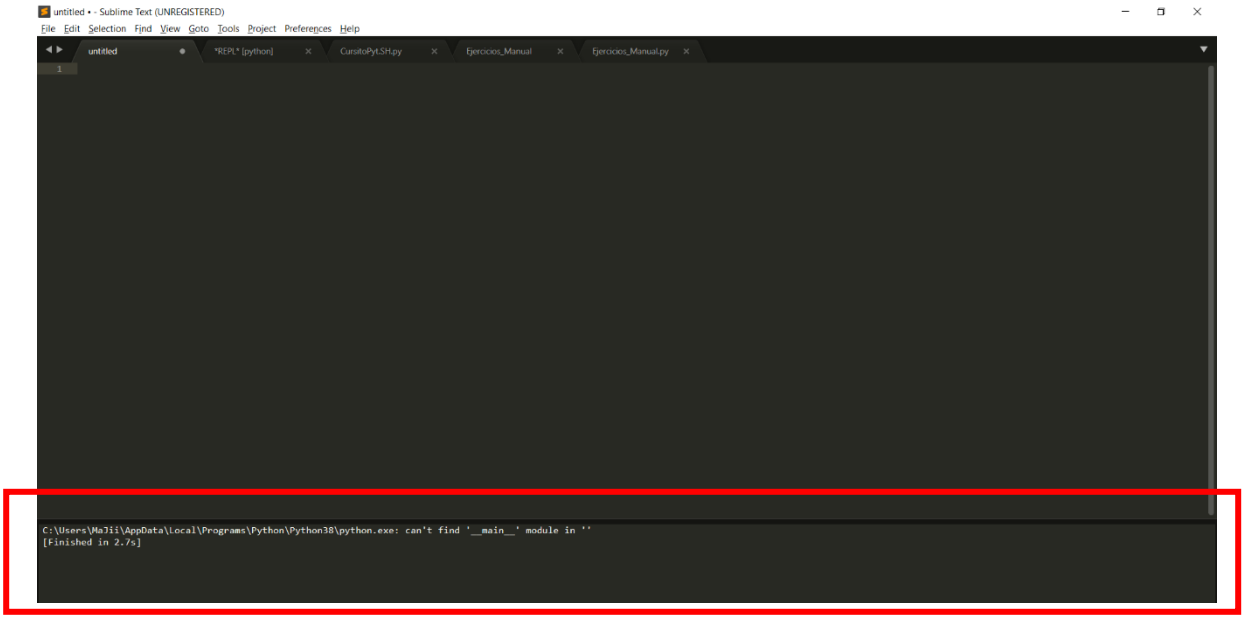


ILUSTRACIÓN 26. INTERFAZ DE SUBLIME TEXT CON VENTANA DE VERIFICACIÓN DE CÓDIGO EN CONSOLA

Crear combinación de teclas:

Si se desea crear una combinación de teclas para acceder al REPL de Python (estilo de trabajo que se mantendrá durante la mayor parte del tiempo del uso del manual) se puede hacer lo siguiente:

1. Dar clic en “Preferences” > “Key Bindings” y observa el panel donde dice “User”.

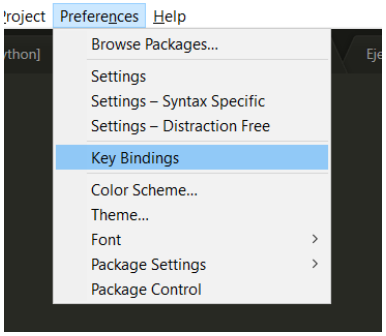


ILUSTRACIÓN 27. DESPLIEGUE DEL MENÚ PREFERENCES PARA CONFIGURACIÓN DE ATAJO DE TECLAS ASOCIADO A SUBLIME REPL

2. En el panel de usuario (la parte derecha) colocar el siguiente código dentro de los corchetes que aparecen:

```
{          "keys":          ["ctrl+alt+b"],          "command":
"run_existing_window_command", "args":
{
    "id": "repl_python_run",
```

Nota: Si se desea asignar otro comando de teclas para ejecutar el REPL puede hacerse.

Deberá observarse como se muestra a continuación:

ILUSTRACIÓN 28. MODIFICACIÓN EN SUBLIME PARA EJECUCIÓN DE REPL CON COMANDO DE TECLAS

3. Guardar con ctrl + s

Ahora se puede ejecutar el código en el REPL mediante la combinación de teclas ctrl + alt + b.

Recomendaciones:

- ✓ Trabajar usando 2 modalidades (detalladas en el tema de "Sintaxis Básica"):
    a) con el programa Sublime como editor de texto para poder crear archivos nuevos, modificarlos y manipularlos. Una manera rápida es ejecutando el comando *ctrl + n* para crear un nuevo archivo o abrir una nueva pestaña.

b) Con el intérprete de Python (REPL) cuando se trabaje con comandos y scripts más sencillos.

- ✓ Es recomendable crear una carpeta con un nombre "Curso Python" o el nombre deseado para poder trabajar de manera ordenada y tener todos los recursos generados en un solo lugar. Posteriormente se pueden crear carpetas por capítulo, actividad o tema.

- ✓ Para guardar el archivo en Python debe de ser añadida la extensión ".py" al final de éste. Dar clic sobre la opción "File" en la barra del menú y, a continuación, clic en "Save As". Posteriormente, elegir la carpeta de destino y el nombre con la extensión antes mencionada: archivoejemplo.py

- ✓ Si se añadirán espacios en el nombre de los archivos emplear guion bajo: "_" (por ejemplo: en lugar de "Archivo nuevo.py" se nombrará como "Archivo_nuevo.py") para poder incluirlos en códigos de futuros ejercicios y por comodidad.

- ✓ Una vez nombrado el nuevo archivo se puede actualizar y guardar cambios con el comando *ctrl + s*.

- ✓ No es recomendable copiar y pegar los comandos, ya que, puede ocurrir un desacomodo del script y tener errores de ejecución.

- ✓ Si se guarda un archivo en Python (".py") será más fácil la adecuación al lenguaje de programación debido a que Sublime es una herramienta muy visual y hace diferenciaciones de código con colores vivos.

- ✓ Siempre que se observe un cuadro como el siguiente, se trata de código ingresado en el editor de texto o de una ejemplificación sobre la estructura de trabajo en un tema:

```
Código
Código
```

- ✓ Cuando se observe de la siguiente manera, se trata de código trabajado en el intérprete de Python o introducción de datos posterior a ejecución de código elaborado en el editor de textos:

```
>>> Código
>>> Código
```

- ✓ Al tener imágenes donde se logran apreciar colores (interfaz de Sublime) suelen ser ejemplos de diferenciación y separación real del código o resultados que deberán de ser obtenidos.

- ✓ El output será la respuesta que se debe esperar o visualizar una vez ejecutado el código del ejercicio que le antecede dentro del cuadro de texto.

```
>>> Código…

Output:
```

```
Código…

Output:
```

Capítulo II. Sintaxis básica, generalidades y conceptos

Modo interactivo y modo script

Al iniciar en Python se puede trabajar de dos formas, mediante el modo interactivo, ejecutando la orden en tiempo real; y en modo script escribiendo el código en Python y ejecutando después con el intérprete. La sintaxis en Python permite usar el modo interactivo activando el intérprete que una vez puesto en marcha, ya se pueden ejecutar órdenes sin ningún problema. Una vez empezando a programar con Python resultará mucho más sencillo utilizar un script con todas las órdenes a ejecutar que dentro del intérprete.

¿Qué es un script?

Un script, secuencia de comandos o (del inglés al español) un guion, es un documento que puede ser simple o complejo que contiene instrucciones, escritas en códigos de programación para poder ejecutar programas relativamente simples. El scripting es un estilo de programación donde se ejecutan diversas funciones en el interior de un programa de computadora: se pueden usar para realizar prototipos de programas, automatizar tareas repetitivas, hacer procesamiento por secciones e interactuar con el sistema operativo y el usuario (debido a esto, los intérpretes de comandos o Shells suelen diseñarse con funcionalidades de programación).

Python es un lenguaje de programación que utiliza scripts y, aunque es sencillo, pueden crearse programas completos, complejos e importantes para la ciencia y la sociedad tales como OpenMDAO de la NASA en donde se da resolutiva a problemas de optimización de diseños multidisciplinarios o algunos más sencillos, pero con presencia en la vida de todos siendo de aporte a Google o Netflix.

¿Qué es un intérprete?

En informática, el Shell o intérprete de órdenes o de comandos es el programa informático que provee una interfaz de usuario para acceder a los servicios del sistema operativo. Para

poder crear y ejecutar scripts se necesita de 2 herramientas: un editor de texto (Sublime) y un intérprete (Sublime REPL para Python).

Prompt

Se llama prompt al carácter o conjunto de caracteres que se muestran en una línea de comandos para indicar que está a la espera de órdenes. Éste puede variar dependiendo del intérprete de comandos y suele ser configurable. En Python el prompt se conoce como chevron Prompt y se compone por tres signos lógicos "mayor qué" de manera consecutiva, se observa de la siguiente manera:

ILUSTRACIÓN 29 A Y 29 B. PROMPT EN SUBLIME REPL PARA PYTHON Y TECLA ENTER.

Una vez introducida la función, la instrucción, comando, etc., deberá ser pulsada la tecla enter para ejecutar la acción y proceder a observar el resultado.

Imput

Es la introducción de datos de distintos tipos desde la entrada estándar que normalmente se corresponde con la entrada de un teclado.

Output

Tras recibir los datos, el programa analizará la información y realizará una acción en consecuencia, mostrando un resultado (a este proceso se le denomina OUTPUT).

¿Qué es un objeto en Python?

En Python todo es considerado como un objeto: una variable, una secuencia de caracteres, una lista, etc. Normalmente, un objeto es creado a partir de una clase (aunque esto no es necesario) y son entidades que almacenan información en forma de argumento para poseer atributos, y también realizan acciones organizadas a través de métodos que permiten cambiar dichos atributos o estados del objeto o de otros objetos de la clase. Se puede consultar esto más detalladamente en el tema "Clases".

Estado (en objetos)

Cuando se tiene un objeto, sus propiedades toman valores. Por ejemplo, cuando se considera una tableta o cápsula como objeto, la propiedad *color* tomará un valor en concreto, como por ejemplo blanca o rosa. El valor concreto de una propiedad de un objeto se llama estado.

Comentarios

La sintaxis en Python determina que los comentarios empiezan con el carácter hash (#) y se observa de un color diferente al texto o instrucciones que serán tomadas en cuenta dentro del script o de la instrucción. Además, pueden incluirse en una línea propia o al lado de una orden. Un ejemplo de comentario puede observarse de la siguiente manera dentro de Sublime Text:

ILUSTRACIÓN 30.EJEMPLO DE VISUALIZACIÓN DE COMENTARIOS OPCIÓN 1

Otra opción son las triples dobles comillas, deberán incluirse en ambos lados del comentario o se alterará el código:

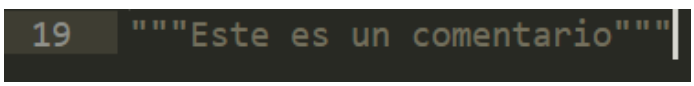


ILUSTRACIÓN 31. EJEMPLO DE VISUALIZACIÓN DE COMENTARIOS OPCIÓN 2

Identación

Se llama identación de código al hecho de utilizar sangrado (mover ligeramente hacia la derecha) en las líneas de código para facilitar la lectura e indicar visualmente si se encuentra en el interior de una función, bucle, condicional, etc. Lo correcto es emplear 4 espacios por cada nivel de Identación y **NUNCA** mezclar tabuladores con espacios, lo cual ya está deshabilitado para Python 3. En Sublime, al definir que se emplea el lenguaje de Python la tecla Tab se configura automáticamente para añadir los 4 espacios para la Identación correcta (esto puede observarse en la parte inferior derecha del programa). Además, hay ciertas estructuras en Python que van a presentar identación automáticamente.

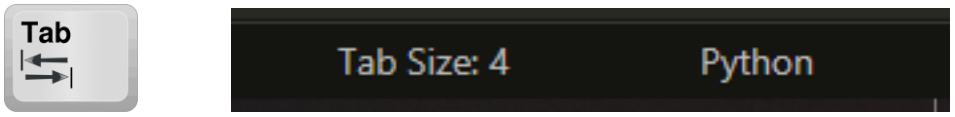


ILUSTRACIÓN 32A Y 32B. TECLA "TABULADOR" E INDICACIÓN DE ESPACIOS ASIGNADO POR DEFECTO EN SUBLIME TEXT PARA DICHA TECLA

Palabras exclusivas de Python

En los lenguajes de programación existen palabras RESERVADAS, lo cual, implica que estas palabras no pueden ser empleadas dentro del código que se esté elaborando, ya que, están asignadas a funciones específicas dentro del lenguaje, en este caso, Python. Para poder conocer estas palabras rápidamente, se puede introducir lo siguiente dentro del intérprete de Python:

```
>>> kwlist
```

Para ejecutar la acción se deberá emplear tecla enter. Deberá aparecer una lista con las palabras reservadas dentro del lenguaje de Python en su versión 3:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'el
    import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

ILUSTRACIÓN 33. PALABRAS RESERVADAS PARA PYTHON 3

Se puede verificar si una palabra está reservada utilizando el módulo integrado `keyword`, de la siguiente forma:

```
>>> import keyword
>>> keyword.iskeyword('as')

Output: True

>>> keyword.iskeyword('x')

Output: False
```

Ayuda

Cuando se está trabajando en Python o cualquier lenguaje de programación es común desconocer el uso o funcionalidad total de los elementos que lo componen, así como su correcta sintaxis. También se puede desear saber más sobre lo que se está trabajando en el script o para cualquier otro aspecto útil, por lo que, se puede utilizar la función o comando `help()`, por ejemplo, de la siguiente manera:

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

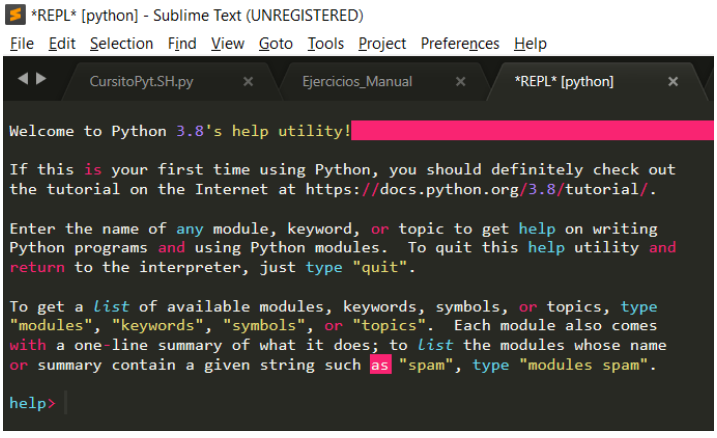


ILUSTRACIÓN 34. VISUALIZACIÓN DEL MODO INTERACTIVO DE AYUDA EN PYTHON

Se puede consultar, por ejemplo, sobre el módulo `os`. Se introduce la palabra "os" a continuación de la palabra "help" en el REPL y dar clic en la tecla enter:

```
help>os
```

Se desplegará toda una serie de información útil sobre el módulo o tema seleccionado de esta forma:

```
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're on.

MODULE REFERENCE
    https://docs.python.org/3.8/library/os

    The following documentation is automatically generated from the Python
    source files.  It may be incomplete, incorrect or include features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at the
    location listed above.

DESCRIPTION
    This exports:
      - all functions from posix or nt, e.g. unlink, stat, etc.
      - os.path is either posixpath or ntpath
      - os.name is either 'posix' or 'nt'
      - os.curdir is a string representing the current directory (always '.')
      - os.pardir is a string representing the parent directory (always '..')
      - os.sep is the (or a most common) pathname separator ('/' or '\\')
      - os.extsep is the extension separator (always '.')
      - os.altsep is the alternate pathname separator (None or '/')
      - os.pathsep is the component separator used in $PATH etc
      - os.linesep is the line separator in text files ('\r' or '\n' or '\r\n')
      - os.defpath is the default search path for executables
      - os.devnull is the file path of the null device ('/dev/null', etc.)

    Programs that import and use 'os' stand a better chance of being
    portable between different platforms.  Of course, they must then
    only use functions that are defined by all platforms (e.g., unlink
    and opendir), and leave all pathname manipulation to os.path
    (e.g., split and join).
```

ILUSTRACIÓN 35. EJEMPLO DE VISUALIZACIÓN DE BÚSQUEDA DE AYUDA

Para salir de ayuda se puede, simplemente, introducir “q” y dar clic en la tecla enter. Automáticamente se completará el campo con la palabra “quit” y al dar el segundo enter, saldrá.

```
    UNICODE = b'V'
    __all__ = ['PickleError', 'PicklingError', 'UnpicklingError', 'Pickler

FILE
    c:\users\majii\appdata\local\programs\python\python38\lib\pickle.py


help> q

You are now leaving help and returning to the Python interpreter.
If you want to ask for help on a particular object directly from the
interpreter, you can type "help(object)".  Executing "help('string')"
has the same effect as typing a particular string at the help> prompt.

***Repl Closed***
```

ILUSTRACIÓN 36. VISUALIZACIÓN DE FINALIZACIÓN DE AYUDA INTERACTIVA Y PARO DE EJECUCIÓN DE REPL

¿Existe una manera correcta de escribir códigos en Python? El PEP-8

La comunidad de usuarios de Python ha adoptado una guía de estilo que facilita la lectura del código y la consistencia entre programas de distintos usuarios. Esta guía no es de seguimiento obligatorio, pero es altamente recomendable de seguir. El documento completo se denomina PEP 8. Dentro de este documento se encuentran aspectos importantes que ayudan al programador y al usuario a que sea mucho más fácil de entender y de modificar (si es necesario). Algunos de estos aspectos, son:

- Identación
- Comentarios
- Tamaño máximo de línea
- Líneas y espacios en blanco
- Importaciones de código
- Idioma del código sugerido (inglés)

Si se desea consultar la guía completa de estilo para Python (copiando y pegando en el navegador) el siguiente enlace donde se encontrará en el idioma original: https://www.python.org/dev/peps/pep-0008/

Y si se prefiere la alternativa en el idioma español: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjp4_Hyq5vrAhUWCM0KHb4eBoIQFjAAegQIAxAB&url=http%3A%2F%2Frecursospython.com%2Fpep8es.pdf&usg=AOvVaw2M3yJYnkRofKEJG2qUfHhJ

Lo anterior se puede complementar con el pequeño resumen del *Zen de Python* el cual, se puede consultar por medio del intérprete de la siguiente manera:

```
>>> import this
```

Aparecerá una lista con frases útiles al momento de crear código:

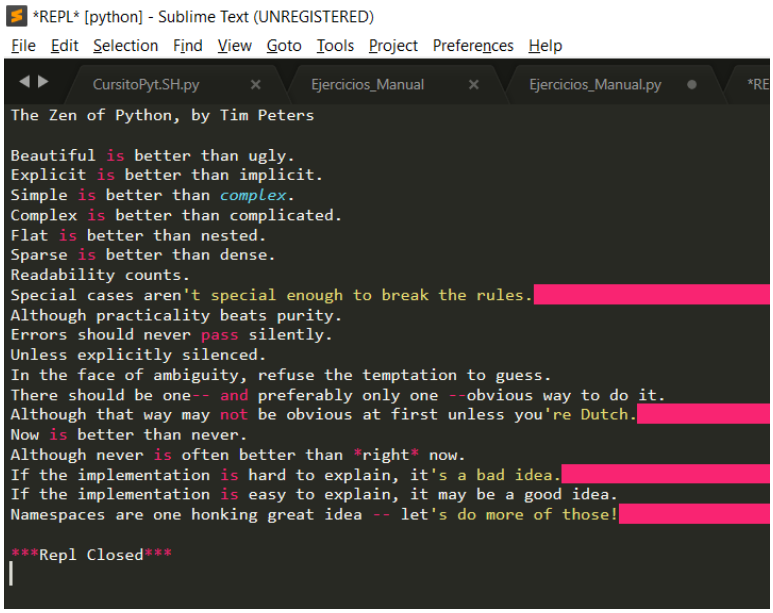


ILUSTRACIÓN 37. VISUALIZACIÓN DEL ZEN DE PYTHON

Mi primer programa en Python

Dentro del editor de texto crea un nuevo archivo y guárdalo como “miprimer.py” y escribir lo siguiente dentro:

```
print("Hola mundo!")
usuario = input("¿Cómo te llamas? Introduce tu nombre aquí:")

print("Hola", usuario, "bienvenido/a a la programación con
Python \n", "espero que te sea de mucha utilidad ¡y lo
disfrutes!")
```

Ejecuta con el comando antes mencionado: ctrl + alt + b. Introduce tu nombre ¡y listo! Crear el primer programa con Python no fue tan difícil, pero es necesario el poder comprender cómo está compuesto, por qué el texto se observó con esa distribución, por qué una instrucción utiliza comillas y otra no y por qué puede funcionar así. Además, Python es una herramienta muy potente, por lo que será necesario comprender a detalle las estructuras básicas del lenguaje. Para seguir avanzando en la compresión y manejo de Python, lo ideal será comenzar con los temas siguientes:

Variables y tipos de datos

En cualquier lenguaje de programación se tiene que realizar el manejo de diversos tipos de datos. Los tipos de datos definen un conjunto de valores que tienen una serie de características y propiedades determinadas.

En Python todo valor que pueda ser asignado a una variable (lugar donde se almacena algo) tiene asociado un tipo de dato. Como fue mencionado anteriormente, en Python todo es considerado como un objeto. Los tipos de datos pueden considerarse también como clases (donde se definen las propiedades y que puede hacerse con ellas) y las variables serían las instancias (objetos) de los tipos de datos. El tipo de dato establecerá qué valores puede tomar una variable y qué operaciones o acciones pueden llevarse a cabo sobre la misma.

¿Qué tipos de datos se pueden encontrar en Python?

Los tipos principales y básicos de datos son los numéricos (enteros, complejos y de punto flotante), los booleanos y los strings o cadenas de caracteres. Además, es posible encontrar otros tipos de datos como lo son: secuencias (listas, tuplas, rangos), mapas (diccionarios), conjuntos (tipo set), iteradores, clases, instancias y excepciones.

También es necesario tener en cuenta que estos datos pueden ser mutables (modificables) o inmutables (no modificables).

TABLA 1. TIPOS DE DATOS EN PYTHON

Categoría de tipo	Nombre	Descripción
Números inmutables	`int`	Entero
	`long`	entero long
	`float`	coma flotante
	`complex`	complejo
	`bool`	booleano
Secuencias inmutables	`str`	cadena de caracteres
	`unicode`	cadena de caracteres Unicode
	`tuple`	tupla
	`xrange`	rango inmutable
Secuencias mutables	`list`	lista
	`range`	rango mutable
Mapeos	`dict`	diccionario
Conjuntos mutables	`set`	conjunto mutable
Conjuntos inmutables	`frozenset`	conjunto inmutable

Asignación o definición de variables

Las variables en Python se crean cuando se definen por primera vez, es decir, cuando se les asigna un valor (cualquiera de los que se mencionan anteriormente) por primera vez. Para asignar un valor a una variable se utiliza el operador de igualdad (=). El nombre de la variable y el contenido de este será decidido por el usuario, sin embargo, en los ejemplos se emplea el nombre y tipo de variable que le corresponde para facilitar el entendimiento:

```
integer_variable = 1
float_variable   = 1.1
string_variable  = "Hola"
boolean_variable = True
lista_variables   = [1,2,3]
diccionario_variables                                          =
{"jose":660234567,"Antonio":660234989}
tuplas_variables = ("hola","adios")
```

Tipos numéricos: números enteros, de punto flotante (ya no es necesario especificarlo en Python 3) y complejos: `Int, float y complex`

Tipo booleano **(bool)**

En Python la variable de tipo booleano se representa como `bool.` Esta clase se puede instanciar únicamente con dos valores: `True` para representar que es verdadero y `False` para representar falso. Una particularidad de Python es que cualquier objeto se puede usar en un contexto donde se necesite corroborar si algo es verdadero o no, por lo tanto, cualquier objeto puede usarse en la condición de un `if` o un `while` (estructuras de control vistas en el tema "Control de Flujo") o como operandos de una operación booleana.

Los siguientes objetos o instancias también es correcto considerarlas como falsas:

- Secuencias o colecciones vacías
- Valores de tipo numérico iguales a 0
- `None`

Y cualquier objeto es considerado como verdadero con excepción de aquellos que implementen los siguientes métodos:

- `__bool__()` y este devuelva en respuesta un `False`
- `__len__()` y este devuelva como respuesta 0

Tipo string o cadena de caracteres `(str)`

Un string es una secuencia inmutable de caracteres en formato *Unicode.* Para crear un string se tiene que introducir una secuencia de caracteres entre comillas simples o dobles. Se puede usar indistintamente comillas simples o dobles, con una particularidad. Si en la cadena de caracteres se necesita usar una comilla simple, tienes dos opciones: usar comillas dobles para encerrar el string, o bien, usar comillas simples, pero anteponer el carácter \ a la comilla simple del interior de la cadena. El caso contrario es similar.

```
>>> hola = 'Hola "Mundo"'
>>> hola_2 = 'Hola \'Mundo\''
>>> hola_3 = "Hola 'Mundo'"

>>> print(hola)

Output: Hola "Mundo"

>>> print(hola_2)

Output: Hola 'Mundo'

>>> print(hola_3)

Output: Hola 'Mundo'
```

Comprobación de tipo de variable

Para conocer o comprobar qué tipo de dato se ha introducido en una variable se puede realizar de manera muy sencilla. Si se retoma el ejemplo de asignación de variables se puede reafirmar de qué tipo de datos se trata:

```
>>> integer_variable = 1
>>> float_variable   = 1.1
>>> string_variable  = "Hola"
>>> boolean_variable = True
>>> list_variable    = [1,2,3]
>>>diccionario_variable  =
{"jose":660234567,"Antonio":660234989}
>>> tuplas_variable = ("hola","adios")
```

Una vez introducidas las variables, para comprobar su tipo se hará de la siguiente manera:

`type()` recibe como parámetro un objeto y lo devuelve al mismo tiempo

```
>>> type (integer_variable)
>>> type (float_variable)
>>> type (string_variable)
>>> type (boolean_variable)
>>> type (list_variable)
>>> type (dict_variable)
>>> type (tuplas_variable)
```

`isinstance()` que recibe 2 parámetros: un objeto y un tipo. Este devolverá `True` si el objeto es del tipo pasado como parámetro y False de no ser así. Por ejemplo:

```
>>> isinstance(3, float)

Output:
False

>>> isinstance(3, int)

Output:
True
```

Conversión de tipos

Conocer el cómo realizar cambios de tipos de datos resulta de utilidad. Por ejemplo, si se tiene un dato numérico como "edad" que es de tipo string con un valor de "15" se podría decir que, aunque la variable contiene datos de tipo carácter de igual manera es un número. Sin embargo, si se intenta sumar 15 a "edad", el intérprete nos arrojará un error porque son datos de tipos distintos:

```
>>> edad = “15”
>>> edad = edad + 15

Output:
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Esto puede arreglarse si se realiza una conversión del tipo de dato. En Python existen las siguientes funciones para realizar conversiones:

- **`str()`** la cual devuelve la representación del objeto que se pasa como parámetro en forma de cadena de caracteres
- **`int()`** devuelve una variable numérica desde un número o una cadena de caracteres
- **`float()`** devuelve un número de punto flotante a partir de un número o secuencia de caracteres
- **`complex()`** devuelve un número complejo a partir de un número o secuencia de caracteres

Si se introduce de manera equivocada o inválida el objeto que se pasa como parámetro, entonces, el intérprete nos arrojará un error. Si se aplica lo anterior al ejemplo de la edad, se obtiene lo siguiente:

```
>>> edad = "15"
>>> edad = int(edad) +15

>>> print (edad)

Output:
30
```

Operadores

En los lenguajes de programación, los operadores son los elementos que relacionan de forma diferente los valores de una o más variables, es decir, un operador aritmético permite realizar

una operación matemática, un operador relacional permite comparar datos y un operador lógico permite realizar una operación lógica.

TABLA 2. OPERADORES DE ASIGNACIÓN

Operador	Sintaxis	Función
Igual a	=	Asigna a la variable del lado izquierdo cualquier variable o resultado del lado derecho.
Restar a	−=	Resta a la variable del lado izquierdo el valor del lado derecho.
Sumar a	+=	Suma a la variable del lado izquierdo el valor del lado derecho.
Multiplicar a	*=	Multiplica a la variable del lado izquierdo el valor del lado derecho
Calcular exponente	**=	Calcula el exponente a la variable del lado izquierdo el valor del lado derecho.
Dividir a	/=	Divide a la variable del lado izquierdo el valor del lado derecho
División entera de	//=	Calcula la división entera a la variable del lado izquierdo el valor del lado derecho.
De módulo	%=	Devuelve el resto de la división a la variable del lado izquierdo el valor del lado derecho.

Ejemplo:

```
>>> a = 1
>>> b = 2
>>> a -= b
>>> a

Output:
-1

#Observar que el cambio es permanente y el nuevo valor de a
es -1#
```

TABLA 3. OPERADORES ARITMÉTICOS

Operador	Sintaxis
Suma	+
Resta	-
Negación	-
Multiplicación	*
Exponente	**
División	/
División entera	//
Módulo	%

Orden de importancia descendente

1. Exponente: **
2. Negación: -
3. Multiplicación, división, división entera, módulo: *, /, //, %
4. Suma, Resta: +, -

Ejemplo:

```
>>> a
Output: -1

>>> a + b
Output: 1
```

TABLA 4. OPERADORES DE RELACIÓN O RELACIONALES

Operador	Sintaxis	Función
De igualdad	==	Evalúa si los valores son iguales para diferentes tipos de datos
Desigualdad	!= ó <>	Evalúa si los valores son distintos
Menor qué	<	Evalúa si el valor del lado izquierdo es menor que el valor del lado derecho.
Mayor qué	>	Evalúa si el valor del lado izquierdo es mayor que el valor del lado derecho.
Menor o igual qué	<=	Evalúa si el valor del lado izquierdo es menor o igual que el valor del lado derecho.
Mayor o igual qué	>=	Evalúa si el valor del lado izquierdo es mayor o igual que el valor del lado derecho.

Ejemplos:

```
>>> a = -1
>>> b = 2
>>> c = -1

>>> a == b

Output:
False

>>> a == c

Output:
True
```

Operadores lógicos

Estos son los distintos tipos de operadores con los que puede trabajar con valores booleanos.

TABLA 5. OPERADORES LÓGICOS

Operador	Función
`And`	Evalúa si el valor del lado izquierdo y el lado derecho se cumple
`Or`	Evalúa si el valor del lado izquierdo o el lado derecho se cumple
`Not`	Devuelve el valor opuesto al valor booleano

Funciones

Una función es un bloque de código con un nombre asociado, que recibe cero o más argumentos como entrada, sigue una secuencia de sentencias u órdenes, la cuales ejecuta una operación deseada y devuelve un valor y/o realiza una tarea, así, este bloque puede ser llamados cuando se necesite. El uso de funciones es un componente muy importante del paradigma de la programación llamada estructurada y tiene varias ventajas:

- **modularización**: permite segmentar un programa complejo en una serie de partes o módulos más simples, facilitando así la programación y el depurado.
- **reutilización**: permite reutilizar una misma función en distintos programas.

Python dispone de una serie de funciones integradas al lenguaje, y también permite crear funciones definidas por el usuario para ser usadas en sus propios programas. Las funciones disponibles en el lenguaje se clasifican de la siguiente manera:

- Personalizadas o creadas por el usuario
- Avanzadas
- Recursivas
- Orden Superior
- Integradas

A continuación, se revisarán algunos ejemplos de cada una de las clases de funciones que se mencionaron anteriormente.

Funciones definidas por el usuario

Sentencia `def`

Es una definición de función para crear objetos que son funciones definidas por el usuario. Una definición de función es una sentencia ejecutable. Su ejecución enlaza el nombre de la función en el namespace local actual a un objecto función (un envoltorio alrededor del código ejecutable para la función). Este objeto función contiene una referencia al namespace local global como el namespace global para ser usado cuando la función es llamada.

La definición de función no ejecuta el cuerpo de la función; esto es ejecutado solamente cuando la función es llamada. La sintaxis para una definición de función en Python es:

```
def nombre(lista_de_parametros):
    """docstring_de_funcion_opcional(comentario)"""
    sentencias (cuerpo de función)
    return [expresion o valor]
```

Donde:

- `nombre`, es el nombre de la función.
- `lista_de_parametros`, es la lista de parámetros que puede recibir una función.
- `docstring_de_funcion`, es la cadena de caracteres usada para documentar la funcialidad o algún comentario acerca de la función. Ésta es opcional para el desarrollador o para el usuario final.
- sentencias, es el bloque de sentencias en código fuente python que realizar cierta operación dada.
- `return`, es la sentencia de devolución en código Python.
- `expresion`, es la expresión o variable que devuelve la sentencia `return`.

Un rápido ejemplo que puede resultar muy práctico para entender cómo funciona la definición de una nueva función puede ser tomando un sistema de calificaciones, donde se definirá como aprobatoria o reprobatoria una calificación o nota. Se pueden asignar dos

posibles respuestas: aprobado o reprobado. Se deben de definir bajo qué parámetros será retornado un mensaje u otro. Al momento de estar ingresando el código en consola o en el editor de texto se debe ser sumamente cuidadosos con la IDENTACIÓN, ya que, suelen ser los errores más comunes cuando la dificultad de código comienza a elevarse.

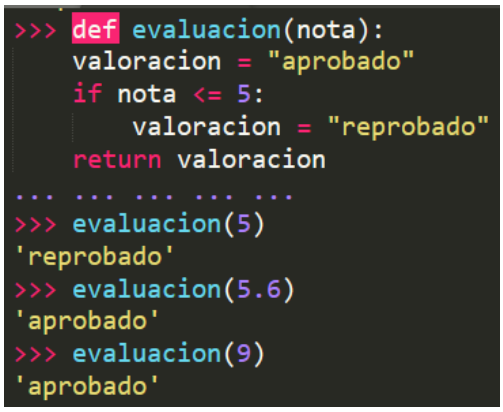


ILUSTRACIÓN 38. EJEMPLO DE EJECUCIÓN DE UNA FUNCIÓN SENCILLA

En caso de que se esté trabajando desde el editor de textos porque el código es más extenso y complejo o simplemente por comodidad, deberá tener una adición de la instrucción `print()` y la función que se crea de la siguiente manera. Si se desea ejecutar esta función se deberá aplicar la combinación de teclas que se asignaron al REPL: ctrl + alt + b.

```
def evaluacion(nota):
     valoracion = "aprobado"
     if nota <= 5:
          valoracion = "reprobado"
     return valoracion

print(evaluacion(8))
#Aquí se debe introducir la nota que se evaluará#

Output:
aprobado
```

En esta función es llamada “evaluación” y debe de recibir el parámetro de “nota” para ser puesta en marcha. La función incluye el resultado por defecto de “aprobado” al ser ejecutada. Si la nota fuese menor o igual al 5, el resultado que se mostraría en consola sería

aquel de “reprobado”. Como se trabaja desde el editor de textos, se tendrá que imprimir el resultado, llamando a la función e introduciendo el parámetro a evaluar (la calificación). Si se desea un rango más justo o evaluar parámetros diferentes como concentraciones de soluciones, diámteros de tabletas, temperaturas, etc., es un buen inicio para practicar la creación de funciones. Las estructuras de control de flujo como `if` y `return` se explican en el capítulo de “Control de Flujo”.

Argumentos indeterminados

En estos casos se pueden utilizar los parámetros indeterminados por posición y por nombre o por ambos.

Sentencia **`pass*`**

Es una operación nula, cuando es ejecutada nada sucede. Eso es útil como un contenedor cuando una sentencia es requerida sintácticamente, pero no necesita código que ser ejecutado, por ejemplo, para una función que no hace nada o para una clase que no posee un método aún:

```
>>> class Mutaciones: pass #Clase sin método aún#
>>> def Consultar_cama_paciente(numero_cama): pass #Una
función sin un objetivo aún#
```

Sentencia **`return`**

Las funciones pueden comunicarse con el exterior de estas al proceso principal del programa usando la sentencia `return.` El proceso de comunicación con el exterior se hace devolviendo valores. A continuación, un ejemplo de función usando `return`. Esta función también puede ser un ejemplo simple de una función con argumentos múltiples (dos o más):

```
>>> def suma (numero1, numero2):
...     return numero1 + numero2
...
>>> suma (1,16)
17
```

ILUSTRACIÓN 39. EJEMPLO DE ACCIÓN DE SENTENCIA RETURN

Funciones Avanzadas

Funciones de predicado

Las funciones de predicado no es más que una función la cual dice si algo es `True` o `False`, es decir, es una función que devuelve un tipo de datos booleano.

Funciones anónimas y funciones Lambda

Una función anónima, como su nombre indica es una función sin nombre. Es decir, es posible ejecutar una función sin referenciar un nombre, en Python puede ejecutar una función sin definirla con `def`. De hecho, son similares, pero con una diferencia fundamental: el contenido de una función anónima debe ser una única expresión en lugar de un bloque de acciones.

Las funciones anónimas se implementan en Python con las funciones o expresiones lambda, esta es unas de las funcionalidades más potentes de Python, pero a la vez es la más confusas para los principiantes. Más allá del sentido de función que se tiene hasta el momento, con su nombre y sus acciones internas, una función en su sentido más trivial significa realizar algo sobre algo. Por tanto, se podría decir que, mientras las funciones anónimas lambda sirven para realizar funciones simples, las funciones definidas con `def` sirven para manejar tareas más extensas. Cabe mencionar que las funciones lambda pueden expresar cualquier cosa que expresa una función normal, pero no a la inversa.

Por ejemplo, pensando en la dispensación de medicamentos en una farmacia u hospital, es viable llevar a cabo (de acuerdo con un inventario que se posea) una verificación de

seguridad de la petición que se realice: hay medicamentos que necesitan una receta debido a su uso indebido o riesgo a la salud pública. Puede crearse una función lambda, donde, al realizar la búsqueda de las presentaciones disponibles de un medicamento, se destaquen aquellos en donde es necesaria una receta (verificarla) y para surtirla, se requiera acceso a un almacenamiento distinto al de los demás medicamentos.

```python
#Formatos específicos:

destacar_valor = lambda control: " ¡ el/la {} es un medicamento controlado ! No surtir sin receta. ".format(control)

medicamento = input("Introduzca medicamento que buscará en inventario: ")

if medicamento == "clonazepam":
    print(destacar_valor(medicamento))

elif medicamento == "morfina":
    print(destacar_valor(medicamento))

elif medicamento == "fentanilo":
    print(destacar_valor(medicamento))

elif medicamento == "codeina":
    print(destacar_valor(medicamento))

else:
    print ("Presentaciones disponibles: ...")

```

ILUSTRACIÓN 40. EJEMPLO DE UTILIZACIÓN DE FUNCIONES LAMBDA

Funciones recursivas

Las funciones recursivas son funciones que se llaman a sí mismas durante su propia ejecución. Ellas funcionan de forma similar a las iteraciones, pero deben encargarse de planificar el momento en que dejan de llamarse a sí mismas o tendrá una función recursiva infinita. Estas funciones se suelen utilizar para dividir una tarea en subtareas más simples. Existen las funciones recursivas con retorno y sin este.

Funciones de orden superior

Las funciones de Python pueden tomar funciones como parámetros y devolver funciones como resultado. Una función que hace ambas cosas o alguna de ellas se llama función de orden superior. Un ejemplo de estas funciones pueden ser `filter()` y `map()`.

Funciones integradas

El intérprete de Python tiene un número de funciones integradas (built-in) dentro del módulo `__builtins__`, las cuales están siempre disponibles.

Estas funciones son abundantes y extensas, por lo que, se puede consultar a detalle su uso dentro del intérprete (**Ilustración 40**) utilizando el comando de ayuda `help()` o se puede consultar en la página oficial de Python donde se almacenan todos los documentos e información para el uso de las diferentes versiones del lenguaje en el siguiente enlace: https://docs.python.org/3/ y realizando una búsqueda acerca de "`builtins`".

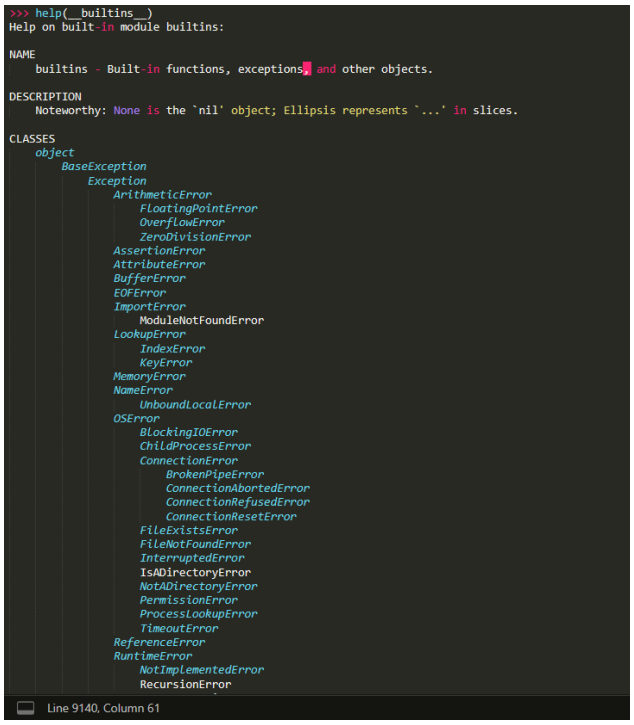


ILUSTRACIÓN 41. VISUALIZACIÓN DEL MÓDULO `__BUILTINS__`

Algunos ejemplos de las funciones más útiles y/o comunes que se pueden emplear en Python son:

Funciones generales

compille()

Devuelve un código objeto Python. Se usa la función integrada Python para convertir de la cadena de caracteres de código al código objeto.

```
>>> exec(compile('a=5\nb=7\nprint a+b','','exec'))

Output:
12
```

dir()

Si es llamado sin argumentos, devuelve los nombres en el ámbito actual. De lo contrario, devuelve una lista alfabética de nombres que comprende alguno(s) de los atributos de un objeto dato, y de los atributos legibles desde este.

```
>>> dir(__builtins__)
```

execfile()*

Lee y ejecuta un script Python desde un archivo. En Python 3 la alternativa para `execfile()` es:

```
>>> exec(open("./filename").read())
```

help()* El uso de esta función puede ser consultada en la sección de “Sintaxis Básica > Ayuda”

len()

Devuelve el número de elementos de un tipo de secuencia o colección.

```
>>> SARS_CoV_2_Seq =
"MESLVPGFNEKTHVQLSLPVLQVRDVLVRGFGDSVEEVLSEARQ"
>>> len(SARS_CoV_2_Seq)

Output:
44
```

license()

Imprime el texto de la licencia.

```
>>> license()

Output:
See https://www.python.org/psf/license/
```

open()*

La función `open()` es definida dentro del módulo integrado `io`, esta le permite abrir un archivo usando el tipo objeto `file`, devuelve un objeto del tipo `file` y se llama habitualmente con dos a tres argumentos.

*La función puede consultarse más detalladamente en la sección de "Archivos".

range()

Devuelve una lista conteniendo una progresión aritmética de enteros. La sintaxis general de la función es:

```
>>> range(inicio, detener[, paso])

Output: lista de enteros
```

type()*

Devuelve el tipo del objeto que recibe como argumento.

*Esta función la puede verse ejemplificada en "Sintaxis Básica > Variables y tipos de datos

Funciones de entrada y salida

input()

Lee una cadena de caracteres desde la entrada estándar.

```
>>> dato = input("Por favor, ingresa un dato: "); dato;
type(dato)

Input:
Por favor, ingresa un dato: "Este es un ejemplo simple"

Output:
'"Este es un ejemplo simple"'
<class 'str'>
```

raw_input()*

Cumple la función de `input()` en Python 2, si se introduce a la consola, éste ya no será identificado por el intérprete y arrojará un error indicando que el nombre de esa variable no ha sido definido:

```
>>> raw_input("Escribe un mensaje: ")

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

print()

Permite mostrar texto en pantalla. El texto para mostrar se escribe como argumento de la función.

```
>>> print("Facultad de Estudios Superiores Cuautitlán")

Output:
Facultad de Estudios Superiores Cuautitlán
```

Funciones numéricas

abs()

Devuelve el valor absoluto de un número (entero o de punto flotante).

```
>>> abs(12)

Output: 12

>>> abs(-3)

Output: 3

>>> abs(-2.5)

Output: 2.5
```

bin()

Devuelve una representación binaria de un número entero o entero long, es decir, lo convierte de entero a binario.

```
>>> bin(89)

Output:
'0b1011001'
```

cmp()

Devuelve un valor negativo si *x* < *y*, un valor cero si *x*==*y*, un valor positivo si *x* > *y* :

```
>>> cmp(1,2)

Output: -1

>>> cmp(2,2)

Output: 0

>>> cmp(2,1)

Output: 1
```

complex()

Devuelve un número complejo complex. Es un constructor, que crea un entero complex a partir de un entero, entero long, entero float (cadenas de caracteres formadas por números y hasta un punto), o una cadena de caracteres que sean coherentes con un número entero.

float()

Devuelve un número coma flotante. Es un constructor, que crea un punto flotante a partir de un entero, entero long, entero float (cadenas de caracteres formadas por números y hasta un punto) o una cadena de caracteres que sean coherentes con un número entero.

```
>>> float(8)

Output: 8.0

>>> float(678+8)

Output: 686.0

>>> float("243")

Output: 243.0
```

int()

Sólo procesa correctamente cadenas que contengan exclusivamente números. Si la cadena contiene cualquier otro carácter, la función devuelve una excepción *ValueError.*

```
>>> int(8.3642976723)
8
```

max()

Si recibe más de un argumento, devuelve el mayor de ellos. Si recibe un solo argumento, devuelve el mayor de sus elementos. Debe ser un objeto iterable (con la capacidad de repetirse): puede ser una cadena de caracteres, o alguno de los otros tipos de secuencia o colección.

```
>>> max(SARS_CoV_2_Seq)

Output: 'V'

>>> max(8,9,12.55,100,90328)

Output: 90328
```

min()

Funciona de manera opuesta a la función de `max()`.

```
>>> min(SARS_CoV_2_Seq)

Output:
'A'

>>> min(8,9,12.55,100,90328)

Output:
8
```

round()

Redondea un número flotante a una precisión dada en dígitos decimal (por defecto 0 dígitos). Esto siempre devuelve un número entero a diferencia de Python 2 que devuelve siempre un número de punto flotante.

```
>>> round(89.45)

Output:
89
```

Funciones booleanas

bool()

Es un constructor, el cual crea un tipo de datos booleanos, devuelve un tipo booleano `True` cuando el argumento dado es correcto, de lo contrario devolverá `False.` Se puede probar con valores enteros, tipo punto flotante y booleanos tal cual.

```
>>> bool()

Output: False

>>> bool(True)

Output: True

>>> bool(int(0.1))

Output: False
```

Funciones de cadenas de caracteres

capitalize()

Devuelve una cadena de caracteres con mayúscula la primera palabra.

```
>>> Universidad = "facultad de Estudios Superiores
Cuautitlán"
>>> Universidad.capitalize()

Output:
'Facultad de estudios superiores cuautitlán'
```

find()

Devuelve un valor numérico 0 si encuentra el criterio de búsqueda o -1 si no coincide el criterio de búsqueda enviado por parámetros en la función.

```
>>> "Escherichia coli".find("coli")

Output(será el valor donde comienza lo que se buscó):
12

>>> "Escherichia coli".find("aeruginosa")

Output:
-1
```

index()

Es como la función `find()` pero arroja una excepción *`ValueError`* cuando la sub-cadena no es encontrada.

```
>>> "Escherichia coli".index("aeruginosa")

Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

isalpha()

Devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres alfabéticos. Si posee espacios en blanco, estará incorrecto.

```
>>> "Manual De Python".isalpha()

Output: False #(Porque posee espacios en blanco)

>>> "Python".isalpha()

Output: True
```

isdigit()

Funciona de la misma manera que **`isalpha(),`** pero identifica caracteres con dígitos.

```
>>> "123456789".isdigit()

Output:
True
```

islower()

La función `islower()` devuelve un valor booleano `True` o `False` si coincide que la cadena contenga caracteres en minúsculas.

```
>>> 'farmacia'.islower()

Output: True

>>> 'Licenciatura en Farmacia'.islower()

Output: False
```

istitle()

Devuelve un valor booleano `True` o `False` si coincide que la cadena de caracteres sea capitales en cada palabra.

```
>>> "Manual de Python".title()

Output: 'Manual De Python'

>>> "Manual de Python".istitle()

Output: False

>>> "Manual De Python".istitle()

Output: True
```

lower()

Devuelve una cadena de caracteres con minúsculas en cada palabra.

```
>>> seq_Pseudomona = "ACCTACGTGCAACGTGAGATCTACGTACGAGT"
>>> seq_Pseudomona.lower()

Output:
'acctacgtgcaacgtgagatctacgtacgagt'
```

replace()

Si encuentra el criterio de la búsqueda de la sub-cadena o la remplaza con la nueva sub-cadena enviado por parámetros en la función.

```
>>> "Drogomultiresistencia".replace("Drogo", "Farmaco")

Output: 'Farmacomultiresistencia'
```

split()

Devuelve una lista con la cadena de caracteres separada por cada índice de la lista.

```
>>> aminoácidos = ("AUG, ATT, AGA, ATC, ATG".split())
>>> aminoácidos

Output:
['AUG,', 'ATT,', 'AGA,', 'ATC,', 'ATG']
```

str()

Es el constructor del tipo de cadenas de caracteres, se usa crear una carácter *o* cadenas de caracteres mediante la misma función `str()`.

```
>>> str(2345)

Output:
'2345'
```

upper()

Devuelve una cadena de caracteres con mayúsculas en cada palabra.

```
>>> DNASeq = input("Secuencia DNA: ").upper()

Input + Output 1:

Secuencia DNA: agcatcgactagcatacgcagcatttcag
#aqui se puede o no incluir las comillas, sin embargo,
identificará la secuencia de igual manera#

>>> DNASeq #Llamado de la variable#

Output 2:
'AGCATCGACTAGCATACGCAGCATTTCAG'
```

La secuencia que se observa es el input, seleccionado o creado por el usuario.

Funciones de secuencias

dict()

Es el constructor del tipo de diccionario, esta función se usa crear un diccionario:

```
>>> dic(python = 2.7, zope = 2.13, plone = 5.1)

Output:
{'zope': 2.13, 'python': 2.7, 'plone': 5.1}
```

next()

Devuelve el próximo elemento desde un iterador.

```
>>> elemento = iter([1,2,3,4,5])
>>> next(elemento)
1
>>> next(elemento)
2
```

list()

Es el constructor del tipo de lista, se usa crear una lista mediante la misma función `list()` de un iterable.

```
>>> TATAbox = "TATAAA"
>>> list(TATAbox)

Output:
['T', 'A', 'T', 'A', 'A', 'A']
```

tuple()

Es el constructor del tipo de tuplas, se usa crear una tupla mediante la misma función `tuple()` de un iterable.

```
>>> tupple(TATAbox)

Otput:
('T', 'A', 'T', 'A', 'A', 'A')
```

set()

Es el constructor del tipo de conjuntos, se usa crear un conjunto mutable mediante la misma función `set()` de un objeto iterable lista:

```
>>> >>> genes_resistencia = ["blaTEM", "blaSHV", "blaCARB",
"blaOXA", "blaCTX-M", "blaGE"]
>>> genes_resistencia

Otput:
['blaTEM', 'blaSHV', 'blaCARB', 'blaOXA', 'blaCTX-M',
'blaGE']
>>> genes_resistencia_set = set(genes_resistencia)
>>> genes_resistencia_set

Output:
{'blaCARB', 'blaSHV', 'blaTEM', 'blaGE', 'blaCTX-M',
'blaOXA'}
```

zip()

En Python 2 La función devuelve una lista de tuplas, donde cada tupla contiene el elemento desde cada uno de los tipos de secuencias de argumento. La lista devuelta es truncada en longitud a la longitud de la secuencia de argumentos más corta. En Python 3 devolverá una ubicación del archivo:

```
>>> zip(['python', 'biopython', 'django'], [2.7, 2.13,
5.1])

#Python 2 Output: [('python', 2.7), ('biopython', 2.13),
('django', 5.1)]

#Python 3 Output: <zip object at 0x000001E7B8C675C0>
```

Funciones de objetos

isinstance()

Permite corroborar si un objeto es una instancia de una clase. La sintaxis general es:

```
>>> isinstance(objeto, tipo)
```

Esta función devuelve `True` si el objeto especificado es del tipo especificado, de lo contrario `False`. Los parámetros son:

- **Objeto:** un objeto (como una variable) es requerido.
- **Tipo:** un tipo o una clase, o una tupla de tipos y/o clases.

```
>>> persona1 = Persona("V-13458796", "Leonardo",
"Caballero", "M")
>>> isinstance(persona1, Persona)

Output:
True
```

Ejemplo práctico:

```
>>> def carga_prot(AAseq):

    """ Retorna la carga de una proteína """

    protseq = AAseq.upper()
    carga = -0.002
    AACarga = {"C":-.045,"D":-.999,"E":
.998,"H":.091,"K":1,"R":1,"Y":-.001}

    for aa in protseq:
        carga += AACarga.get(aa,0)
    return carga
#Para usar la función se debe llamar con el parámetro
siguiente (elección nuestra):

>>> carga_prot("QTALLVVLVLLAVALQATEAGPYGA")

Output: -1.001
>>> carga_prot("EEARGPLRGKGDQKSAVSQKPRSRGILH")

Output:4.094
```

Si no se indican los parámetros al llamar a la función, entonces, ocurrirá lo siguiente:

```
>>> carga_prot()

Output:
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
Carga_prot()
TypeError: carga_prot() takes exactly 1 argument (0 given)
```

En este ejemplo, la función devuelve un número (de tipo punto flotante).

Strings

Son un tipo de secuencia de símbolos delimitados por quotes (´), doble quote(“) y triple quote doble (“ “ “), por ejemplo:

```
>>> “Este es un string en Python”
```

Es importante conocer que sirven para diferenciar comillas dentro de frases con comillas:

```
>>> “Mezclar comillas es pasar al lado obscuro con
posibilidad de errores´

Output:
  File "<stdin>", line 1
    “Mezclar comillas es pasar al lado obscuro con
posibilidad de errores´
            ^
SyntaxError: invalid character in identifier
```

Las triples comillas funcionan para crear strings en bloques con ayuda de un salto de línea \n en el código. En Python 3 los strings son Unicode por defecto, lo que significa que soportan caracteres extraños como la ñ, acentos o tildes, símbolos, etc. Pero no hay que olvidar que mientras más sencillas y universales sean los datos que se manejan en general, mejor para el código y los usuarios.

Propiedades comunes de las secuencias de caracteres o strings

Dado que las secuencias comparten propiedades comunes, se pueden aplicar a diversos tipos de datos tales como listas, tuplas, etc. A continuación, se listan brevemente algunas de las propiedades comunes de los strings con ejemplos sencillos para poder comprenderlo mejor:

Indexing

La indexación se volverá a ver en las listas, pero también se incluirá aquí. Dado que los elementos de las secuencias están ordenados, se podrá tener acceso a cualquier elemento a través de un índice que comienza en cero:

```
>>> point = (23,56,11)
>>> point[0]

Output: 23

>>> sec_prot = "MRVLLVALALLALAASATS"
>>> sec_prot[0]

Output:'M'

>>> parámetros = ['UniGene','dna','Mm.248907',5]
>>> parametros[2]

Output: 'Mm.248907'
```

También se puede tener acceso indicando números negativos:

```
>>> point[-1]

Output: 11

>>> mi_secuencia[-4]

Output: 'S'
```

Para acceder a un elemento que está dentro de una secuencia, que a su vez está dentro de otra secuencia, debe usar otro índice.

```
>>> secdatos = ("MRVLLVALALLA",12,"5FE9EEE8EE2DC2C7")
>>> secdatos[0][5]

Output: 'V'
```

Aquí se indica que entre al primer elemento o elemento 0 de los elementos de la tupla y, que de ese elemento se indique qué contiene en la quinta posición (sin olvidar que Python comienza el conteo desde 0).

Rebanar o cortar

Se puede seleccionar una parte de una secuencia mediante la notación de sectores. "Rebanar" consiste en usar dos índices separados por dos puntos (:). Estos índices representan una posición en el espacio existente entre los elementos. La cadena "Python" se puede representar como:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
```

Dentro del intérprete se puede ver de la siguiente manera:

```
>>> nombre = "Python"
>>> nombre[0:2]

Output: 'Py'
```

Al omitir el primer subíndice, el valor del índice se predetermina a la primera posición (0). Por otro lado, cuando se omite el segundo subíndice, el valor del índice se dirige por defecto a la última posición (-1):

```
>>> nombre[4:]

Output: 'on'
```

Hay un tercer índice opcional para omitir posiciones (argumento de paso):

```
>>> nombre[1:5]

Output: 'ytho'

>>> nombre[1:5:2]

Output: 'yh'
```

Un paso con un número negativo se usa para contar hacia atrás. Entonces -1 (en la tercera posición) se puede usar para invertir una secuencia:

```
>>> nombre[::-1]

Output:'nohtyP'
```

Prueba de pertenencia

Se puede verificar si un elemento pertenece a una secuencia utilizando este método. La estructura será `dato_buscado` **`in`** `objeto_de_búsqueda`:

```
>>> point = (23,56,11)
>>> 11 in point

Output: True

>>> Sec_prot = "MRVLLVALALLALAASATS"
>>> "X" in Sec_prot

Output: False
```

Concatenación

Se pueden concatenar dos o más secuencias de la misma clase usando el signo "+":

```
>>> point = (23,56,11)
>>> point2 = (2,6,7)
>>> point + point2

Output:
(23,56,11,2,6,7)

>>> Sec_DNA = "ATGCTAGACGTCCTCAGATAGCCG"
>>> TATAbox = "TATAAA"
>>> TATAbox + Sec_DNA

Output:
'TATAAAATGCTAGACGTCCTCAGATAGCCG'
```

Secuencias de diferentes tipos de datos no pueden ser concatenados:

```
>>> point + TATAbox

Output:
Traceback (most recent call last):
File "<pyshell#48>", line 1, in <module>
point + TATAbox
TypeError: can only concatenate tuple (not "str") to tuple
```

Capítulo III. Listas

Las listas son uno de los objetos más versátiles de Python. Una lista es una colección de objetos ordenada. Está representado por elementos separados por comas y encerrado entre corchetes.

```
>>> mi_lista = [“a”, “b”,…]
```

```
mi_lista = [“a”, “b”,…]
```

Ya se ha observado un ejemplo de creación de una lista al emplear el comando `split()`. Esta es una lista con 7 elementos y todos son de tipo string.:

```
>>> "python, guia, curso, tutorial, bioinformática, fesc,
SARS-CoV-2".split()

Output:
‘['python,', 'guia,', 'curso,', 'tutorial,',
'bioinformática,', 'fesc,', 'SARS-CoV-2']’
```

Para crear una lista, lo único que hay que hacer es elegir un nombre y colocar entre corchetes los elementos que esta incluirá. Las listas pueden poseer elementos de diferente o del mismo tipo de datos:

```
>>> Primera_lista = [1,2,3,4,5]
>>> Segunda_lista = [“SARS-CoV-2”, 2019, “SARS”, “MERS”,
2020]
```

Las listas también pueden contener, a su vez, otras listas (listas anidadas) o pueden ir vacías. Las listas vacías no tienen ninguna funcionalidad por sí mismas, sin embargo, son útiles si se quiere guardar en ellas contenido más tarde:

```
>>> Lista_anidada = [1,2,3,”Primera_lista”,4,5,“MERS”]
>>> Lista_vacia = []
```

Inicialización de listas

Si se sabe de antemano que una lista que va a ser creada poseerá cierta cantidad de elementos, entonces, se pueden inicializar con valores predeterminados:

```
>>> Codones = [None] * 5
>>> Codones

Output:
[None, None, None, None, None]
```

Este tipo de inicialización de listas puede ser útil cuando se trabaja con listas grandes y el número de elementos se conoce con anterioridad. Definir una lista con un tamaño fijo es más eficiente que crear una lista vacía expandiéndola según sea necesario. Fijar el tamaño de las listas no tienen la sobrecarga de listas que cambian posiciones en la memoria.

Acceder a elementos de una lista

Como uno de los otros tipos de datos de secuencia de caracteres, se puede acceder a los elementos de la lista mediante un índice a partir de cero.

```
>>> First_list = [A,T,G,U,C]
>>> First_list[0]
>>> First_list[2]

Output 1: A            Output 2: G
```

También se puede acceder a las listas desde la derecha (o final de la lista) empleando números negativos:

```
>>> First_list=[A,T,G,U,C]
>>> First_list[-1]
>>> First_list[-2]

Output 1: C              Output 2: U
```

Otra manera de obtener listas es transformar un objeto que no es una lista en una nueva lista usando la función `list()`:

```
>>> Seq_SARS_CoV_2 = "CTACTAGTGGTAGATGGGTA"
>>> list(Seq_SARS_CoV_2)

Output:
['C', 'T', 'A', 'C', 'T', 'A', 'G', 'T', 'G', 'G', 'T', 'A',
'G', 'A', 'T', 'G', 'G', 'G', 'T', 'A']
```

Para regresar la lista a su forma original de string se debe de utilizar la función `join()` como alternativa a la función `str()`, ya que, ésta la convierte en un string, pero no en el string original.

Modificando listas

A diferencia de los strings, las listas pueden modificarse agregando, eliminando o cambiando los elementos que en ellas existen:

Añadiendo/añadir:

Existen 3 maneras de añadir elementos en una lista, empleando las funciones: `append`, `insert` y `extend`.

append ()

Añade un elemento al final de la lista:

```
>>> Lista = [1,2,3,4,5]
>>> Lista.append(99)
>>> Lista

Output:
[1, 2, 3, 4, 5, 99]
```

insert ()

Inserta un elemento en la posición indicada:

```
>>> Lista = [1,2,3,4,5]
>>> Lista.insert(4, "adicional")
>>> Lista

Output:
[1, 2, 3, 4, 'adicional', 5]
```

extend()

Extiende una lista al añadir elementos al final de la lista original:

```
>>> Lista = [1,2,3,4,5]
>>> Lista.extend([6,7,8,9,10])
>>> Lista

Output:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Eliminar

También se pueden eliminar elementos de una lista.

pop():

Remueve o elimina el elemento en la posición del index y regresa al punto donde fue llamado. Sin parámetros lo que regresará será el último elemento:

```
>>> Lista = [1,2,3,4,5]
>>> Lista.pop()
>>> Lista

Output: [1, 2, 3, 4]

>>> Lista.pop(0)
>>> Lista

Output:[2,3,4]
```

`remove():`

Elimina el elemento especificado en el parámetro. En el caso donde hay más de una copia del mismo objeto en la lista, elimina el primero, contando desde la izquierda. A diferencia de `pop()`, esta función no devuelve nada por lo que debe llamarse la lista nuevamente a consola.

```
>>> Lista = [1,2,3,4,5]
>>> Lista.remove(1)
>>>Lista

Output:[2,3,4,5]
```

Si se intenta eliminar un elemento que no existe, la consola arrojará un error.

Capítulo IV. Tuplas

Las tuplas son listas inmutables. La característica principal de la tupla es que una vez creada, no se puede modificar. Es por eso por lo que se les conoce como "listas inmutables". Los objetos de Python a veces se dividen en mutable e inmutable. Como su nombre lo indica, los objetos inmutables no pueden ser modificados después de que se crean. Se pueden distinguir fácilmente una tupla de una lista porque los elementos de la tupla están encerrados entre paréntesis en lugar de corchetes. Para crear tuplas de un solo elemento hay que incluir una coma después del elemento.

```
>>> mi_tupla = ("a",”b”,…)
```

```
mi_tupla = ("a",”b”,…)
```

Por ejemplo:

```
>>> Tupla = (“AUG”, “ACG”, “ATC”)
>>> Tupla_unica = (99,)
```

Con la coma final y entre paréntesis, está claro que es una tupla y no una expresión. No está permitido agregar o eliminar elementos de una tupla por ser un objeto inmutable como se mencionó antes. Al intentarlo con un método empleado en las listas, arrojará un error:

```
>>> Tupla

Output:('AUG', 'ACG', 'ATC')
>>> Tuple.append("ACT")

Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Si las tuplas no pueden modificarse y resultan ser un tanto más complejas ¿para qué usarlas? existen diferencias conceptuales de los datos almacenados como una lista y como una tupla:

las listas deben de contener una cantidad de objetos que es variable y todos estos objetos son de un mismo tipo donde el orden que estos posean no es de importancia y puede ser alterado en cualquier momento. Una tupla puede contener elementos que pueden ser diferenciados tales como datos de coordenadas o un diccionario en donde los elementos de la tupla poseen un orden específico que no cambia debido a que están ligadas a su posición inicial para, de esa manera, seguir su funcionalidad. Otra de las ventajas de las tuplas es que son muy útiles para trabajar de manera segura: las tuplas son inmutables y no pueden ser modificadas. La velocidad de procesamiento de las tuplas también es mayor a la de una lista, por lo que se puede tomar en cuenta al momento de elegir la modalidad de trabajo.

Métodos asociados a las tuplas

Indexación

Al igual que en las listas, también puede usarse la función de `index` en las tuplas. Como los elementos en las secuencias están ordenados, se puede ganar acceso a cualquier elemento a través de un índice que comienza en cero. Para acceder a un elemento que está dentro de una secuencia que a su vez está dentro de otra secuencia, se necesita emplear otra función de indexación adicional como se vio anteriormente:

```
>>> Secuencia_datos = ("MRVLLVALALLA",12,"5FE9EEE8EE2DC2C7")
>>> Secuencia_datos [0][6]

Output: 'A'
```

Contar

Este método recibe un elemento como argumento, y cuenta la cantidad de veces que aparece en la tupla. Se debe tomar en cuenta que es indispensable incluir el argumento completo y no sólo una fracción de este, si se comete ese error el resultado será igual a 0.

```
>>> Secuencia_SARSCOV2 = ("AUG", "ATC", "TAT")
>>> Secuencia_SARSCOV2.count("AUG")

Output: 1
```

Capítulo V. Diccionarios

Mientras que las listas, tuplas y cadenas son tipos de datos ordenados o secuenciales, los conjuntos y diccionarios son contenedores de datos desordenados. Los diccionarios, también conocidos como matrices asociativas o hashes en otros lenguajes comunes, constan de pares de palabras clave y su valor asociado, lo cual es indicado de la siguiente manera entre llaves: `{palabra_clave: valor}`

Son estructuras de datos particularmente útiles porque a diferencia de las listas y tuplas, los valores no están restringidos a ser indexados únicamente por los números enteros correspondientes a la posición secuencial en la serie de datos. Más bien, las claves de un diccionario sirven como índice y pueden ser de cualquier tipo de datos inmutables (strings, listas, números o tuplas de datos inmutables). La sintaxis básica de un diccionario es:

```
>>> mi_diccionario = {""}
```

Por ejemplo, se puede crear un diccionario sobre aminoácidos con las abreviaturas de 3 letras e incluir su peso molecular para posteriores aplicaciones:

```
>>> aminoácidos = {"ala" :( "a", "alanina", 89.1), "cys" :(
"c", "cisteína", 121.2)}
```

Una opción adicional para crear un diccionario a partir de otros objetos como una lista o una tupla puede ser el método `dict.fromkeys()`. Los valores que se asignarán por defecto a cada uno de los elementos clave será `None`, el cual puede modificarse posteriormente con un método que será visto un poco más adelante. Si ya se han asignado valores a los elementos clave, únicamente, se debe de emplear la sentencia `dict.fromkeys(objeto_a_convertir)`:

```
>>> secuencias_virus = ("Ébola", "Hanta", "Lassa",
"Dengue")
>>> dicc_sec_vir = dict.fromkeys(secuencias_virus)
>>> dicc_sec_vir

Output:
{'Ébola': None, 'Hanta': None, 'Lassa': None, 'Dengue':
None}
```

Si se desea acceder a los elementos de un diccionario mediante corchetes, de forma análoga a una tupla o lista, se puede emplear el método siguiente, por ejemplo:

```
>>> aminoacidos ["ala"]

Output:
("a", "alanina", 89.1)
```

Adición o corrección de elementos

Se pueden añadir elementos nuevos a diccionarios que ya existen de la siguiente manera:

```
>>> aminoácidos ["gly"] = ("g", "glicina", 75.07)
```

También se puede corregir un elemento para modificar la información que contiene. Para verificar el cambio y la adición, únicamente se deberá llamar al diccionario a consola de la siguiente manera:

```
>>> aminoacidos

Output:
{'ala': ('a', 'alanina', 89.1), 'cys': ('c', ' cisteína ',
121, 2), 'gly': ('g', 'glicina', 75.07)}
```

Como otro ejemplo muy similar al anterior y propuesta de ejercicio, los diccionarios se pueden utilizar para crear tablas de búsqueda de las propiedades de una colección de proteínas estrechamente relacionadas. Cada clave podría establecerse en un identificador

único para cada proteína, como su ID de UniProt (por ejemplo: P0DTC1), y los valores correspondientes podrían ser una estructura de datos de tupla que contiene el punto isoeléctrico de la proteína, el peso molecular, el código de acceso a PDB (si existe una estructura), y así sucesivamente. Si se desea consultar la página de UniProt puede acceder al siguiente enlace: https://www.uniprot.org/

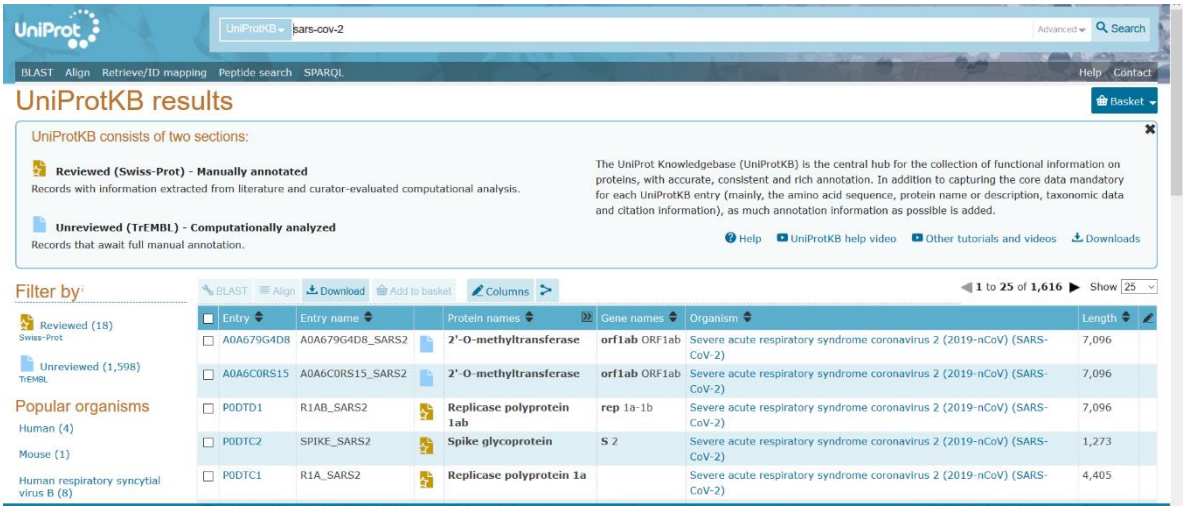


ILUSTRACIÓN 42. PÁGINA DE RESULTADOS DE BÚSQUEDA DE UNIPROT

Eliminar elementos

Para eliminar un elemento se puede emplear la función `del` de la siguiente manera y verificar como en el ejemplo anterior:

```
>>> del aminoacidos ["gly"]
```

Una alternativa es utilizando el método `.pop()` :

```
>>>dicc_sec_vir.pop("Ébola")
```

Si se quieren eliminar todos los elementos del diccionario se aplica el método `clear()`:

```
>>> aminoacidos.clear()
```

Consultar los valores

Para verificar la cantidad de elementos claves dentro del diccionario se utiliza el comando `.keys()` a continuación del nombre del diccionario:

```
>>> aminoácidos.keys()

Output:
dict_keys(['ala', 'cys'])
```

Consultar las palabras clave

Funciona de manera similar el verificar los valores o argumentos dados a los elementos clave. Se debe de adicionar el comando `.values()` a continuación del nombre del diccionario:

```
>>> aminoácidos.values()

Output:
dict_values([('a', 'alanine', 89.1), ('c', ' cisteína ',
121, 2)])
```

Cantidad de elementos de un diccionario/longitud del diccionario

Para saber el número de elementos que contiene un diccionario (elementos clave) se puede lograr de una manera muy simple, introduciendo a la consola lo siguiente:

```
>>> len(aminoacidos)

Output: 2
```

Comparación de diccionarios

La función `cmp()` ese usa en Python para comparar valores y claves de dos diccionarios. La función devuelve el valor 0 si ambos diccionarios son iguales, devuelve el valor 1 si el primer diccionario es mayor que el segundo diccionario y devuelve el valor -1 si el primer diccionario es menor que el segundo diccionario.

```
>>> cmp(diccionario_1, diccionario_2)
```

Sin embargo, este método es propio de Python 2, por lo que en Python 3 arrojará un error. Si se quieren comparar diccionarios en Python 3 puede hacerse uso de los sets o conjuntos, los cuales son el tema siguiente.

Para conocer si alguno de los diccionarios en uso o creados tienen algún término en común, se debe aplicar el siguiente método de comprensión de diccionarios, donde el resultado es un tercer diccionario en caso de poseer términos en común (el nuevo diccionario poseerá esos términos):

```
>>> dic1{elementos}
>>> dic2{elementos}
>>> dic3 = {k:v for (k,v) in dic2() if k in dic1}
```

Capítulo VI. Sets o conjuntos

Los sets son un tipo de datos que no se encuentra comúnmente en otros lenguajes de programación. Incluso en Python, los conjuntos no son muy populares, ya que, se implementaron como tipo de datos nativo recientemente. Un conjunto o un set es una estructura que se encuentra con frecuencia en matemáticas. Es similar a una lista, con dos diferencias destacadas: los elementos no conservan un orden implícito y cada elemento es único. Los usos más comunes de los conjuntos son pruebas de pertenencia, eliminación de duplicados, y la aplicación de operaciones matemáticas: intersecciones, uniones, diferencias, y diferencias simétricas.

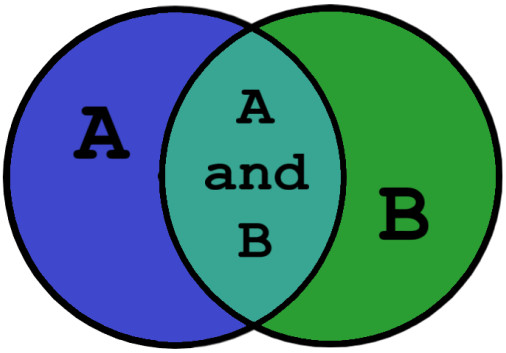


ILUSTRACIÓN 43. REPRESENTACIÓN GRÁFICA DE CONJUNTOS

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves {}, o bien usar el constructor de la clase `set()` y pasar como argumento un objeto iterable (que cuenta con capacidad de repetirse) como una lista, una tupla, una cadena, etc.

```
# Crea un conjunto con una serie de elementos entre llaves
# Los elementos repetidos se eliminan, los espacios cuentan
>>> c = {1, 3, 2, 9, 3, 1}
>>> c

Output:
{1, 2, 3, 9}

# Crear un conjunto a partir de un string
>>> FAC = set('FES Cuautitlán')
>>> FAC

Output:
{'S', 'l', 'i', 'a', ' ', 'E', 'F', 'C', 'á', 'n', 't', 'u'}

# Crea un conjunto a partir de una lista
>>> unicos = set([3, 5, 6, 1, 5])
>>> únicos

Output:
{1, 3, 5, 6}
```

Algunas de las funciones que pueden aplicarse para los sets son: `len()`, `max()` y `min()`, `discard()`, `remove()`, `pop()`, etc.

Capítulo VII. Control de flujo

`if-else` y `elif`

En Python, la sentencia `if` se utiliza para ejecutar un bloque de código si, y solo si se cumple una determinada condición. Por tanto, `if` es usado para la toma de decisiones. La estructura básica de esta sentencia `if` es la siguiente:

```
if condición:
    Bloque de código
```

Es decir, sólo si la condición es verdadera o es cumplida, se ejecutarán las sentencias que forman parte del bloque de código. La condición puede ser un literal, el valor, el valor de una variable, el resultado de una expresión, el valor devuelto por una función, etc. Hay ocasiones en que la sentencia `if` básica no es suficiente y es necesario ejecutar un conjunto de instrucciones o sentencias cuando la condición se evalúa a `False.` Para ello se utiliza la estructura `if ... else...` Esta es estructura es:

```
if condición:
    bloque de código (cuando condición se evalúa a True)
else:
    bloque de código 2 (cuando condición se evalúa a False)
```

También es posible que se den situaciones en que una decisión dependa de más de una condición. En estos casos se usa una sentencia `if` compuesta, cuya estructura es como se indica a continuación:

```
if cond1:
    bloque cond1 (sentencias si se evalúa la cond1 a True)
elif cond2:
    bloque cond2 (sentencias si cond1 es False pero cond2 es
True)
   ...
else:
    bloque else (sentencias si todas las condiciones anteriores
se evalúan a False)
```

Ejemplos

Para comenzar a trabajar con control de flujo de programación puede resultar más cómodo y práctico el trabajar en el editor de texto y, posteriormente, ejecutar el REPL. Importante recordar guardar el archivo con extensión .py y una vez terminado el código que se está trabajando, guardar con ctrl + s y ejecutar con la combinación ctrl + alt + b.

Como primer ejemplo se trabajará con la sentencia `if` y `else`. Este programa será para conocer la nomenclatura de 3 letras de los aminoácidos a partir de la nomenclatura de 1 sola letra. Se elaborará un diccionario pequeño con aminoácidos, posteriormente será necesario solicitar la introducción de un aminoácido en formato de una sola letra para procesar lo introducido, brindar la respuesta y, en caso de no tener la información hacerlo saber al usuario.

```
trans = {"A":"Ala","N":"Asn","D":"Asp","C":"Cys"}
aa = input("Introduzca una letra correspondiente a un
aminoácido: ")

if aa in trans:
    print("La nomenclatura de 3 letras correspondiente a
"+aa+" es: "+trans[aa])
else:
    print("Lo siento, esa letra (aminoácido) no se
encuentra en el diccionario")
```

Se pueden utilizar tantas condiciones `elif` como desee evaluar. Se deberá tener en cuenta que una vez que una condición se evalúa como verdadera, las condiciones restantes no serán evaluadas. El siguiente ejemplo hará uso de la instrucción `elif:`

```
ADN = input("Introduzca su secuencia de DNA/ADN: ")
seqtamaño = len(ADN)
if seqtamaño < 10:
     print("El primer debe tener por lo menos 10
nucleótidos")
elif seqtamaño < 25:
     print("El tamaño es correcto")
else:
     print("El primer es demasiado largo")
```

Este programa solicita una secuencia de ADN ingresada con el teclado en tiempo de ejecución. Esta secuencia se llama “ADN”. En la línea 2 se calcula su tamaño y este resultado está vinculado al nombre `seqtamaño`. En la línea 3 hay una evaluación. Si `seqtamaño` es inferior a diez, el mensaje "El primer debe tener al menos diez nucleótidos" será impreso o presentado en pantalla. El programa fluye al final de esta sentencia `if`, sin evaluar cualquier otra condición en esta declaración `if`. Pero si no es cierto (por ejemplo, si la longitud de la secuencia fuera 15), ejecutaría la siguiente condición y su bloque asociado en caso de que esta condición se evalúe como verdadera. Si a longitud de la secuencia fuera de un valor superior a 10, el programa omitiría la línea 4 (el bloque de código asociado con la primera condición) y evaluaría la expresión en la línea 5. Si se cumple esta condición, se imprimirá "Este tamaño es correcto". Si no hay ninguna expresión que se evalúe como verdadera, se ejecuta el bloque `else`.

Condicionales anidados

Una sentencia condicional puede contener a su vez otra sentencia dentro de ella, esto es lo que se conoce como condicional anidada(o). Un ejemplo de esto puede verse a continuación, empleando el ejemplo anterior como base para el nuevo script:

```
DNA = input("Introduzca su secuencia de DNA/ADN: ")
seqtamaño = len(DNA)

if seqtamaño < 10:
     print("El primer debe tener por lo menos 10
nucleótidos")

     if seqtamaño==0:
          print("Por favor, introduzca una secuencia.")

elif seqtamaño < 25:
     print("El tamaño es correcto")

else:
     print("El primer es demasiado largo")
```

Se puede observar que en la línea 7 se introduce un condicional `if` dentro de otro y, además se verifica que exista introducción de datos para advertir al usuario de no dejar el dato en blanco.

for Loop (bucle **for**)

El bucle `for` es una estructura de control permite que el código se ejecute repetidamente mientras se mantiene una variable con el valor de un objeto iterable. La sintaxis general es:

```
for variable in iterable:
     bloque de código
```

Hay que tener en cuenta los dos puntos al final: son obligatorios al igual que la Identación para el código que le sigue al mismo. Esta estructura resulta en la repetición del bloque de código tantas veces como elementos estén en el iterable objeto. En cada iteración, la variable toma el valor del elemento actual en la iterable. En el siguiente código para recorrer una lista (bases nucleotídicas) con cuatro elementos. En cada iteración, "*x*" toma el valor de uno de los elementos de la lista:

```
bases = ["C","T","G","A"]
for x in bases:
     print(x)


Output:
C
T
G
A
```

También se puede ejecutar un pequeño script sobre mutaciones:

```
mutacion = False
for i in
"AACTCTACTCTCATCACAACTCATACTACTACCCATCTATTACATTACTAAACTATCT
ATCAAACCCTA":
    if(i == "ATCCCT"):
     mutacion = True

if mutacion == True:
     print("Mutacion")
else:
     print("Sin mutacion")
```

Para poder introducir la secuencia propia se podría añadir: `Secuencia = input("Introduce tu secuencia: ")` antes del bucle `for` e incluir la comparativa dentro de la condición `if`. ¿Puedes mejorarlo? ¡Inténtalo!

`while` Loop (bucle **`while`**)

La sentencia o bucle `while` en Python es una sentencia de control de flujo que se utiliza para ejecutar un bloque de instrucciones de forma continuada mientras se cumpla una condición determinada. La sintaxis básica de un bucle `while` es:

```
while condición:
     bloque de código
```

Es muy importante tener en cuenta que debe haber una instrucción dentro del bloque para hacer que la condición `while` sea falsa. De lo contrario, se entrará en un bucle infinito. Un ejemplo sencillo:

```
>>> a = 40
>>> while a < 100:
     print(a)
     a = a+10
... ... ...

Output:
40
50
60
70
80
90
```

`Pass, continue, break` y excepciones

`pass:` A veces no hay necesidad de una elección alternativa en una declaración `if`, en este caso simplemente puede evitarse o pasarse por alto. Para que el mismo código sea más legible, Python proporciona la declaración de paso o `pass`. Esta declaración es como un marcador de posición, tiene cualquier otro propósito que poner algo cuando se requiere una declaración sintácticamente.

`continue`: la sentencia `continue` cuando se ejecuta, obliga a Python a dejar de ejecutar el código que haya dentro del bucle y a iniciar una nueva iteración (es decir, a volver al comienzo del bucle y seguir con la ejecución del programa). Por ejemplo, cuando se indica el imprimir en pantalla una lista de números hasta el 100 y, en caso de ser el número 5, ignorar la instrucción de imprimirlo y continuar la instrucción hasta completar:

```
>>> for n in range(100):
...  if n == 5:
...       continue
...  print(n)
...

Output:
0
1
2
3
4
6
7
8
9

#y continua hasta el 100 para este ejemplo#
```

Excepciones

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservada, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa. En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias `try`, `except` y `finally`.

`try:` dentro del bloque `try` se ubica todo el código que pueda llegar a levantar una excepción, se utiliza el término levantar para referirse a la acción de generar una excepción.

`except:` éste ayudará a especificar el error que se puede tener y para el cual se generará la excepción. Puede verse una sintaxis general de una excepción a continuación.

`finally:` esta sentencia siempre será ejecutada antes de salir de la sentencia `try`, ya sea que una excepción haya ocurrido o no. Cuando ocurre una excepción en la sentencia `try` y no fue manejada por una sentencia `except` (u ocurrió en una sentencia `else`), es

relanzada luego de que se ejecuta la sentencia `finally`. La sentencia `finally` es también ejecutada "a la salida" cuando cualquier otra sentencia de la sentencia `try` es dejada vía `break`, `continue` o `return`.

```
try:
    # aquí colocar el código que puede lanzar excepciones
except Error:
    # entrará aquí en caso de que se haya producido una excepción de Error especificado
except:
    # entrará aquí en caso de que se haya producido una excepción que no corresponda a ninguno de los tipos especificados en los except previos
finally:
    #acciones finales, usualmente de limpieza. Este bloque SIEMPRE se va a ejecutar, haya surgido la excepción o no
```

`break`: es una sentencia que se pueden usar en bucles `for` y `while` y, simplemente, terminar el bucle actual (salir) y continuar con la ejecución de la siguiente instrucción, por ejemplo, en el siguiente script.

Se trata de un programa de operaciones matemáticas simples tales como suma, resta, multiplicación y división. En este caso, en la primera parte del código son definidas las funciones para cada una de las operaciones aritméticas, donde se especifica en los parámetros que se deben de introducir 2 números para poder realizar la operación. Adicional a la función de división, se aprecia un manejo de excepción en donde no se permitirá realizar una división entre cero. Además, se deberá verificar que ambos valores sean numéricos, por lo que, dentro del bucle `while` se intentará leer ambos valores, en caso de no poder obtener valores numéricos el bucle utilizará la sentencia `break` para emplear la opción de `ValueError` y así mostrar un mensaje que nos otorgue la instrucción de introducir datos correctos. Este bucle se repetirá hasta que se logre introducir datos válidos gracias a la inclusión de la excepción (`except`) la cual nos devuelve al inicio del bucle hasta que se cumpla la condición que se está buscando para el programa que acaba de ser creado. Los valores para introducir serán libres, así como la elección de la operación que se va a realizar,

por lo que se debe de recordar que al ejecutar el programa hay que llamar a la función en el input (operación en este caso) que se quiera realizar. Es posible observar lo anterior en la Ilustración 44.

```
def suma(num1, num2):
    return num1+num2

def resta(num1, num2):
    return num1-num2

def multiplica(num1, num2):
    return num1*num2

def divide(num1,num2):
    try:
        return num1/num2

    except ZeroDivisionError:
        print("No se puede dividir entre cero")
        return "Operación errónea"

while True:
    try:
        op1=(int(input("Introduce el primer número: ")))

        op2=(int(input("Introduce el segundo número: ")))

        break

    except ValueError:
        print("Los valores introducidos no son correctos. Por favor introduce valores numéricos.")


operacion=input("Introduce la operación a realizar (suma,resta,multiplica,divide): ")

if operacion=="suma":
    print(suma(op1,op2))

elif operacion=="resta":
    print(resta(op1,op2))

elif operacion=="multiplica":
    print(multiplica(op1,op2))

elif operacion=="divide":
    print(divide(op1,op2))

else:
    print ("Operación no contemplada")


print("Operación ejecutada. Continuación de ejecución del programa ")
```

ILUSTRACIÓN 44. ESTRUCTURA DE PROGRAMA PARA OPERACIONES MATEMÁTICAS

Aplicaciones interesantes (con ejemplos)

Estimar la carga neta de una proteína:

A un pH fijo, es posible calcular la carga neta de una proteína sumando la carga de sus aminoácidos individuales. Ésta es una aproximación, ya que, no tiene en cuenta si los aminoácidos están expuestos o inmersos en la estructura proteica. Este script tampoco tiene en cuenta el hecho de que la cisteína agrega carga solo cuando no es parte de un puente disulfuro (¿te gustaría intentar modificar el script para tomar esta consideración? ¡Inténtalo!). Ya que es un valor aproximado, el valor obtenido debe considerarse únicamente como una estimación.

Este ejemplo será trabajado desde el editor de texto.

```
protseq = input("Introduzca la secuencia proteica: ")
carga = -0.002

AACarga = {"C":-.045,"D":-.999,"E":-
.998,"H":.091,"K":1,"R":1,"Y":-.001}

for aa in protseq:
    if aa in AACarga:
        carga += AACarga[aa]
else:
     pass

print(carga)
```

Por último, hay que ejecutar sublime REPL para proceder a introducir la secuencia proteica. El problema con este programa es que reconoce los aminoácidos solo en mayúsculas. Si el usuario ingresa un aminoácido en minúsculas, se ignora. Una forma de solucionar este problema es ampliando el diccionario `AACarga` con las letras minúsculas como claves y, una mejor opción es convertir todos los aminoácidos que se introduzcan a mayúsculas usando `upper()`.

Además, si se desea que devuelva más de un valor, se puede indicar que devuelva una lista o una tupla. La función también podría modificarse para devolver, además de la carga neta, la proporción de aminoácidos cargados:

```
def carga_y_prop(AAseq):
     “””Retorna la carga neta de una proteína y la
proporción de aminoácidos con carga”””

     protseq = AAseq.upper()
     carga = -0.002
     cp_o = 0
     AACarga = {"C":-.045,"D":-.999,"E":-
.998,"H":.091,"K":1,"R":1,"Y":-.001}

     for aa in protseq:
          carga += AACarga.get(aa,0)
     if aa in AACarga:
          cp_o += 1
          prop = (100*cp_o/len(AAseq))

     return (carga, prop)


Ejecución del script:
>>> carga_y_prop("qtallvvlvllavalqateagpyga")

Output:
(-1.0009999999999999, 8.0)

>>> carga_y_prop("eeargplrgkgdqksavsqkprsrgilh")

Output:
(4.0940000000000003, 39.285714285714285)
```

Se debe tener especial cuidado al introducir el argumento, ya que, si se introduce la secuencia en mayúsculas la instrucción donde se indica convertir las letras ingresadas a mayúsculas no se ejecutará, por lo que nunca se definirá la variable `AAseq` y con eso no puede ejecutarse el bucle debido a que `prop` nunca será definida tampoco, a lo que se obtendría el siguiente error:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 16, in carga_y_prop
UnboundLocalError: local variable 'prop' referenced before
assignment
```

Se pueden añadir especificaciones a los resultados (como para obtener solamente la carga o la proporción) que se requiere obtener en consola añadiendo sub-scripts de indexación:

```
>>> carga_y_prop ("qtallvvlvllavalqateagpyga")[0]
>>> carga_y_prop ("eeargplrgkgdqksavsqkprsrgilh")[1]
```

Búsqueda de zona de menor degeneración en primers para PCR

Para encontrar primers de PCR, es mejor utilizar una región de ADN con menos degeneración (o más conservación). Esto se hace para tener una mejor oportunidad de encontrar la secuencia objetivo. El objetivo del siguiente programa es buscar esta región. Dado que un primer de PCR tiene aproximadamente 16 nucleótidos, para dar espacio para el diseño del primer, el espacio de búsqueda debe tener una longitud de al menos 45 nucleótidos. Se debería encontrar una región de 15 aminoácidos en la secuencia de entrada, ya que, 15 aminoácidos proporcionan una región de búsqueda de 45 nucleótidos (3 nucleótidos por aminoácido). Cada aminoácido está codificado por una determinada cantidad de codones. Por ejemplo, la valina (V) puede estar codificada por cuatro codones diferentes (GTT, GTA, GTC, GTG), mientras que el triptófano (W) está codificado solo por un codón (TGG). Lo más probable es que una región rica en valinas tendrá más variabilidad que una región con mucho triptófano.

```
protdeg =
{"A":4,"C":2,"D":2,"E":2,"F":2,"G":4,"H":2,"I":3,"K":2,"L":
6,"M":1,"N":2,"P":4,"Q":2,"R":6,"S":6,"T":4,"V":4,"W":1,"Y"
:2}
protsec = input("Secuencia proteica: ").upper()

segsvalores = []

for aa in range(len(protsec)):
     segmento = protsec[aa:aa+15]
     degen = 0

     if len(segmento)==15:
          for x in segmento:
               degen += protdeg.get(x,3.05)
          segsvalores.append(degen)
     else:
          pass

min_value = min(segsvalores)
minpos = segsvalores.index(min_value)

print (protsec[minpos:minpos+15])
```

Este programa toma una cadena (`protsec`) ingresada por el usuario. El programa usa un diccionario (`protdeg`) para almacenar la cantidad de codones que corresponde a cada aminoácido. De la línea 6 a la 8 se genera deslizamiento ventanas de longitud 15. Por cada 15 segmentos de aminoácidos, la cantidad de codones se evalúa, luego, se selecciona el segmento con menos degeneración (línea 15). Cabe destacar que en la línea 9 hay una verificación del tamaño del segmento, ya que, cuando la secuencia se desliza lejos de `protsec`, la subcadena tiene menos de 15 aminoácidos.

Version con **`while`**

```
ProtSec = input("Secuencia proteica: ").upper()

ProtDeg =
{"A":4,"C":2,"D":2,"E":2,"F":2,"G":4,"H":2,"I":3,"K":2,"L":
6,"M":1,"N":2,"P":4,"Q":2,"R":6,"S":6,"T":4,"V":4,"W":1,"Y"
:2}

SecsValores = []; SecsSecs = []; segmento = ProtSec[:15]; a
= 0

while len(segmento)==15:
     degen = 0

for x in segmento:
     degen += ProtDeg.get(x,3.05)
     SecsValores.append(degen)
     SecsSecs.append(segment)
     a += 1; segmento = ProtSec[a:a+15]

print (SecsSecs[SecsValores.index(min(SecsValores))])
```

Una versión más simplificada

Una versión que también es funcional es la siguiente:

```
ProtSec = input("Secuencia proteica: ").upper()
ProtDeg =
{"A":4,"C":2,"D":2,"E":2,"F":2,"G":4,"H":2,"I":3,"K":2,"L":
6,"M":1,"N":2,"P":4,"Q":2,"R":6,"S":6,"T":4,"V":4,"W":1,"Y"
:2}
degen_tmp = max(ProtDeg.valores())*15

for n in range(len(ProtSec)-15):
     degen = 0

for x in ProtSec[n:n+15]:
     degen += ProtDeg.get(x,3.05)
if degen <= degen_tmp:
     degen_tmp = degen
     sec = ProtSec[n:n+15]

print(sec)
```

Generadores

Los generadores son una forma sencilla y potente de iterador. Un generador es una función especial que produce secuencias completas de resultados en lugar de ofrecer un único valor. En apariencia es como una función típica, pero en lugar de devolver los valores con `return` lo hace con la declaración `yield`. Hay que precisar que el término generador define tanto a la propia función como al resultado que produce. Una característica importante de los generadores es que tanto las variables locales como el punto de inicio de la ejecución se guardan automáticamente entre las llamadas sucesivas que se hagan al generador, es decir, a diferencia de una función común, una nueva llamada a un generador no inicia la ejecución al principio de la función, sino que la reanuda inmediatamente después del punto donde se encuentre la última declaración `yield` (que es donde terminó la función en la última llamada).

Capítulo VIII. Clases

Las clases son una plantilla o modelo a partir de la cual pueden crearse objetos y donde se determinan los atributos (características) y métodos (acciones realizables) que tendrán.

Atributos: Son variables de diferentes tipos (Entero; Texto; Booleanos) que pueden tener valores por defecto o le podrán ser asignadas al momento de la Instancia de un objeto determinando sus características y estado.

Instancia: Palabra que refiere a la creación de un objeto partiendo de una clase. Durante la instancia son asignados los valores iniciales para los atributos. Las instancias se logran utilizando el método especial `__init__()` el cual establece un primer parámetro inicial: *`self`*.

`self:` se usa para acceder a un atributo dentro del objeto (clase) en sí. Si no se prefijó una variable con *`self`* en un método de clase, no se podría acceder a esa variable en otros métodos de la clase o fuera de la clase. Por lo tanto, puede omitirse si se desea que la variable sea local para ese método. Del mismo modo, si se tuviera un método y no se tuviera ninguna variable que quisiera compartir con otros métodos, se podría omitir el *`self`* de los argumentos del método.

Métodos: Son funciones; acciones realizables por el objeto que permiten cambiar sus atributos o el de otros objetos de la clase permitiendo un cambio de estado del objeto; a menudo requiriendo argumentos (valores) para determinados parámetros.

La sintaxis general de una nueva clase (empleando la sintaxis más adecuada o menos problemática) es:

```
class Nombre_clase():
"""Descripción de la clase opcional"""
     Atributos_de_clase_opcionales =
     def __init__(self,n_argumentos):
          self.argumento1 =        #Atributos de instancia
          self.argumento_n =

     def método(self):                    #Métodos
          self.método = #pueden estar relacionados con los
argumentos
```

Este ejemplo (completo, con la clase y la subclase visto más adelante en "Herencia") define la clase `SecBase`, que simplemente almacena una secuencia y proporciona un método de acceso para ello. También define `RNASec` como una subclase de `SecBase`. Debido a que `SecBase` define `get_sec` ese método puede ser invocado en instancias de `RNASec`. El método `traduccion` se puede llamar en una instancia de `RNASec`, pero no en una instancia de `SecBase` más general, ya que, no define ese método. Aunque una secuencia RNA no tiene trabajo de inicialización para hacer para sus propios fines, debe reenviar a `SecBase .__ init__` los argumentos que espera:

```
class SeqBase:
    def __init__(self, seqstring):
        self.seq = seqstring

    def get_seq(self):
        return self.seq

    def contenido_gc(self):
        """Regresa el contenido de G y C en BaseSeq"""
        seq = self.get_seq().upper()
        return (seq.count('G') + seq.count('C')) / len(seq)

dna_seq = input("Introduce tu secuencia de DNA: ")
base = SeqBase(dna_seq)
print(base.contenido_gc())
```

Otro ejemplo relacionado con una clase de medicamentos líquidos (jarabe, suspensión, elixir, etc.) que pueden tener características en común:

```
class Medicamentos_liquidos():

     def __init__(self, api, contenido):

          self.api = api
          self.contenido = contenido
          self.disolucion = False
          self.desintegra = False
          self.absorbe = False

     def liberar(self):
          self.liberacion = True

     def desintegrar(self):
          self.desintegra = False

     def absorber(self):
          self.absorbe = True

     def estado(self):
          print("Principio activio: ", self.api, "\nDosis:
          ", self.contenido, "\nLiberación: ",
          self.disolucion, "\nDesintegración: ",
          self.desintegra)


     def estado(self):
          print("Principio activio: ", self.api, "\nDosis:
", self.contenido, "\nLiberación: ", self.disolucion,
"\nDesintegración: ", self.desintegra)
```

Capítulo IX. Módulos

Cuando se va a escribir un programa, además de disponer de las sentencias y funciones que constituyen el lenguaje de programación para construir el programa se puede hacer uso del código anterior que nosotros mismos u otros usuarios hayan generado. Por ejemplo, si es necesario leer un fichero de secuencias fasta y anteriormente se escribió una función capaz de realizar el trabajo sería absurdo repetir todo el script. Una alternativa sencilla sería copiar la función desde el programa anterior. Esto, que puede funcionar para problemas sencillos, sería muy complicado para proyectos un poco más grandes.

`import`

Se puede realizar la importación de módulos que se han creado por nosotros, que están en nuestra biblioteca personal, las que han creado otras personas o los que vienen incluidos en Python. Para visualizar los módulos importables con ayuda del comando de ayuda:

```
help> modules

Please wait a moment while I gather a list of all available modules...

Ejercicios_Manual   binascii            mimetypes           symtable
Label               binhex              mmap                sys
OpenSSL             bisect              mmapfile            sysconfig
Paquete             break_num           mmsystem            tabnanny
__future__          builtins            modulefinder        tarfile
_abc                bz2                 msilib              telnetlib
_ast                cProfile            msvcrt              tempfile
_asyncio            calendar            multiprocessing     test
_bisect             carga_prot          netbios             test_data
_blake2             carga_prot_new      netrc               test_pycosat
_bootlocale         cat                 nntplib             textwrap
_bz2                certifi             nt                  this
_cffi_backend       cffi                ntpath              threading
_codecs             cgi                 ntsecuritycon       time
_codecs_cn          cgitb               nturl2path          timeit
_codecs_hk          chardet             numbers             timer
_codecs_iso2022     chunk               odbc                tkinter
_codecs_jp          cmath               opcode              token
_codecs_kr          cmd                 operator            tokenize
_codecs_tw          code                optparse            tqdm
_collections        codecs              os                  trace
_collections_abc    codeop              parser              traceback
_compat_pickle      collections         pathlib             tracemalloc
_compression        colorsys            pdb                 traduccion_seq
_contextvars        commctrl            perfmon             tty
_csv                compileall          permanente          turtle
_ctypes             concurrent          pickle              turtledemo
_ctypes_test        conda               pickletools         types
_datetime           conda_env           pip                 typing
```

ILUSTRACIÓN 45. RESULTADOS DE AYUDA INTERACTIVA EN BÚSQUEDA DE TODOS LOS MÓDULOS INCLUIDOS EN PYTHON

Los programadores procuran no repetir un mismo trabajo varias veces, por lo que los lenguajes de programación incluyen formas de utilizar código anterior. En Python utilizar una

función de un programa anterior es muy sencillo, simplemente ha de ser importada al nuevo programa. Si se supone que tiempo atrás se escribió un programa llamado `analisis_secuencia` que incluía una función llamada `leer_secuencias_fasta`, para poder utilizar esta función en un nuevo programa lo que procede hacer es importarla:

La sintaxis general para importar módulos completos es:

```
>>> from modulo_a_importar import *
>>> función a usar(aplicación de la función)
```

```
from modulo_a_importar import *
función a usar(aplicación de la función)
```

Para importar una función en específico se puede realizar de la siguiente manera:

```
>>> from modulo_a_importar import función
>>> función(aplicación de la función)
```

Cuando en un proyecto se crean numerosos módulos estos se pueden organizar en directorios dando lugar a paquetes de módulos (packages).

Paquetes distribuibles

Utilizar paquetes ofrece varias ventajas. En primer lugar, permite unificar distintos módulos bajo un mismo nombre de paquete, pudiendo crear jerarquías de módulos y submódulos, o también subpaquetes.

Por otra parte, permiten distribuir y manejar fácilmente el código como si fueran librerías instalables de Python. De esta forma se pueden utilizar como módulos estándar desde el intérprete o scripts sin cargarlos previamente.

Para crear un paquete lo que se debe hacer es crear un fichero especial *init* vacío en el directorio donde se encuentren todos los módulos que se quieren agrupar en nuestro

ordenador. De esta forma cuando Python recorra este directorio, será capaz de interpretar una jerarquía de módulos. La manera de crear el módulo será desde Sublime:

1. Abrir un nuevo archivo (ctrl +n) .
2. Guardar el nuevo archivo dentro de la carpeta donde se encuentran nuestros módulos. El nombre deberá ser `__init__` con la extensión *.py y no deberá poseer nada en su interior.
3. Al finalizar, se deberá observar como a continuación:

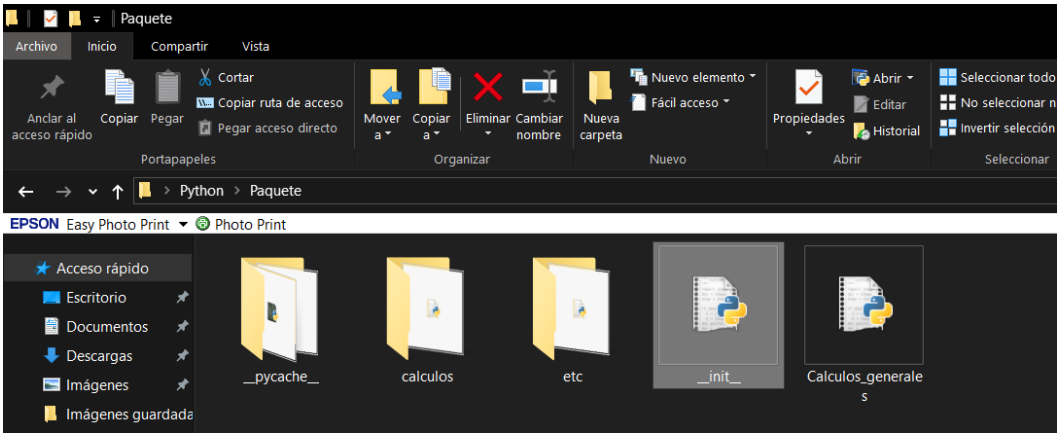


ILUSTRACIÓN 46. PRESENCIA DE ARCHIVO __INIT__ EN LA CARPETA DE PAQUETES

4. Se debe de crear el archivo de “Setup” en la carpeta principal. Será desde SublimeText y deberá incluir lo siguiente con los datos e información que se desee trabajar. En este caso será el programa de cálculos aritméticos visto anteriormente:

```
from setuptools import setup

setup(
      name = "Paquetecalculos",
      #Nombre del paquete#
      version = "1.0",
      #Versión actual (esta dependerá de las actualizaciones que
      se realicen)#
      description = "Paquete de redondeo y potencia",
      #Descripción rápida y concreta#
      autor = "FESC-C1",
      #Nombre del autor, institución, etc.#
      autor_email = "majofarma@comunidad.unam.mx",
      #Útil para contactar sobre dudas del paquete#
      url = "https://www.miurl.com",
      #Aqui va un enlace para consultar info sobre el paquete o
      sobre el autor del paquete, etc.#
      packages = ["Paquete",
"Paquete.calculos.Redondeo_potencia"]
      #Esta es la ruta de acceso en el ordenador. Primero va la
      carpeta   origen y después la ruta completa. #
      )
```

5. Para poder crear el paquete se debe de entrar al símbolo del sistema de Windows. Puede buscarse como “cmd” en la barra de búsqueda.

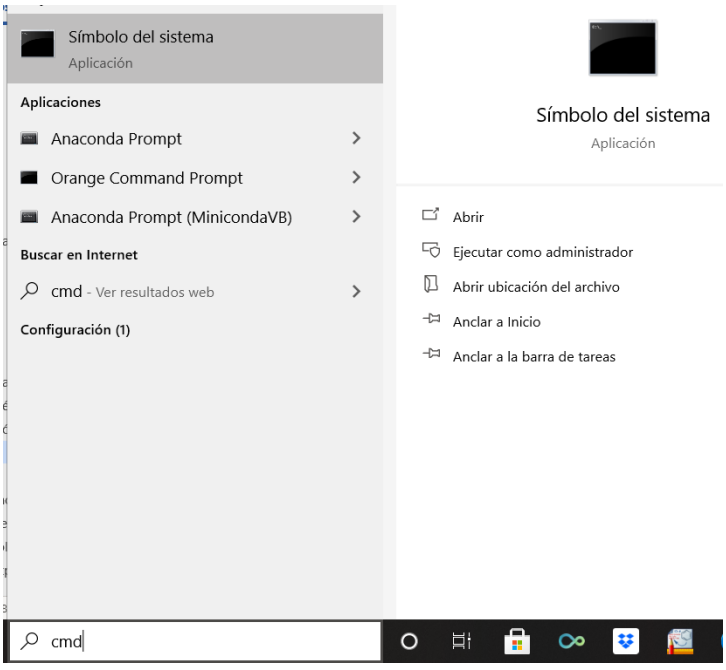


ILUSTRACIÓN 47. BÚSQUEDA DEL SÍMBOLO DEL SISTEMA EN MENÚ DE WINDOWS 10

6. Se introducirá una ruta similar a la siguiente:

```
C:\Users\Nombre_Usuario>cd Desktop

C:\Users\Nombre_Usuario\Desktop>cd Python

C:\Users\Nombre_Usuario\Desktop\Python>  python  setup.py
sdist

#En nombre de Usuario se podrá visualizar el nombre asignado
a nuestra PC. En este caso la carpeta principal destinada a
Python está en el escritorio, se deberá prestar especial
atención para que la ruta sea correcta y se ejecute el
proceso de manera adecuada.
```

Realizará varios procesos, verificar que no hay errores y observar la creación de 2 carpetas: *.egg y un `dist` (en éste está el archivo con extensión*.tar.gz donde está el paquete para instalar). Se observará algo similar a lo siguiente:

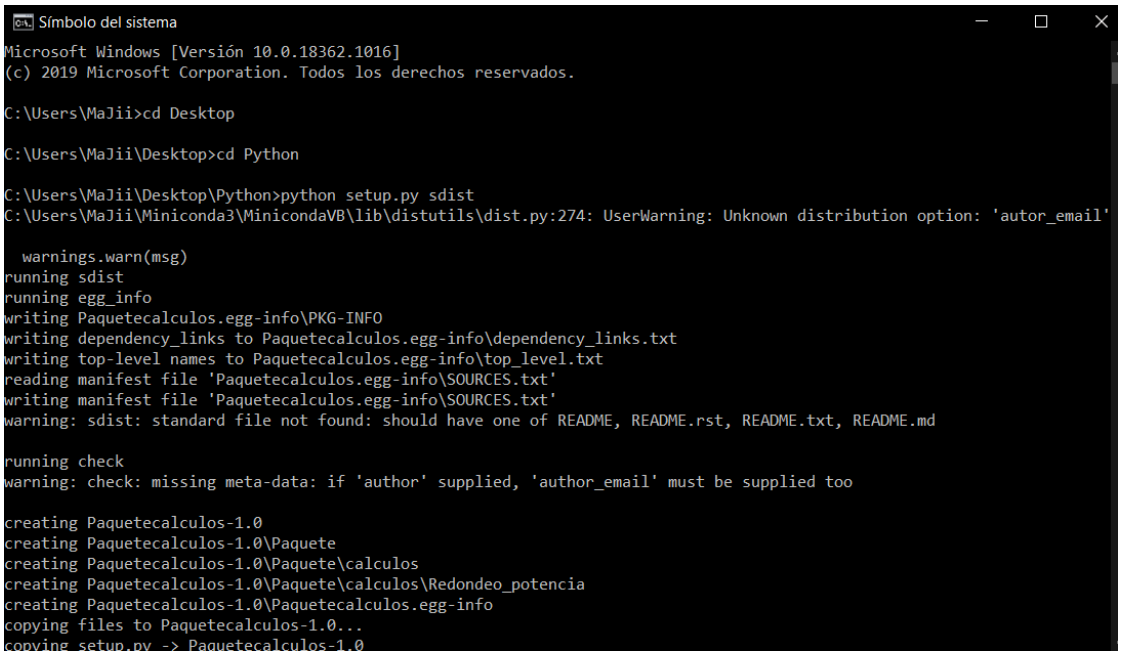


ILUSTRACIÓN 48. VISUALIZACIÓN DE INSTRUCCIONES Y PROCESOS EN EL SÍMBOLO DEL SISTEMA DE WINDOWS

7. Entrar desde cmd a la carpeta `dist` que está dentro de la carpeta donde se encuentra el paquete (deberá de ser un archivo con extensión *.tar.gz creado a partir de la ejecución del setup) e introducir "`pip3 install nombre del paquete.tar.gz`".

   Nota: Para actualizar el `pip` en caso de ser necesario se debe de introducir lo siguiente:

   ```
   C:\Users\Nombre_de_Usuario>python -m pip install --upgrade
   ```

8. Se deberá de probar usando el módulo o la función que es requerida desde cualquier lugar y esta deberá de ser ejecutada correctamente. Para desinstalar dentro de `dist` teclear: `pip3 unistall` y el paquete a desinstalar.

Biopython

Biopython es un ejemplo de paquete que resulta de sumo interés para las ciencias químico-biológicas. Biopython es una librería que recoge utilidades para trabajar con datos biológicos. Algunas de dichas utilidades incluyen funciones y clases para trabajar con secuencias, resultados Blast, estructuras de proteínas y árboles filogenético, entre muchas otras cosas.

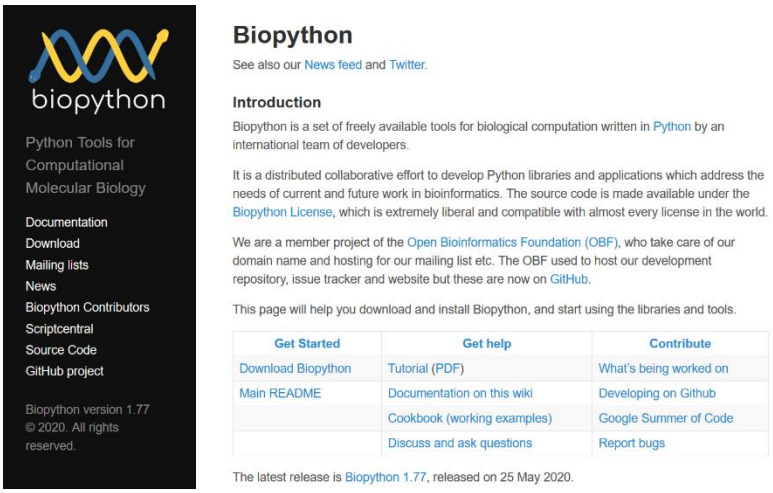


ILUSTRACIÓN 49. PÁGINA DE INICIO DE BIOPYTHON

Para poder trabajar con Biopython es necesario importar el paquete. Debe ser descargado, acceder a la carpeta `dist` y ejecutar el mismo comando para instalarlo. Sin embargo, es mucho más amigable y sencillo el trabajar con Bipython en sistemas operativos como Linux o MacOS. Se puede emplear una máquina virtual para trabajar en ambiente Linux, pero esto requiere de mucho espacio en el disco duro del ordenador y de una computadora con potencia suficiente.

Algunos ejemplos de lo que es posible realizar con módulos de Biopython se muestran a continuación de manera ilustrativa. Estos ejemplos no pueden ser llevados a cabo sin realizar la instalación del paquete. Algunos ejemplos pueden resultar similares o incluso casi iguales a ejemplos anteriores en donde se tuvieron que crear funciones, pulir el código, hacer correcciones, probar su funcionalidad, etc., y esto es debido a que Biopython está especialmente destinado para trabajar con datos biológicos y puede ahorrar mucho trabajo en la creación de funciones para trabajar con secuencias o datos similares y, de esta manera, eficientizar el trabajo para obtener resultados útiles de manera rápida. Es una herramienta sumamente útil. Se debe considerar que existen libros enteros para aprender a cómo emplear este paquete y todas las funcionalidades y aplicaciones de Biopython.

Ejemplos

1. Se creará un objeto `Seq` con la secuencia `'ATCG'` y se verificará: ¿Cuántos residuos tiene? ¿Cuáles son las tres primeras letras? ¿Y la última? ¿Cuántas adeninas tiene? Convertir la secuencia en una cadena de texto normal sin alfabeto.

```
>>> from Bio.Seq import Seq
>>> secuencia = Seq('ATCG')
>>> secuencia

Output: Seq('ATCG', Alphabet())

>>> secuencia

Output: ATCG

>>> len(secuencia)

Output: 4

>>> secuencia[:3]

Output: Seq('ATC', Alphabet())

>>> secuencia[-1]

Output: 'G'

>>> secuencia.count('A')

Output: 1

>>> str(secuencia)

Output: 'ATCG'
```

2. Crear la secuencia reversa y complementaria de 'ATCG'.

```
>>> secuencia = Seq('ATCG')
>>> secuencia.reverse_complement()

Output: Seq('CGAT', Alphabet())
```

3. Traducir la secuencia 'ATGGCCATTGT' a proteína.

```
>>> secuencia = Seq('ATGGCCATTGT')
>>> secuencia.translate()

Output: Seq('MAI', ExtendedIUPACProtein())
```

4. Crear dos secuencias ‘ATGGCCATTGT’ y comprobar si son iguales.

```
>>> secuencia = Seq('ATGGCCATTGT')
>>> secuencia2 = Seq('ATGGCCATTGT')
>>> secuencia == secuencia2
#Aqui se da una explicación por parte del paquete de por
qué no deben de compararse así las secuencias

Output: False

>>> str(secuencia) == str(secuencia2)

Output: True

>>> id(secuencia) == id(secuencia2)

Output: False

>>> id(secuencia)

Output: 140353539658896

>>> id(secuencia2)

Output: 140353539660880
```

5. Crear un `SeqRecord` con la secuencia `‘ATCG’`, la id `‘secuencia’` y la descripción ‘prueba’. Un `SeqRecord` utilizado en Biopython es un objeto `Seq` empleado para contener una secuencia con identificadores (ID y nombre), descripción y, opcionalmente, anotación y subcaracterísticas:

```
>>> from Bio.Seq import Seq
>>> from Bio.Seq import SeqRecord
>>> SeqRecord(Seq('ATCG'), id = 'secuencia', description =
'prueba')
SeqRecord(seq = Seq('ATCG', Alphabet()), id ='secuencia',
name = '<unknown name>', description = 'prueba', dbxrefs =
[])
```

6. A continuación, un ejemplo más ordenado y trabajando con una secuencia proteica

```
from Bio.Seq import Seq
from Bio.SeqRecord import SeqRecord

record = SeqRecord(
    Seq("MKQHKAMIVALIVICITAVVAALVTRKDLCEVHIRTGQTEVAVF"),
    id="YP_025292.1",
    name="HokC",
   description=" proteína de membrana tóxica, pequeña",
)

print(record)
```

7. La manera de extraer información de un `SeqRecord` es:

```
from Bio import SeqIO

for index, record in enumerate(SeqIO.parse("ls_orchid.gbk",
"genbank")):
    print(
        "index %i, ID = %s, length %i, with %i features"
        % (index, record.id, len(record.seq),
len(record.features))
    )

Output:
index 0, ID = Z78533.1, length 740, with 5 features
index 1, ID = Z78532.1, length 753, with 5 features
index 2, ID = Z78531.1, length 748, with 5 features
...
```

Si se ejecuta el siguiente comando brindará información interesante sobre el objeto:

```
>>> print(record)

Output:
ID: Z78439.1
Name: Z78439
Description: P.barbatum 5.8S rRNA gene and ITS1 and ITS2
DNA
Number of features: 5
/molecule_type=DNA
/topology=linear
/data_file_division=PLN
/date=30-NOV-2006
/accessions=['Z78439']
/sequence_version=1
/gi=2765564
/keywords=['5.8S ribosomal RNA', '5.8S rRNA gene',
'internal transcribed spacer', 'ITS1', 'ITS2']
/source=Paphiopedilum barbatum
/organism=Paphiopedilum barbatum
/taxonomy=['Eukaryota', 'Viridiplantae', 'Streptophyta',
..., 'Paphiopedilum']
/references=[Reference(title='Phylogenetics of the slipper
orchids (Cypripedioideae: Orchidaceae): nuclear rDNA ITS
sequences', ...), Reference(title='Direct Submission',
...)]
Seq('CATTGTTGAGATCACATAATAATTGATCGAGTTAATCTGGAGGATCTGTTTACT
...GCC')
```

8. Leer un fichero de secuencias FASTA y hacerlas todas minúsculas.

```
from Bio import SeqIO
def hacer_fasta_minusculas(fichero):
    'Lee las secuencias de un fichero y las hace
minusculas'

    for seq_record in SeqIO.parse(fichero, 'fasta'):
        print '>' + seq_record.id
        print str(seq_record.seq).lower()

if __name__ == '__main__':
    hacer_fasta_minusculas('secuencias.fasta')
```

9. Pasar un fichero de genbank a FASTA.

```
from Bio import SeqIO

def convertir_secuencia(fichero_entrada, formato_entrada,
                        fichero_salida,  formato_salida):
    'Convierte un fichero de secuencia de un formato a
otro'

    secuencias = SeqIO.parse(fichero_entrada,
formato_entrada)
    SeqIO.write(secuencias, fichero_salida, formato_salida)

    #Alternativamente se podria utilizar la funcion convert
    #esto es mas eficiente
    #SeqIO.convert(fichero_entrada, formato_entrada,
    #              fichero_salida, formato_salida)


if __name__ == '__main__':
    formato_entrada = 'genbank'
    fichero_entrada = 'sequence.gb'
    formato_salida  = 'fasta'
    fichero_salida  = 'sequence.fasta'
    convertir_secuencia(fichero_entrada, formato_entrada,
                        fichero_salida,  formato_salida)
```

10. Filtrar las secuencias de un fichero sanger fastq por calidad media.

```
from Bio import SeqIO

def filtrar_secuencias(fichero_entrada, fichero_salida,
                       umbral_calidad):
    'Dado un fichero fastq filtra las secuencias por
calidad'

    formato = 'fastq'

    #Esto no funcionara para millones de secuencias
    #porque lo carga todo en memoria, habria que usar
    #generadores
    secuencias_filtradas = []
    for seq_record in SeqIO.parse(fichero_entrada,
                                  formato):
        calidades =
seq_record.letter_annotations['phred_quality']

        calidad_media = float(sum(calidades)) /
len(calidades)

        print calidad_media

        if calidad_media < umbral_calidad:
            continue

        secuencias_filtradas.append(seq_record)

    SeqIO.write(secuencias_filtradas, fichero_salida,
formato)

if __name__ == '__main__':
    filtrar_secuencias('seqs_illumina.fastq',
                       'seqs_filtradas.fastq',
                       35)
```

Si se quieren poner en práctica estos ejercicios se puede instalar Biopython en el ordenador con sistema operativo Windows. Es necesaria una herramienta adicional: Visual Studio. Puede descargarse desde el siguiente enlace:

https://visualstudio.microsoft.com/es/thank-you-downloading-visual studio/?sku=BuildTools&rel=16#

y se puede ahondar más sobre el paquete de Biopython, así como poner en práctica estos ejemplos o crear los propios.

Microsoft Visual Studio es un entorno de desarrollo integrado (IDE, por sus siglas en inglés) para Windows y macOS. Es compatible con múltiples lenguajes de programación, tales como C++, C#, Visual Basic .NET, F#, Java, Python, Ruby y PHP, al igual que entornos de desarrollo web, como ASP.NET MVC, Django, etc.

Capítulo X. Herencia

En programación orientada a objetos, la herencia es la capacidad de reutilizar una clase extendiendo su funcionalidad. Una clase que hereda de otra puede añadir nuevos atributos, ocultarlos, añadir nuevos métodos o redefinirlos. Las clases pueden tener una jerarquía similar a la de la vida real:

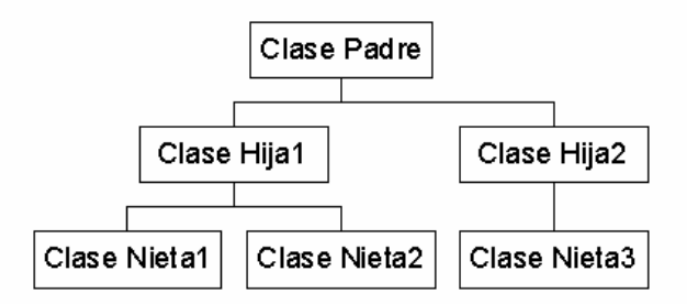


ILUSTRACIÓN 50. JERARQUÍA DE LAS CLASES

Para poder heredar de una clase a otra hay que estructurar la nueva clase de la siguiente manera:

```
class clase_nueva(clase_que_hereda):
#Después se sigue la misma estructura para la creación de
una clase#
```

La herencia va a sobrescribir los métodos de la clase padre o superior a la subclase o clase hija y no es necesario reescribir todo el código. Si es añadido un método nuevo sólo será escrito éste. Se puede visualizar en el ejemplo de los medicamentos líquidos:

```
class Suspension(Medicamentos_liquidos):
     resuspender = ""
     def tipo(self):
          self.tipo = "Suspensión estructurada"

     def estado(self):
          print("Principio activio: ", self.api,
          "\nContenido: ", self.contenido, "\nLiberación:
          ", self.disolucion, "\nDesintegración: ",
          self.desintegra, "\n", self.tipo) #Aqui se están
          sobreescribiendo métodos

miSuspension = Suspension("Naproxeno", "3.2g/100mL")
```

```
miSuspension.tipo() #si solo se incluye esto si funciona,
pero no se informa. Para que se visualice hay que
sobreescribir el método estado#

miSuspension.estado()

#ejecutar REPL para observar el output#
```

Sin embargo, es mejor emplear la función `super()` la cual es útil para herencia simple (herencia de una sola clase padre a hija) o múltiple (herencia de varias clases superiores a subclases).

`super()`: Esta función nos permite invocar y conservar un método o atributo de una clase padre (primaria) desde una clase hija (secundaria) sin tener que nombrarla explícitamente. Esto brinda la ventaja de poder cambiar el nombre de la clase padre (base) o hija (secundaria) cuando se desee y aun así mantener un código funcional y sencillo.

Retomando el ejemplo anterior de la creación de la clase `SecBase`, para realizar la herencia a la nueva subclase `RNASec` se observaría como a continuación. En esta herencia si se lleva a cabo el uso de la función `super()`:

```
class RNASec(SecBase):
     CodonTabla = {'UUU': 'F', 'UCU': 'S', 'UAU': 'Y',
'UGU': 'C',

     }

     def traduccion_codon(self, codon):
          return self.CodonTabla[codon.upper()]

     def __init__(self, seqstring):
          super().__init__(seqstring) # aquí ose observa
que super() no emplea self #

     def translate(self, frame=1):
          #Produce una secuencia proteica al traducir la
secuencia de RNA comenzando del frame 1, 2 o 3#


     return''.join([self.traduccion_codon(self.get_sec()[n:
n+3])
                for n in
                range(frame-1,
                     # ignora 1 o 2 bases después de 3
                     len(self.get_sec()) -
                     (len(self.get_sec()) - (frame-1)) % 3,
                     3)])
```

Capítulo XI. Polimorfismo

La técnica de polimorfismo en Python y lenguaje orientado a objetos significa la capacidad de tomar más de una forma. Una operación puede presentar diferentes comportamientos en diferentes instancias. El comportamiento depende de los tipos de datos utilizados en la operación. El polimorfismo es ampliamente utilizado en la aplicación de la herencia. El polimorfismo es la capacidad de enviar el mismo mensaje (solicitud para ejecutar un método) a diferentes objetos, cada uno de los cuales parece realizar la misma función. Sin embargo, la forma en que se maneja el mensaje depende de la clase del objeto.

```
#Para poder poner en práctica el polimorfismo se puede
prescindir de cierta parte del código y hacer un script
sencillo#

class Tableta():

     def contenidoAPI(self):
          print("Contiene principio activo")

class Jarabe():

     def contenidoAPI(self):
          print("Contiene principio activo")


class Suspensión():

     def contenidoAPI(self):
          print("Contiene principio activo")


def contenidoAPI(API): #Aqui se almacena miMedicamento y el
polimorfismo hace que se transforme en tipo que es llamando
(Jarabe) y que llame a la clase Jarabe con su método#
     API.contenidoAPI() #No se sabe cual está llamando#


miMedicamento = Jarabe() #Aqui es posible cambiar los tipos
de objetos(Jarabe, suspensión, tableta, etc.)#

contenidoAPI(miMedicamento) #Objeto de tipo Jarabe
```

En este ejercicio se trabaja con medicamentos que tienen una característica en común: todos poseen un principio activo. Sin embargo, considerando todas las formas farmacéuticas que se poseen tanto en la vida real como en el ejemplo, todas son muy distintas entre sí. Sin embargo, la función que contiene cada uno de ellos es la misma, porque se adapta al objeto al que se quiera, demostrando la capacidad de adaptación del polimorfismo en el lenguaje.

Capítulo XII. Encapsulamiento

El encapsulamiento es una manera de proteger o volver privado el código que se está creando o ha sido creado. Aunque no existe un método preciso o exclusivo dentro de Python, se puede lograr de manera muy sencilla colocando dos guiones bajos delante de lo que se desea encapsular (métodos, atributos, etc.). Las encapsulaciones se realizan cuando se necesita y se realizan bajo el criterio propio del programador y por el comportamiento del programa. La manera de encapsular es como en el siguiente ejemplo:

```
def __método(self): #encapsulación de método
     self.__atributo = #encapsulamiento de propiedad o
atributo
```

Como el código encapsulado no es accesible, no se puede llamar a la consola desde fuera (o sea colocando __ en los llamados o en el intérprete).

Capítulo XIII. Manipulación de archivos

El manejo o manipulación de archivos es útil para escribir o leer cadenas de caracteres para/desde otros archivos (otros tipos deben ser convertidas a cadenas de caracteres). Para esto Python incorpora un tipo integrado llamado `file`, el cual, es manipulado mediante un objeto archivo, el cual fue generado a través de una función integrada en Python. A continuación, se describen los procesos típicos y sus referencias a funciones propias del lenguaje. El objetivo de manipulación de archivos externos es la persistencia de los datos, poder guardar nuestro trabajo, poder trabajar con bases de datos, inclusive trabajar el mayor tiempo posible desde la consola sin necesidad de salir a crear carpetas nuevas o explorar el contenido de otras más.

Las fases necesarias básicas en la manipulación de archivos son:

1. Creación o existencia de un archivo
2. Apertura del archivo
3. Manipulación
4. Cierre
5. Decisión final (eliminación, etc.)

Abajo se presenta una tabla con las funciones más esenciales al momento de realizar manipulación de archivos en Python.

TABLA 6. FUNCIONES ÚTILES ESENCIALES EN LA MANIPULACIÓN DE ARCHIVOS EN PYTHON

Modo	Función
`r`	el archivo se abre en modo de solo lectura, no se puede escribir (argumento por defecto).
`w`	modo de solo escritura (si existe un archivo con el mismo nombre, se borra).
`a`	modo de agregado (append), los datos escritos se agregan al final del archivo.
`r+`	el archivo se abre para lectura y escritura al mismo tiempo.
`b`	el archivo se abre en modo binario, para almacenar cualquier cosa que no sea texto.
`U`	el archivo se abre con soporte a nueva línea universal, cualquier fin de línea ingresada será como un \n en Python.

Abrir un archivo:

La forma preferida para abrir un archivo es usando la función integrada `open()`.

Leer un archivo:

Para leer un archivo es usando algunas de los métodos del tipo objeto file como `read()`, `readline()` y `readlines()`.

Escribir:

Para escribir un archivo es usando el método del tipo objeto file llamado `write()`.

Cerrar:

Para cerrar un archivo es usando el método del tipo objeto file llamado `close()`.

Módulo os

El módulo os de Python permite realizar operaciones dependientes del Sistema Operativo como crear una carpeta, listar contenidos de una carpeta, conocer acerca de un proceso, finalizar un proceso, etc. Este módulo tiene métodos para ver variables de entornos del Sistema Operativo con las cuales Python está trabajando en mucho más. A continuación, de manera general el módulo os:

Crear una nueva carpeta

```
>>> import os
>>> os.makedirs("Carpeta nueva")
```

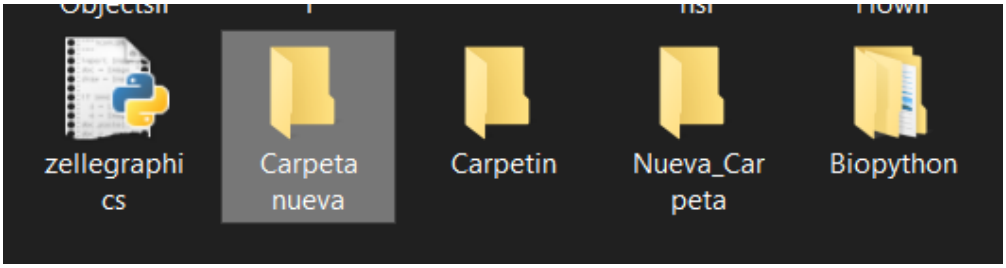


ILUSTRACIÓN 51. VISUALIZACIÓN DE CREACIÓN DE UNA NUEVA CARPETA DESDE SUBLIME TEXT

Listar el contenido de una carpeta

```
>>> import os
>>> os.listdir("./")

Output:
['abdc.py', 'Algoritmos con Python.txt', 'archivo_manual.txt',
'archivo_nuevo.txt', 'Bioinformatics for beginners genes,
genomes, molecular evolution, databases and analytical tools by
Supratim Choudhuri Michael Kotewicz.pdf', 'Bioinformatics...]
```

Mostrar el actual directorio de trabajo

```
>>> import os
>>> os.getcwd()

Output:
'C:\\Users\\Usuario\\Desktop\\Python'
```

Mostrar el tamaño del archivo en bytes del archivo pasado en parámetro

```
>>> import os
>>> os.path.getsize("DNA.txt")

Output: 1189
```

Comprobar los parámetros del archivo

```
>>> import os
>>> os.path.isfile("DNA.txt") #verifica si es un archivo

Output: True

>>> os.path.isdir("DNA.txt") #Verifica si es un directorio o
carpeta

Output: False
```

Cambiar directorio/carpeta

```
>>> import os
>>> os.chdir("Carpeta nueva")#cambia de carpeta a la indicada
>>> os.getcwd()

Output:
'C:\\Users\\Usuario\\Desktop\\Python\\Carpeta nueva'

>>> os.listdir("./")

Output: []

>>> os.chdir("../")
>>> os.getcwd()

Output:
'C:\\Users\\Usuario\\Desktop\\Python'
```

Renombrar un archivo

```
>>> import os
>>> os.rename("Carpeta nueva","Carpeta_Módulo_os")
```

Eliminar una carpeta

```
>>> os.rmdir("Carpeta_Módulo_os")
>>> os.chdir("Carpeta_Módulo_os")

Output:
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] El sistema no puede encontrar el
archivo especificado: 'Carpeta_Módulo_os'
```

Módulo `io`

Este módulo provee las facilidades principales de Python para manejar diferentes tipos de E/S(entrada y salida). Hay tres diferentes tipos de E/S: texto E/S, binario E/S y E/S sin formato.

Estas son categorías generales y varios respaldos de almacenamiento se pueden usar para cada una de ellas. Un objeto concreto perteneciendo a cualquiera de estas categorías se llama un objeto tipo file o archivo. Independiente de su categoría, cada objeto stream también tendrá varias capacidades: puede ser solamente para lectura, solo escritura, o lectura y escritura. También permite arbitrariamente acceso aleatorio (buscando adelante o hacia atrás en cualquier lugar) o solamente acceso secuencial.

Para abrir un archivo y leer lo que contiene, la sintaxis general sería así:

```
f = open("archivo.txt", 'r')
lines = f.readlines()
for line in lines:
       print(line)
```

Para crear un nuevo archivo:

```
from io import open

#Puede abrir el archivo en modo lectura, escritura o append.
A continuación, se abrirá en modo escritura (write)#

archivo_texto = open("archivo_nuevo.txt","w") #Aqui se puede
nombrar el nuevo archivo#

frase = "Este es \n un ejemplo rápido \n para comprender la
creación de archivos" #Información escrita#

archivo_texto.write(frase) #escritura dentro del archivo#

archivo_texto.close()
#Se crea un archivo de texto y contiene lo ahí indicado
(recuerda manejar todo dentro de una misma carpeta, será más
fácil). Ejecutar REPL y verificar la creación del archivo#
```

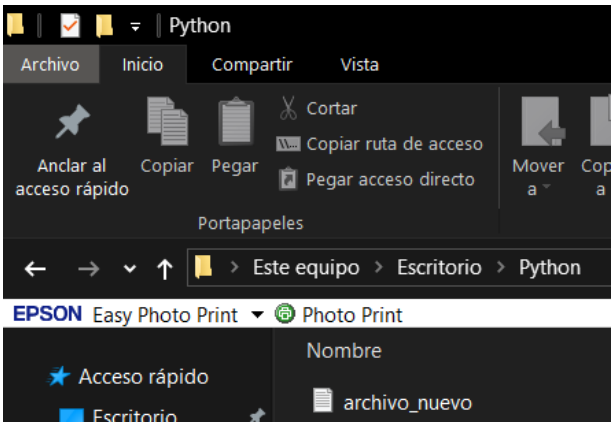


ILUSTRACIÓN 52. CREACIÓN DE ARCHIVO NUEVO (VERIFICACIÓN)

Y verificar que el contenido en su interior sea el que ha sido especificado por el usuario programador (nosotros):

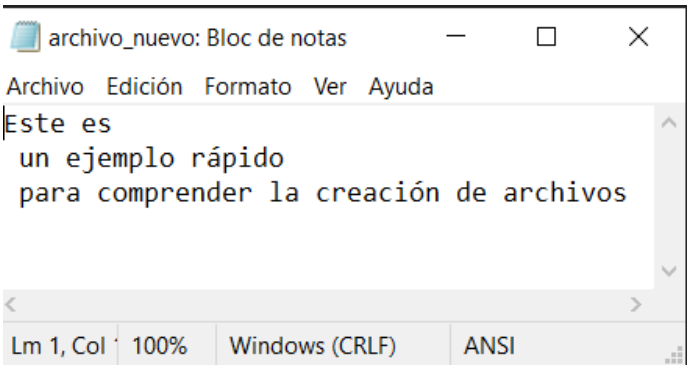


ILUSTRACIÓN 53. VERIFICACIÓN DE CONTENIDO DEL ARCHIVO

Para abrir el archivo con Python en modo lectura, introducir lo siguiente:

```
#Para modo lectura:
from io import open

archivo_texto = open("archivo_nuevo.txt","r")

texto = archivo_texto.read()

archivo_texto.close()

print(texto)

#Ejecutar REPL y obser el texto contenido en el archivo
```

La función de `readlines()` sirve para guardarlo dentro de una lista de líneas de texto manipulables:

```
from io import open

archivo_texto = open("archivo_nuevo.txt","r")

lineas_texto = archivo_texto.readlines()

archivo_texto.close()

print(lineas_texto)
```

Para acceder a una línea en específico (a) o añadir más líneas de texto (b) sería de la siguiente manera:

```
# a)
print(lineas_texto[0])

#Aqui aparecerá la línea específica

#b)
from io import open

archivo_texto = open("archivo_nuevo.txt","a")

archivo_texto.write("\n espero que estés aprendiendo mucho")

archivo_texto.close()
```

Siempre que se añadan líneas de texto en modo "`r+`" o lectura y escritura, las líneas añadidas de texto se observarán al principio del archivo (habrá una sobreescritura):

```
from io import open

archivo_texto = open("archivo_nuevo.txt","r+")

archivo_texto.write("Sobreescribir texto")
```

Para manipular el puntero dentro de ficheros de texto usar: `seek()`, esta función le dirá al puntero donde situarse dentro de las líneas de texto del archivo que esté siendo trabajado.

```
from io import open
archivo_texto = open("archivo_nuevo.txt","r")
print(archivo_texto.read())
archivo_texto.seek(0)
#despues de leerlo se recorre hasta el principio de nuevo#
```

Para comenzar a trabajar desde otro punto diferente del inicio del archivo de texto se puede observar a continuación:

```
from io import open

archivo_texto = open("archivo_nuevo.txt","r")

print(archivo_texto.read())

archivo_texto.seek(11) #se situara en el caracter no.11 y
comenzará la siguiente instrucción desde ese lugar

print(archivo_texto.read())

archivo_texto.close()

El output después de ejecutar REPL será:

Este es
 un ejemplo rápido
 para comprender la creación de archivos
 espero que estés aprendiendo mucho
 un ejemplo rápido
 para comprender la creación de archivos
 espero que estés aprendiendo mucho
```

Una aplicación curiosa es la de aplicar operaciones matemáticas a los archivos de texto. Por ejemplo: tener varias copias de archivos que son, inicialmente iguales, pero conforme avanzan o incluso, en los segmentos finales, son diferentes (sólo por ejemplificar algo

burdamente). Para observar la parte final del texto se puede aplicar el siguiente código y ejecutar en REPL:

```
from io import open

archivo_texto = open("archivo_nuevo.txt","r")

archivo_texto.seek(len(archivo_texto.read())/2) #nos devuelve la
mitad del texto que posee el archivo

print(archivo_texto.read())
```

Para modificar líneas existentes dentro del texto (no se puede emplear este script con más líneas de las que posee el archivo). Lo que hará será cambiar la línea de texto existente por la que se indique:

```
from io import open

archivo_texto = open("archivo_nuevo.txt","r+")

lista_texto = archivo_texto.readlines();

lista_texto[1] = " Esta linea ha sido incluida desde el exterior
\n"

archivo_texto.seek(0)

archivo_texto.writelines(lista_texto)

archivo_texto.close()
```

Pero la manipulación de archivos no sólo se trata de archivos en formato de texto *.txt, también se pueden manipular archivos en formato FASTA. Es un formato de fichero informático basado en texto, utilizado para representar secuencias de ácidos nucleicos, de péptidos, y en el que los pares de bases o los aminoácidos se representan usando códigos de una única letra. El formato también permite incluir nombres de secuencias y comentarios que preceden a las secuencias en sí. La simplicidad del formato FASTA hace fácil el manipular

y analizar secuencias usando herramientas de procesamiento de textos y lenguajes como Python.

Los archivos con formato FASTA suelen tener estructura especifica. Según la NCBI: “Una secuencia en formato FASTA comienza con una descripción de una sola línea, seguida de líneas de datos de secuencia. La línea de descripción (defline) se distingue de los datos de secuencia por un símbolo mayor que (">") al principio. Se recomienda que todas las líneas de texto tengan menos de 80 caracteres y no se permiten los espacios en blanco”.

Una secuencia de ejemplo en formato FASTA es:

```
>NC_045512.2 Severe acute respiratory syndrome coronavirus 2
isolate Wuhan-Hu-1, complete genome
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTG
TTCTCTAAA
CGAACTTTAAAATCTGTGTGGCTGTCACTCGGCTGCATGCTTAGTGCACTCACGCAGTATA
ATTAATAAC
TAATTACTGTCGTTGACAGGACACGAGTAACTCGTCTATCTTCTGCAGGCTGCTTACGGTT
TCGTCCGTG
TTGCAGCCGATCATCAGCACATCTAGGTTTCGTCCGGGTGTGACCGAAAGGTAAGATGGAG
AGCCTTGTC
CCTGGTTTCAACGAGAAAACACACGTCCAACTCAGTTTGCCTGTTTTACAGGTTCGCGACG
TGCTCGTAC
GTGGCTTTGGAGACTCCGTGGAGGAGGTCTTATCAGAGGCACGTCAACATCTTAAAGATGG
CACTTGTGG
CCAAGGGTTGAAAAGAAAAAGCTTGATGGCTTTATGGGTAGAATTCGATCTGTCTATCCAG
TTGCGTCAC
```

Donde se espera que las secuencias estén representadas en los códigos de aminoácidos y ácidos nucleicos estándares de la IUB / IUPAC, con estas excepciones: se aceptan letras minúsculas y se mapean en mayúsculas; se puede usar un solo guion para representar un

espacio de longitud indeterminada (gap); y en las secuencias de aminoácidos, U y * son letras y caracteres aceptables.

Los códigos de ácidos nucleicos son:

TABLA 7. CÓDIGOS DE ÁCIDOS NUCLEICOS ACEPTADOS EN FORMATO FASTA

Código ácido nucleico	Significado
A	Adenosina
C	Citosina
G	Guanina
T	Timidina
U	Uracilo
R	G A (pu**R**ina)
Y	T C (pirimidina/p**Y**rimidine)
K	G T (cetona/**K**etone)
M	A C (grupo a**M**ino)
S	G C (interacción fuerte/**S**trong interaction)
W	A T (interacción débil/**W**eak interaction)
B	G T C (no A) (**B** viene tras la A)
D	G A T (no C) (**D** viene tras la C)
H	A C T (no G) (**H** viene tras la G)
V	G C A (no T, no U) (**V** viene tras la U)
N	A G C T (cualquiera/a**N**y)
-	hueco (gap) de longitud indeterminada

Los códigos para aminoácidos son:

TABLA 8. CÓDIGOS DE AMINOÁCIDOS ACEPTADOS EN FORMATO FASTA

Código de aminoácido	Significado
A	Alanina
B	Ácido aspártico o Asparagina
C	Cisteína
D	Ácido aspártico
E	Glutamato
F	Fenilalanina
G	Glicina
H	Histidina
I	Isoleucina
K	Lisina
L	Leucina
M	Metionina
N	Asparagina
P	Prolina
Q	Glutamina
R	Arginina
S	Serina
T	Treonina
U	Selenocisteína
V	Valina
W	Triptófano
Y	Tirosina
Z	Glutamato o Glutamina
X	Cualquier residuo de aminoácido
*	paro de traducción
-	hueco (gap) de longitud indeterminada

Para comprobar que también es posible manipular archivos en este formato: se crea un archivo en formato FASTA muy sencillo y se apreciará que es creado con éxito en nuestro ordenador, con lo cual se vislumbrará que es posible manipular estos archivos tan comunes cuando se va adentrando en el mundo de la bioinformática. Se trabajará con la secuencia antes mostrada en la explicación del formato, la cual corresponde a una fracción del genoma completo del SARS-CoV-2:

```
from io import open

archivo_fasta_SARS = open("secuencia_SARS.fasta","w")

sec_SARS = ">NC_045512.2 Severe acute respiratory syndrome
coronavirus 2 isolate Wuhan-Hu-1, complete genome \n
ATTAAAGGTTTATACCTTCCCAGGTAACAAACCAACCAACTTTCGATCTCTTGTAGATCTGTTC
TCTAAA"

archivo_fasta_SARS.write(sec_SARS)

archivo_fasta_SARS.close()
```

Al ejecutar el código, verificar la creación del archivo. Al colocar el puntero sobre el mismo se observará que su extensión será FASTA:

ILUSTRACIÓN 54. CREACIÓN DE ARCHIVO EN FORMATO FASTA

Para comprobar que el nuevo archivo creado es reconocido como un FASTA se ejecutó en MEGAX64*, el cual reconoció los datos de la secuencia y la secuencia misma, leyendo de manera correcta el archivo:

Para poder realizarlo de la manera tradicional (abriendo el archivo desde fuera) o desde la terminal de Python (en modo interactivo o en modo script) hay que hacerlo de la siguiente manera:

```
import os
os.startfile("C:\Archivos de programa\MEGA-
X\secuencia_SARS.FASTA")
```

Nota: la ubicación del archivo puede ser dentro de la misma carpeta destinada al curso. Al contar con MEGAX o cualquier otro programa que pueda ejecutar archivos de extensión *.FASTA no es necesario especificar con qué programa se abrirá el archivo ni tenerlo dentro de la carpeta del programa, sólo se colocó la ruta de manera ilustrativa.

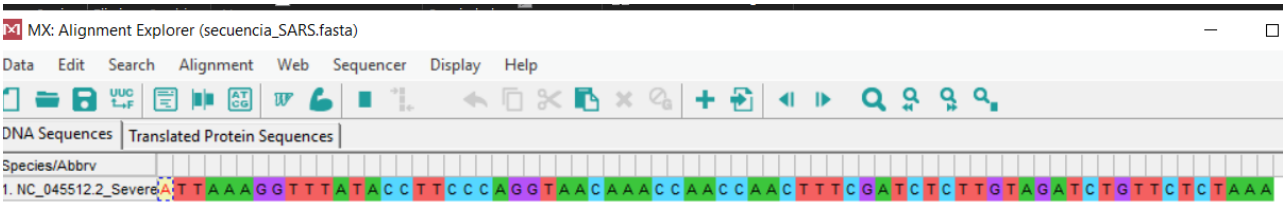


ILUSTRACIÓN 55. VISUALIZACIÓN DE LA APERTURA DEL ARCHIVO FASTA EN MEGAX

*MEGA (Molecular Evolutionary Genetics Analysis) es un software de computadora para realizar análisis estadísticos de la evolución molecular y para construir árboles filogenéticos, entre otras funciones. Si deseas conocer más sobre el proyecto y el uso del software, así como descargar el programa, puedes acceder en el siguiente enlace a la página oficial: https://www.megasoftware.net/

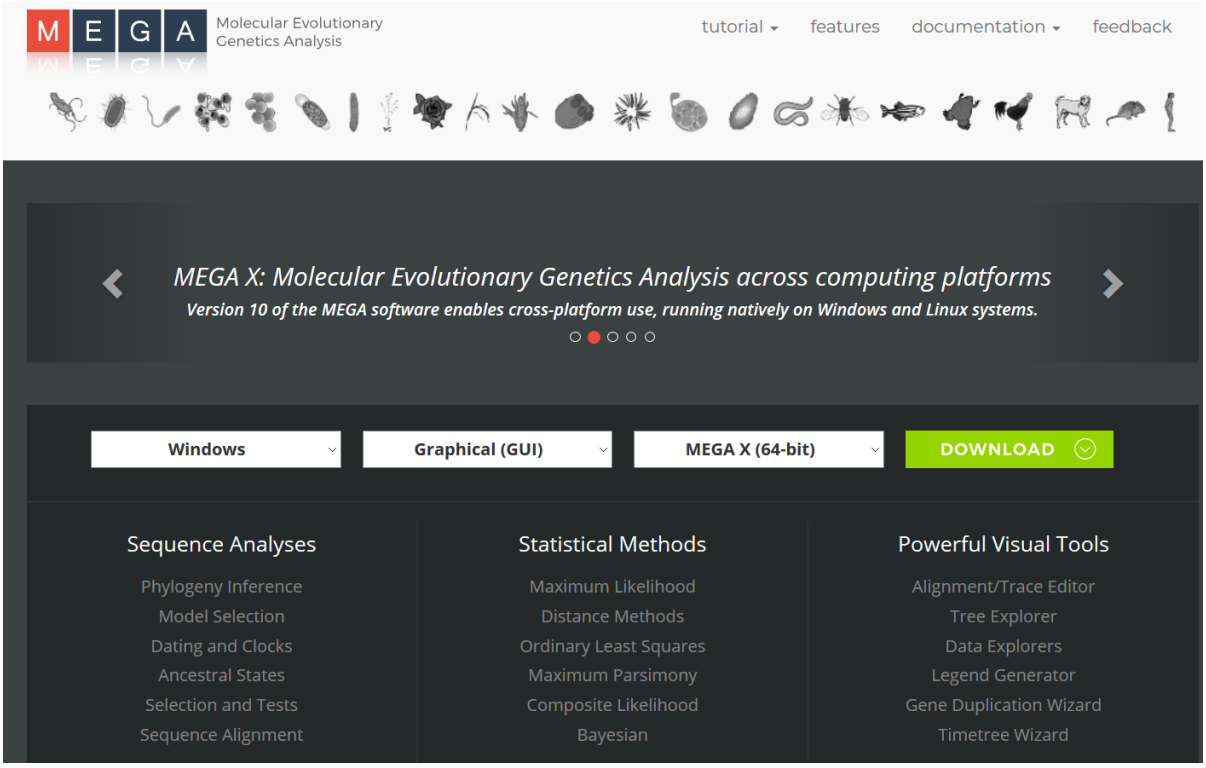


ILUSTRACIÓN 56. PÁGINA DE INICIO DE MEGA

`with` : Una alternativa para manipular archivos

Desde la versión de Python 2.6, se ha ofrecido a los usuarios una nueva herramienta que se puede emplear para abrir de manera segura los archivos: `with`. La sintaxis general toma esta forma:

```
with EXPRESIÓN as VARIABLE:
Bloque de código
```

Por ejemplo:

```
with open("fichero.txt") as f:
    print(f.read())
```

Cuando se ejecuta esta instrucción, se evalúa la expresión contenida. Se llama al método del objeto devuelto por la expresión. Lo que sea devuelto se llama variable. El código en bloque o los bloques siguientes serán ejecutados. Cuando se termina (por extensión del bloque o por un error), se llama al método `__exit__` por default. Este método puede manejar excepciones, por lo que, si hay un error dentro de bloque, el programador tiene la seguridad

de que aún se ejecutará el código. Este código podría usarse para cerrar un recurso abierto sin necesidad de especificarlo. A continuación, se observará un ejemplo que incluye dentro de su estructura la aplicación de la herramienta `with`. Para este ejemplo será necesario descargar la secuencia genómica de SARS-CoV-2, la cual se encuentra disponible en el siguiente enlace: https://www.ncbi.nlm.nih.gov/nuccore/1798174254 , se accedió al formato FASTA y se guardó en un block de notas como un archivo con extensión *.txt

```
>>>def read_seq(inputfile):
    #Leer archivo de DNA y eliminar espacios y saltos de línea

    with open(inputfile,"r") as f:
        seq = f.read()
    seq = seq.replace("\n", "")
    seq = seq.replace("\r", "")
    return seq
... ... ... ... ... ... ... ... ... ...
>>> def translate(seq):
    #Traducción de DNA a proteína

    Tabla_traduccion = {
    'ATA':'I', 'ATC':'I', 'ATT':'I', 'ATG':'M',
    'ACA':'T', 'ACC':'T', 'ACG':'T', 'ACT':'T',
    'AAC':'N', 'AAT':'N', 'AAA':'K', 'AAG':'K',
    'AGC':'S', 'AGT':'S', 'AGA':'R', 'AGG':'R',
    'CTA':'L', 'CTC':'L', 'CTG':'L', 'CTT':'L',
    'CCA':'P', 'CCC':'P', 'CCG':'P', 'CCT':'P',
    'CAC':'H', 'CAT':'H', 'CAA':'Q', 'CAG':'Q',
    'CGA':'R', 'CGC':'R', 'CGG':'R', 'CGT':'R',
    'GTA':'V', 'GTC':'V', 'GTG':'V', 'GTT':'V',
    'GCA':'A', 'GCC':'A', 'GCG':'A', 'GCT':'A',
    'GAC':'D', 'GAT':'D', 'GAA':'E', 'GAG':'E',
    'GGA':'G', 'GGC':'G', 'GGG':'G', 'GGT':'G',
    'TCA':'S', 'TCC':'S', 'TCG':'S', 'TCT':'S',
    'TTC':'F', 'TTT':'F', 'TTA':'L', 'TTG':'L',
    'TAC':'Y', 'TAT':'Y', 'TAA':'_', 'TAG':'_',
    'TGC':'C', 'TGT':'C', 'TGA':'_', 'TGG':'W',
    }
```

```
def translate(seq):
    #Traducción de DNA a proteína#
    return"".join([tabla_traduccion[seq[i:i+3]] for i in range (0, len(seq),
3) if i + 3 <= len(seq)])
... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ... ...

>>> dna = read_seq("SARS.txt") #Guardar el archivo en formato texto (block
notas)#

>>> dna
#con esto se comprueba que lee nuestro archivo y aparecerá la secuencia sin
saltos de línea ni espacios en blanco. En caso de no ser leído o encontrado
significa que no se encuentra en el directorio adecuado y se debe modificar
su lugar de almacenamiento o incluir la ruta completa#

>>> translate (dna)
#Deberá mostrarnos la traducción a aminoácidos##
```

Así es como se podrá visualizar en la consola un ejemplo que emplea el código anterior:

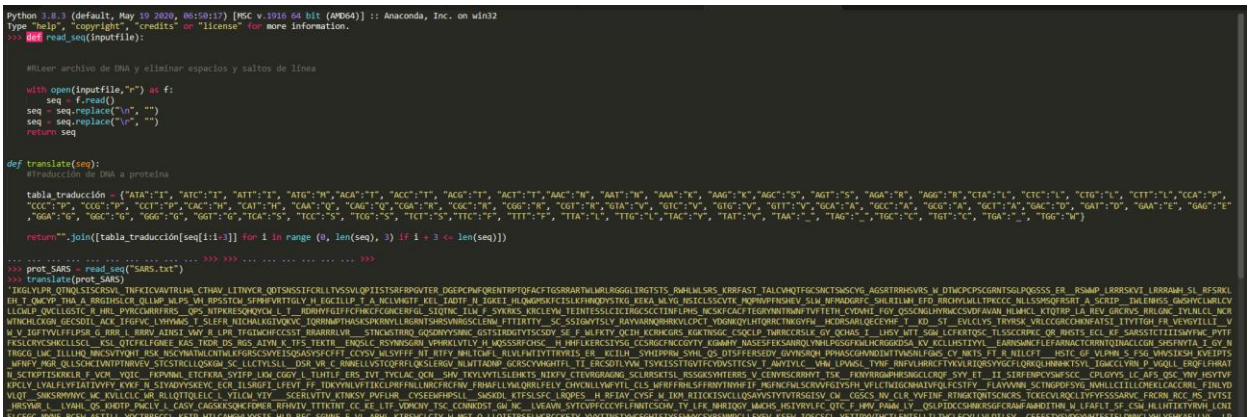


ILUSTRACIÓN 57. VISUALIZACIÓN DE EJECUCIÓN DEL CÓDIGO DE TRADUCCIÓN DE SECUENCIAS

Serialización: **`Pickle`**

Consiste en guardar en un archivo externo un diccionario o un objeto o etc. Este objeto se guardará en código binario (bases de datos, internet, disco duro). Deberá emplearse la biblioteca `pickle`. Las funciones básicas en `pickle` son:

- `dump()` que sirve para volcado de datos al fichero binario externo
- `load()` útil para carga de datos del fichero binario externo

```
import pickle

lista_medicamentos = ["Naproxeno", "APAP", "Ivermectina",
"Hidroxicloroquina"]

fichero_binario = open("lista_medicamentos","wb")#escritura
binaria con wb

pickle.dump(lista_medicamentos, fichero_binario)#vaciado de
datos

fichero_binario.close()

del (fichero_binario) #esto es para eliminarlo de la memoria
```

Se creó en la carpeta, pero no puede leerse, para recuperarlo y leerlo:

```
import pickle

fichero = open("lista_medicamentos", "rb") #leer binario

lista = pickle.load(fichero)

print(lista)
```

Un ejemplo que puede aplicarse sería el de crear archivos con listas de pacientes enfermos, características principales de los mismos e incluso realizar búsquedas de acuerdo con criterios específicos. Para poner en práctica todos los conocimientos que se han adquirido hasta ahora:

```
import pickle

class Paciente:

     def __init__(self, nombre, genero, edad, enfermedad):
          self.nombre = nombre
          self.genero = genero
          self.edad = edad
          self.enfermedad = enfermedad

          print("Búsqueda de pacientes por enfermedad. Enfermedad que
          presenta: ", self.enfermedad)

     def __str__(self):
          return "{} {} {}".format(self.nombre, self.genero, self.edad,
          self.enfermedad)

class ListaPacientes:

     pacientes = []

     def __init__(self):
          listaDePacientes = open("FicheroExterno", "ab+")
          listaDePacientes.seek(0)#colocación del cursos para
          cargar   info en el fichero#

          try:  #Esto es para poder ir monitoreando la info cada que se
          meten datos#

               self.personas.pickle.load(listaDePacientes)
               print("Se cargaron {} pacientes del
               fichero externo.".format(len(self.personas)))

          except:
               print("Fichero vacío")

          finally:
               listaDePacientes.close()
               del(listaDePacientes)


     def agregarPaciente(self, p):
          self.pacientes.append(p)
          self.guardarPacientesFicheroExt()


     def mostarPacientes(self):
          for p in self.personas:
               print(p)
```

```
     def mostarPacientes(self):
          for p in self.personas:
               print(p)

     def guardarPacientesFicheroExt(self):
          listaDePacientes = open("FicheroExterno", "wb")
          pickle.dump(self.pacientes, listaDePacientes)
          del(listaDePacientes)
##############################################################
     def mostrarInfoFicheroExt(self):
          print("La información del fichero externo es: ")
          for p in self.pacientes:
               print(p)

miLista = ListaPacientes()

paciente = Paciente("Mariana", "Femenino", 22, "COVID-19")
miLista.agregarPaciente(paciente)
miLista.mostrarInfoFicheroExt()

#Al ejecutar el REPL se obtiene lo siguiente:

Output:
Fichero vacío
Búsqueda de pacientes por enfermedad. Enfermedad que presenta:
COVID-19
La información del fichero externo es:
Mariana Femenino 22
```

Guardado permanente

Los datos almacenados en archivos de texto son adecuados para muchas aplicaciones. Sin embargo, como la cantidad y la complejidad de los datos aumenta, resulta demasiado complicado emplear sólo texto. Esto es especialmente cierto cuando una aplicación accede de forma selectiva a pequeñas cantidades de datos de un volumen mucho más grande. Para estos casos, Python proporciona tres variantes de almacenamiento persistente o permanente. En el código anterior, el guardado permanente (en binario y en ficheros externos que pueden consultarse después y manipular desde cualquier programa) se puede apreciar, principalmente, en estas partes del código:

```
. . .
def __init__(self):
            listaDePacientes = open("FicheroExterno", "ab+")
            listaDePacientes.seek(0)

. . .

. . .
def guardarPacientesFicheroExt(self):
            listaDePacientes = open("FicheroExterno", "wb")
            pickle.dump(self.pacientes, listaDePacientes)
            del(listaDePacientes)
. . .
```

En la primera sección de código es donde se crea el fichero externo con métodos de adición en binario y se indica el inicio de la lectura de la información de dicho fichero externo desde el inicio del archivo en lugar de la última sección añadida. En la segunda sección de código se indica que la información del código (la lista de los pacientes) deberá de ser introducida (escrita en binario) al fichero antes creado. También indica que la información en consola que ya fue guardada se elimine, con la finalidad de no sobrecargar la memoria y poder tener un espacio de trabajo limpio. En el código también se incluye la lectura del fichero, con el cual se puede monitorear la información que se va introduciendo. El fichero es permanente y sólo puede eliminarse externamente si el usuario así lo desea.

Capítulo XIV. Expresiones Regulares (REGEX)

Ya se ha visto como manipular strings en Python, una de las operaciones más frecuentes o que más pueden emplearse es la de es buscar una cierta cadena, patrón, secuencia, etc., en una lista, tabla o fichero de texto. Esto no supone ninguna dificultad si lo que se está buscando es estático y es conocido con precisión. Por ejemplo, para encontrar un nombre determinado en una lista de contactos, basta con usar funciones como `find()`, la cual ya se ha trabajado anteriormente en el tema "Funciones Integradas".

Pero ¿qué pasa cuando el dato que se está tratando de localizar, o bien no es conocido con exactitud, o bien presenta variantes en su escritura? ¿Y si en una secuencia binaria se quiere encontrar todas las subsecuencias? Ahora, ya no es factible emplear la función `find()`. Es aquí es donde entran en escena las expresiones regulares, una herramienta muy potente que facilita la búsqueda de patrones en un texto. Una característica común de todos los lenguajes de scripting es el soporte de expresiones regulares (REGEX en el área de programación). ¿Qué son las expresiones regulares? Son expresiones que resumen un patrón de texto. Un caso conocido de expresión regular son las abreviaturas que se usan en la mayoría de los sistemas operativos, como usar "ls * .py" (o "dir * .py") para enumerar todos los archivos que terminan en ".py". Estos se conocen como comodines. Cuando se procesa texto, a menudo es necesario dar un tratamiento especial a las cadenas o strings que contienen una condición específica. Por ejemplo, es posible extraer todo lo que esté entre <pre> y </pre> en un archivo HTML, o eliminar de un archivo cualquier carácter que no sea A, T, C o G.

Las aplicaciones biológicas de esta característica son sencillas. Las expresiones regulares se pueden usar para localizar dominios en proteínas, patrones de secuencia en ADN como islas CpG, repeticiones, enzimas de restricción, sitios de reconocimiento de nucleasas, etc. Incluso existen bases de datos biológicas dedicadas a dominios de proteínas, como prosite. Para acceder al sitio se puede hacer mediante el siguiente enlace: https://prosite.expasy.org/

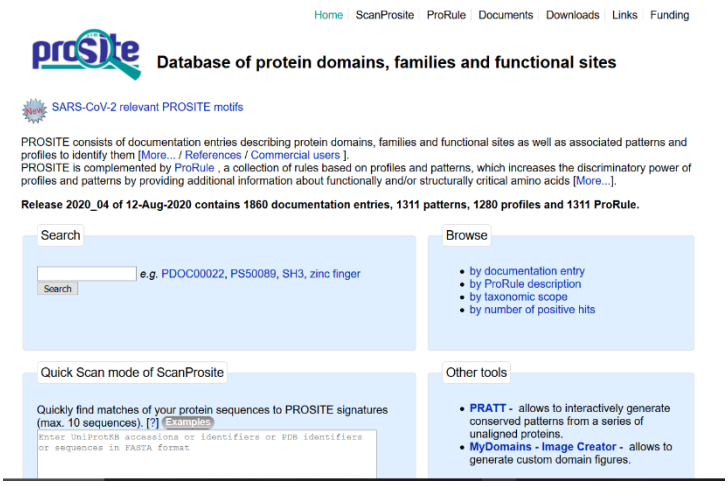


ILUSTRACIÓN 58. PÁGINA DE INICIO DE PROSITE

Sin embargo, las necesidades personales de programación pueden no incluir el uso de expresiones regulares. En este caso, se puede omitir este capítulo final. Sin embargo, en caso de querer probarlo por curiosidad o para seguir mejorando nuestra habilidad en el lenguaje se puede poner en práctica los ejemplos que aparecen a continuación.

Cada idioma tiene su propia sintaxis REGEX. En Python, esta sintaxis es similar a la que se usa en Perl. Aunque la sintaxis de REGEX no es tan difícil de aprender, algunos REGEX podrían convertirse en expresiones confusas y complejas en casos específicos. Debido a esta potencial complejidad, incluso existen libros completos sobre este tema:

ILUSTRACIÓN 59. LIBROS DISPONIBLES EN LA WEB PARA APRENDER A EMPLEAR REGEX

Como dato curioso, en SublimeText existe la opción de realizar búsquedas con expresiones regulares dentro del código. Se debe usar la combinación de teclas ctrl + f (para buscar) y para activar la función (y muchas otras) tan sólo dando clic sobre la que se quiere activar:

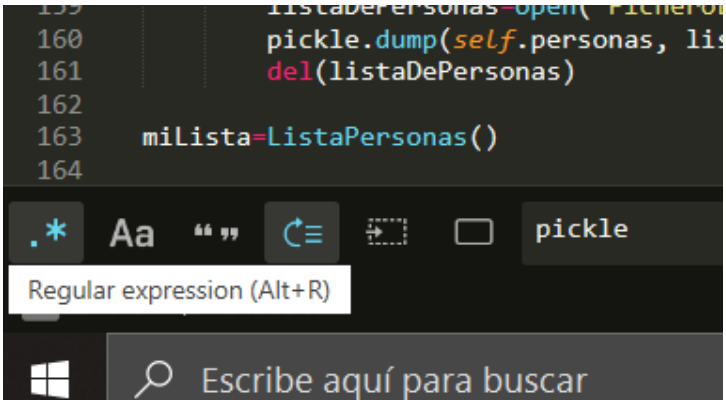


ILUSTRACIÓN 60. HERRAMIENTA DE REGEX EN BARRA DE BÚSQUEDA DE SUBLIME TEXT

Sintaxis de REGEX

En general, las letras y los caracteres coinciden con ellos mismos. "Python" va a coincidir con "Python" (pero no con "python"). Las excepciones a esta regla son los metacaracteres, que son caracteres que tienen un significado especial en el contexto del REGEX: `. ^ $ * +? { [ ] \ | ( )`

Estos son algunos de los caracteres más comunes en REGEX:

TABLA 9. EXPRESIONES REGULARES COMUNES EN PYTHON

Carácter/Expresión y nombre	Función/Ejemplo
. (punto):	Coincide con cualquier carácter, excepto la nueva línea: "ATT.T" coincidirá con "ATTCT", "ATTFT" pero no con "ATTTCT".
ˆ (carat (en inglés):	Coincide con el comienzo de la cadena: "ˆAUG" coincidirá con "AUGAGC" pero no con "AAUGC". Usarlo dentro de un grupo significa "opuesto".
$ (dólar):	Coincide con el final de la cadena o justo antes de una nueva línea al final de la cadena: "UAA $"

	coincidirá con "AGCUAA" pero no con "ACUAAG".
* (estrella/asterisco):	Coincide con 0 o más repeticiones del token anterior: "AT *" coincidirá con "AAT", "A", pero no con "TT".
? (signo de interrogación):	El REGEX resultante coincide con 0 o 1 repeticiones del RE anterior. "¿A?" coincidirá con "A" o "AT".
(...):	Coincide con cualquier expresión regular que esté dentro del paréntesis, y indica el inicio y el final de un grupo. Para hacer coincidir los literales "(" o ")", utilizar\ (o \), o deberán ser encerrados dentro de una clase de caracteres: [(] [)].
(?: ...):	una versión no agrupada de paréntesis regulares. La subcadena emparejada por el grupo no se puede recuperar después de realizar un match.
{n}:	Exactamente n copias del REGEX anterior coincidirán: "(ATTG) {3}" coincidirá con "ATTGATTGATTG" pero no con "ATTGATTG".
{m, n}:	El REGEX resultante coincidirá de m a n repeticiones del REGEX anterior: "(AT) {3,5}" coincidirá con "ATATTATATAT" pero no “ATATTATAT”. Sin m, coincidirá a partir de 0 repeticiones. Sin n, este coincidirá con todas las repeticiones.
[] (corchetes):	Indican un conjunto de caracteres. "[A-Z]" coincidirá con cualquier letra mayúscula y “[a-z0-9]” coincidirá con cualquier letra minúscula o dígito. Los metacaracteres no están activos dentro de los conjuntos REGEX. "[AT *]" coincidirá con "A", "T" o "*". El ˆdentro de un conjunto calculará el complemento de un conjunto. "[ˆR]" coincidir con cualquier carácter excepto "R".
\” (Barra invertida o backslash):	Se utiliza para escapar de los caracteres reservados (para hacer coincidir los caracteres como "?", "*"). Dado que Python también usa la barra invertida como carácter de escape, debe pasar una cadena sin procesar para expresar el patrón.
| (barra vertical):	Como en expresión lógica, se lee como "o". Cualquier cantidad de REGEX puede estar separados por "|". “A | T” coincidirá con “A”, “T” o “AT”.

Módulo **re**

El módulo re proporciona métodos como compilar, buscar, encontrar, hacer coincidir y otros. Estas funciones se utilizan para procesar un texto utilizando un patrón construido con la sintaxis REGEX. Una búsqueda básica funciona como a continuación:

```
>>> import re
>>> ejemplo = re.search("UNAM", "UNAM, FES Cuautitlán Campo 1")
```

El método de búsqueda desde re requiere un patrón como primer argumento y como segundo argumento, una cadena donde se buscará el patrón. En este caso, el patrón se puede traducir como "U, seguido de NAM". Cuando se encuentra una coincidencia, esta función devuelve un objeto de coincidencia (llamado ejemplo en este caso) con información sobre la primera coincidencia. Si no hay coincidencia, devuelve `None`. Se puede consultar un objeto de coincidencia con los siguientes métodos:

```
>>> ejemplo.group()

Output:'UNAM'

>>> ejemplo.span()

Output:(0, 4)
```

`group ()` devuelve la cadena que coincide con REGEX, mientras que `span ()` devuelve una tupla que contiene las posiciones (inicio, final) de la coincidencia (es decir, el (0,) devuelto por ejemplo`.span ().`

Para encontrar todas las coincidencias, y no solo la primera, se usará `findall()`:

```
>>> re.findall("[Hh]ola","Hola, hola, yo estudio en la FESC!")

Output:['Hola', 'hola']
```

Para obtener un objeto para cada coincidencia, existe el método `finditer`. Como ventaja adicional, no devuelve una lista, sino un iterador. Esto significa que cada vez que se invoca `finditer`, devuelve el siguiente elemento sin tener que calcularlos todos a la vez. Como con cualquier iterador, esto optimiza el uso de la memoria.

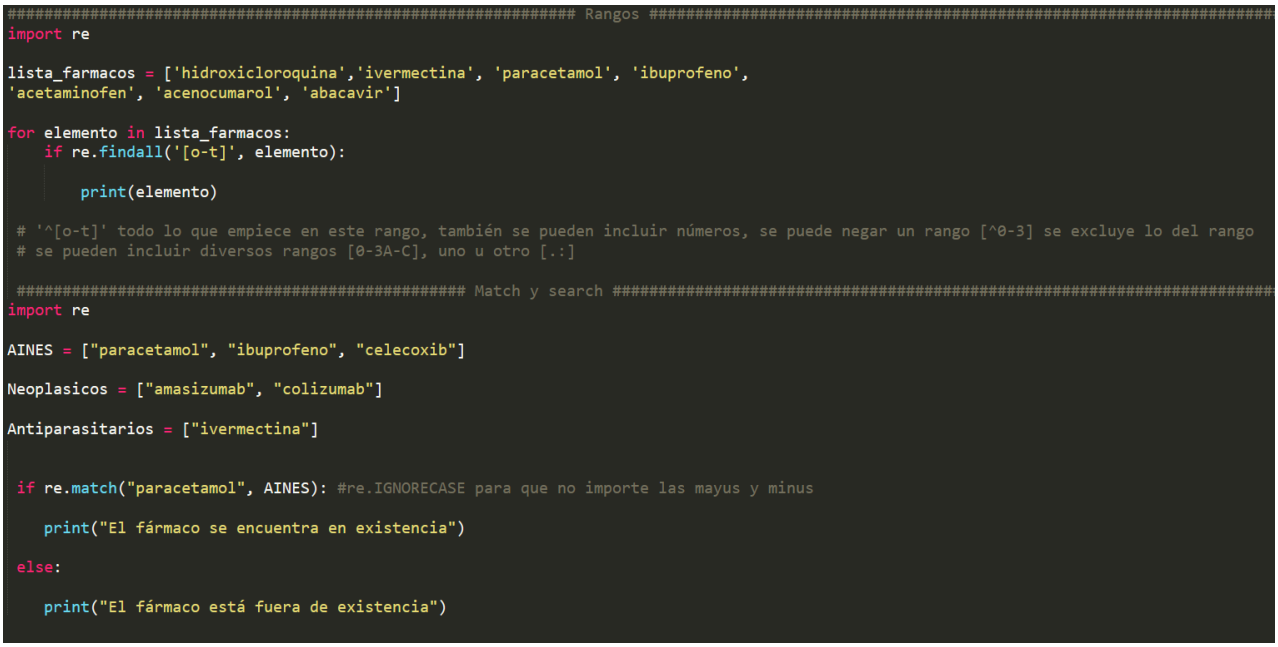
Algunos ejemplos con secuencias:

```
>>> import re
>>> sec = "ATATAAGATGCGCGCGCTTATGCGCGCA"
>>> rgx = re.compile("(GC){3,}")
>>> resultado = rgx.search(sec)
>>> result.group()

Output:
'GCGCGCGC'
```

A veces es necesario hacer coincidir más de un patrón; esto se puede hacer agrupando. Los grupos están marcados con un par de paréntesis (“()”). Pueden ser "capturadores" ("con nombre" o "sin nombre") y "no capturadores". La diferencia entre ellos quedará clara a continuación. Un grupo de "captura" se utiliza cuando necesita recuperar el contenido de un grupo. Los grupos se capturan con grupos. No hay que confundir `group()` con `groups()`. Los grupos devuelven una tupla con todos los subgrupos de la coincidencia. En este caso, dado que la búsqueda devuelve una coincidencia y hay un grupo en el patrón, el resultado es una tupla con un grupo:

```
>>> result.groups()

Output: ('GC',)
```

Otros ejemplos más, relacionados con el área farmacéutica pudieran ser para la administración de inventarios (incluyendo funciones REGEX dentro del programa) en un laboratorio o de medicamentos en una farmacia u hospital:



```
############################################################ Rangos ##########################################################
import re

lista_farmacos = ['hidroxicloroquina','ivermectina', 'paracetamol', 'ibuprofeno',
'acetaminofen', 'acenocumarol', 'abacavir']

for elemento in lista_farmacos:
    if re.findall('[o-t]', elemento):

        print(elemento)

 # '^[o-t]' todo lo que empiece en este rango, también se pueden incluir números, se puede negar un rango [^0-3] se excluye lo del rango
 # se pueden incluir diversos rangos [0-3A-C], uno u otro [.:]

 ############################################### Match y search ###################################################################
import re

AINES = ["paracetamol", "ibuprofeno", "celecoxib"]

Neoplasicos = ["amasizumab", "colizumab"]

Antiparasitarios = ["ivermectina"]


 if re.match("paracetamol", AINES): #re.IGNORECASE para que no importe las mayus y minus

    print("El fármaco se encuentra en existencia")

 else:

    print("El fármaco está fuera de existencia")
```

ILUSTRACIÓN 61. EJEMPLO DE USO DE EXPRESIONES REGULARES

¿Se te ocurre alguna otra manera interesante de usar REGEX? Las mutaciones en humanos, animales, virus o bacterias podría ser un campo interesante para llevar a cabo la aplicación de herramientas tales como REGEX. Por ejemplo:

Muchos trastornos neurodegenerativos hereditarios humanos, como la enfermedad de Huntington (EH), se han relacionado con la expansión anómala del número de repeticiones de trinucleótidos en genes particulares. La gravedad patológica de la EH se correlaciona con el número de (CAG) n repeticiones en el exón-1 del gen ht, que codifica la proteína Huntington. En la enfermedad, un mayor número de repeticiones significa un inicio más temprano de la enfermedad y una progresión más rápida de la misma. El codón CAG especifica la glutamina y la EH pertenece a una amplia clase de enfermedades por poliglutamina. Las variantes sanas (wild-type) de este gen presentan entre 6 y 35 repeticiones en tándem, mientras que más de 35 repeticiones prácticamente aseguran la enfermedad. Se pueden usar expresiones regulares para descifrar el número de repetición de poliglutamina. En primer lugar, esto implica escribir un patrón para encontrar el número de repetición de trinucleótidos por encima de un umbral establecido.

Puedes intentar crear tus propios ejercicios, ejemplos, programas o incluso aplicaciones que pueden ayudarte a resolver problemas propios o problemas que enfrenta la sociedad actual.

Capítulo XV. Interfaces gráficas y `Tkinter`

Otro módulo interesante que puede ayudar en la iniciación de creación archivos ejecutables (programas de interfaz propia) y añadir más complejidad al código es `Tkinter`, con el cual se crean interfaces gráficas. También son intermediados entre el programa y el usuario. Se forman por un conjunto de gráficos, ventanas, casillas, etc.

Algunos otros módulos y paquetes disponibles con funciones y/o características similares:

- WxPython
- PyQT
- PyGTK

`Tkinter` es un puente entre Python y la librería `TCL/TK` para Linux: `$sudo apt-get install python3-tk`, aunque por lo general viene instalada por defecto.

Los comandos básicos que deberán de conocerse en `tkinter` y un par de datos de la sintaxis general son:

`root = Tk()`: Es lo que se debe de construir, o sea, es la ventana.
Frame: es el organizador de widgets, son los aglutinantes y personalizables.
La manera de importar el módulo es como ya se ha visto antes: `from tkinter import*`
`root.mainloop()`: es la indicación de permanecer en un bucle infinito que mantiene en ejecución la interfaz. Deberá de estar a la escucha de eventos para poder interactuar con el usuario.

Creación de la primera ventana sin especificaciones:

```
From tkinter import*
root = Tk()
#Cuerpo con bloque de código de personalización y/o
funcionalidad
raiz.mainloop()
```

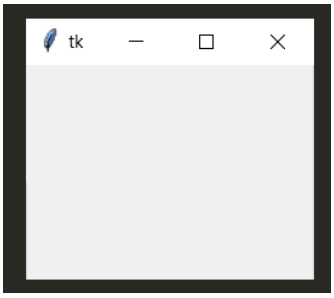


ILUSTRACIÓN 62. PRIMERA VENTANA CREADA CON TKINTER

Para una ventana que ya demuestra inicios de personalización:

```
#Para cambiar el ícono es necesario poseer la Ilustración en
formato .ico, también puede convertirse una Ilustración normal pero
debe ser pequeña

from tkinter import*

root = Tk()

root.title("Mi primera ventana 😊")#Método de título

root.resizable(15,20)#Metodo booleano(números) El primer número es
ancho, el segundo alto (Si se emplea True/False también sirve)#

root.iconbitmap("gato.ico")#agregar íconos#

root.geometry("650x350") #Medidas por defecto#

root.config(bg = "pink") #Esto es para cambiar el color del fondo#

root.mainloop() ##Esto debe estar SIEMPRE AL FINAL
###Para que se abra directamente la ventana sin otra cmd se debe
cambiar la extensión a .pyw(Python-window)
```

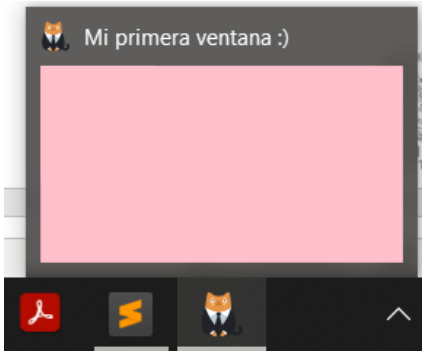


ILUSTRACIÓN 63. PERSONALIZACIÓN DE UNA VENTANA Y APRECIACIÓN DEL ÍCONO

Configurando la raíz y añadiendo un frame personalizable:

```
from tkinter import*

root = Tk()

root.title("Mi primera ventana 😊")

root.resizable()

root.iconbitmap("gato.ico")

#root.geometry("650x350") #Se tiene que tomar en cuenta la
adaptación de contenido para especificar tamaño#

root.config(bg = "pink")

root.config(bd = 15) #Medida de ancho del borde. Esto debe de ir
antes que el tipo de borde#

root.config(relief = "sunken")#Tipo de borde#

root.config(cursor = "hand2")#Configura el cursor al posicionarlo
sobre la root#

miFrame=Frame() #Hay que empaquetar el frame en la root#
```

```
miFrame.pack(side = "left", anchor = "n")#Con esto se empaqueta
para incluirlo en la ventana#

#Se puede definir dónde se colocará el frame en la configuración
Anchor es para puntos cardinales (fill = "y", expand = True)
llena con el frame uno de los ejes o ambos "x", "y", "both",
"none" X no lleva la función de expand#

miFrame.config(bg = "black")

miFrame.config(width = "650", height = "350") #Esta no se adapta,
es fijo#

miFrame.config(bd = 35) #Medida de ancho del borde. Esto debe de
ir antes que el tipo de borde#

miFrame.config(relief = "groove")#Tipo de borde

miFrame.config(cursor = "pirate")#Configura el cursor al
posicionarlo sobre el frame

root.mainloop()
```

ILUSTRACIÓN 64. PERSONALIZACIÓN DE VENTANA, PUNTERO Y CARACTERÍSTICAS DE MANIPULACIÓN

Adición de un `label`

`Label`: widgets usados para mostrar texto o imágenes. No se puede interactuar de ninguna forma con ellos, el ancho y el alto va en pixeles. Sintaxis general:

```
variableLabel = Label(contenedor, opciones)
```

Para crear una ventana con un nuevo label personalizable:

```
from tkinter import*

root = Tk()

miframe = Frame(root, width = 500, height = 400)#Al
ejecutar, no respeta las medidas porque se adaptó al Label#

miframe.pack()

miLabel = Label(miframe, text = "Hola, espero disfrutaras
el manual")

miLabel.pack()

root.mainloop()
```

Por último, para demostrar un poco más de lo que es posible lograr si se sigue incursionando en el mundo de la programación y de Python, se elaborará una ventana donde se incluirá entrada de texto (interacción con el usuario). A esta ventana también se le pueden añadir botones y más características que se pueden descubrir en: https://docs.python.org/3/search.html?q=tkinter

```
from tkinter import*

raiz = Tk()

miFrame = Frame(raiz, width = 1800, height = 900)
miFrame.pack()

cuadroTexto = Entry(miFrame)
cuadroTexto.grid(row = 0, column = 1)

cuadroApellido = Entry(miFrame)
cuadroApellido.grid(row = 1, column = 1)

cuadroDireccion = Entry(miFrame)
cuadroDireccion.grid(row = 2, column = 1)

cuadroUsuario = Entry(miFrame)
cuadroUsuario.grid(row = 3, column = 1)

nombreLabel = Label(miFrame, text = "Nombre: ")
nombreLabel.grid(row = 0, column = 0, sticky = "e")

apellidoLabel = Label(miFrame, text = "Apellido: ")
apellidoLabel.grid(row = 1, column = 0, sticky = "e")

direccionLabel = Label(miFrame, text = "Dirección de casa:
")
direccionLabel.grid(row = 2, column = 0, sticky = "e")

usuarioLabel = Label(miFrame, text = "Nombre Usuario
(nuevo): ")
usuarioLabel.grid(row = 3, column = 0, sticky = "e")

PassLabel = Label(miFrame, text = "Contraseña: ")
PassLabel.grid(row = 4, column = 0, sticky = "e", padx =
15, pady = 15)
##Hay que tener bien determinados y ordenados nuestras
columnas y filas

cuadroPass = Entry(miFrame)
cuadroPass.grid(row = 4, column = 1)
cuadroPass.config(show = "*")#Sirve para mostrar (u
ocultar) ciertos caracteres

raiz.mainloop()
```

Hay que tomar en cuenta que:

- ✓ Widget `Entry`: para ingresar texto desde el usuario
- ✓ `grid()` sirve para hacer una tabla con las filas y columnas que se requieran
- ✓ `row` sirve para especificar la fila
- ✓ La propiedad `sticky` permite alinear las cosas como uno quiera, trabaja con puntos cardinales
- ✓ `column` es para asignar las columnas

Para dimensionar la utilidad de `tkinter` en Python, se propondrá la creación de una calculadora: es totalmente útil en el área de las ciencias químico-biológicas y de la salud porque está llena de cálculos en todas sus áreas. La calculadora puede ser simple o puede ser adaptada a alguna necesidad específica: cálculos de dosis, concentración de soluciones, porcentaje de GC, estadística, simulación de una calculadora científica, etc. Será necesaria la investigación de colocación de botones y algunas funciones extras que se puedan añadir.

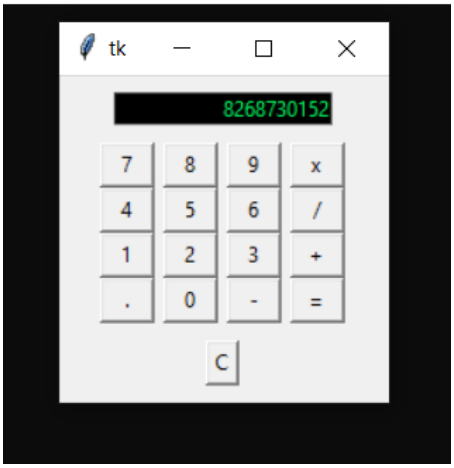


ILUSTRACIÓN 65. CREACIÓN DE CALCULADORA CON TKINTER

La calculadora es totalmente funcional. Hay que recordar que es totalmente personalizable: color, tamaño, distribución, nombre del programa, pantalla de resultados, etc. La recomendación es separar la creación del programa: importación de `tkinter` y paquetes que se consideren necesarios, asignar variables al resultado y a la pantalla inicial (iniciar en 0), crear las funciones para cada operación y para el funcionamiento de dichos botones, así como la creación del frame separado por las filas que poseerá la calculadora. No hay que

olvidar colocar la parte final del código con el bucle que mantendrá en operación al programa con su interfaz gráfica.

Capítulo XVI. Documentación

La documentación en Python es muy útil para añadir comentarios en clases, métodos, módulos, etc., para ayudar al trabajo en equipo por parte del usuario creador a otros que usarán el código e incluso para nosotros mismos. Es especialmente útil en aplicaciones complejas. Inicialmente, la documentación pueden ser simples comentarios que informen acerca del funcionamiento de cada sección; puede funcionar en conjunto con el comando `help()` pero la información que va a ser presentada será aquella que ha sido incluida y es totalmente personalizada y definida por nosotros:

```
class Areas:

    """  Esta clase calcula áreas para diferentes figuras"""

    def areaCuadrado (Lado):

        """     Calcula el área de un cuadrado
        elevando al cuadrado el lado especificado como parámetro """

        return "El área del cuadrado es: " + str(lado*lado)

    def areaTriangulo(base, altura):

        return "El área del triángulo es: " + str((base*altura)/2)

print(Areas.areaCuadrado(3))
print(Areas.areaCuadrado.__doc__) #Así imprimimos los comentarios o documentación
help(Areas.areaCuadrado) #nos describe como funciona la función de acuerdo a la documentación específica incluida

print(Areas.areaTriangulo(8,10))
help(Areas)#ayuda general de clase (debemos inlcuir la clase a la que pertenecen las funciones con Clase.funcion)
```

ILUSTRACIÓN 66. INCLUSIÓN DE DOCUMENTACIÓN ÚTIL EN CÓDIGO

Al ejecutar, además de brindar el resultado de nuestra función, brindará la información de cómo funciona y cómo ésta compuesta la misma:

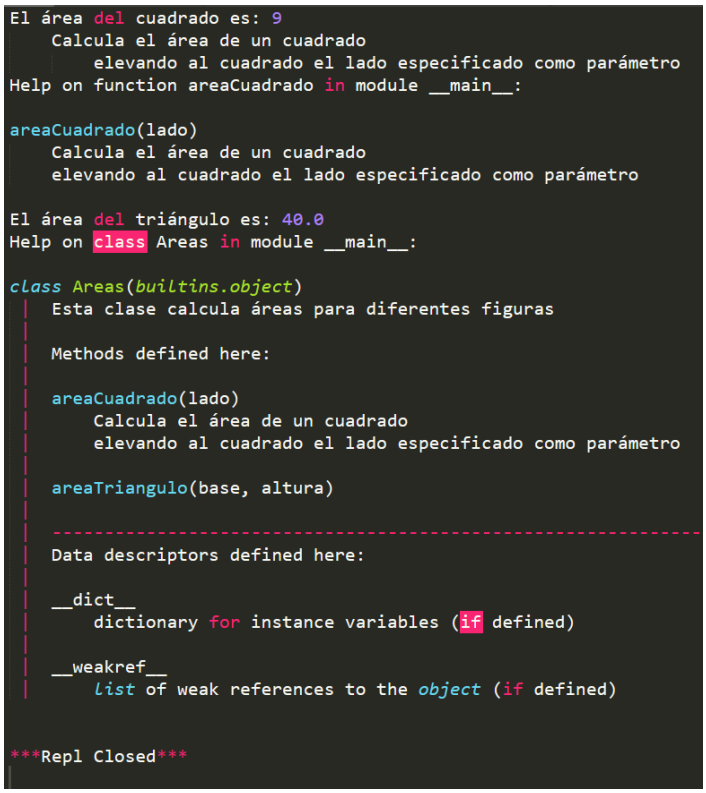


ILUSTRACIÓN 67. AYUDA RESULTANTE DE DOCUMENTACIÓN INCLUIDA EN EL CÓDIGO

Sin embargo, donde realmente radica la importancia de la documentación es en la realización pruebas con `doctest` para verificar si funciona todo correctamente antes de poner en marcha un programa. Las pruebas se realizan después del comentario, añadiendo símbolos simulando un prompr: `>>>` y un espacio en blanco + la función con parámetros y su resultado. Si todo es correcto no deberá aparecer nada, pero si hay algún error se notificará cual es y el resultado correcto esperado, además, se pueden hacer múltiples pruebas a la vez. A continuación, se puede observar como es definida una función para el cálculo del área de un triángulo y se añade dentro de la documentación varias pruebas. Al final se importa el módulo `doctest`. Si todo está bien, al ejecutar el código no se observará nada más que el cierre de REPL, sino, se notificará del error tanto la razón como la línea del código que lo está causando.

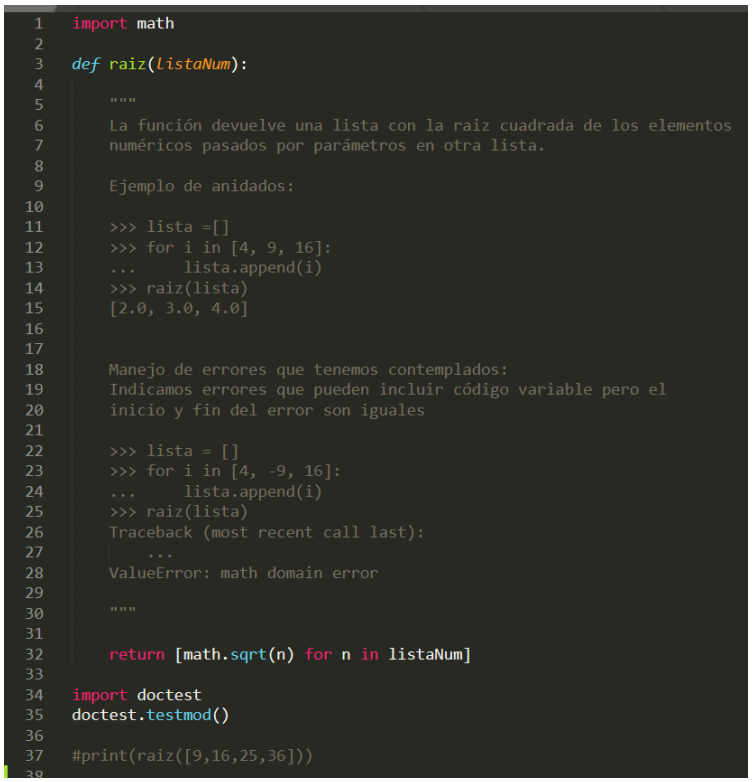
```
def areaTriangulo(base, altura):

    """
     Calcula el área de un triángulo dado

     >>> areaTriangulo(3,6)
     'El área del triángulo es: 9.0'

     >>> areaTriangulo(4,5)
     'El área del triángulo es: 10.0'

     >>> areaTriangulo(9,3)
     'El área del triángulo es: 13.5'

     """

    return "El área del triángulo es: " + str((base*altura)/2)

import doctest
doctest.testmod()
```

ILUSTRACIÓN 68. PRUEBAS DE FUNCIONAMIENTO CON DOCUMENTACIÓN INTERNA

Por último, también es posible añadir el error que se espera que pueda presentar el código. Para poder añadir el error, como puede variar, lo único que hay que escribir será el inicio y el final del error, por ejemplo:

```
Traceback (most recent call last):

          ...

    ValueError: math domain error
```

Por ejemplo, para una función que calcula raíces numéricas, se podrían introducir números negativos, donde el resultado de estos serían números imaginarios, más no negativos. Para poder manejar este error que ya fue contemplado, el código puede verse de la siguiente manera:



```python
import math

def raiz(listaNum):

    """
    La función devuelve una lista con la raiz cuadrada de los elementos
    numéricos pasados por parámetros en otra lista.

    Ejemplo de anidados:

    >>> lista =[]
    >>> for i in [4, 9, 16]:
    ...     lista.append(i)
    >>> raiz(lista)
    [2.0, 3.0, 4.0]


    Manejo de errores que tenemos contemplados:
    Indicamos errores que pueden incluir código variable pero el
    inicio y fin del error son iguales

    >>> lista = []
    >>> for i in [4, -9, 16]:
    ...     lista.append(i)
    >>> raiz(lista)
    Traceback (most recent call last):
        ...
    ValueError: math domain error

    """

    return [math.sqrt(n) for n in listaNum]

import doctest
doctest.testmod()

#print(raiz([9,16,25,36]))
```

ILUSTRACIÓN 69. MANEJO DE ERRORES Y EXCEPCIONES CON DOCUMENTACIÓN

Capítulo XVII. Bases de datos con Python: SQLite

Python es capaz de manejar bases de datos de otros sitios y locales. El lenguaje que nos permitirá manipular estos archivos es SQL (structures Query Language). El módulo `sqlite3` implementa interfaces compatibles con Python 2 y 3 y SQLite, la cual es una base de datos relacional al proceso. Una base de datos es un conjunto de datos que se almacenan de manera sistemática para poder ser usados de manera posterior. Los datos que son almacenados suelen poseer características en común. Una base de datos en SQLite se almacena como un solo archivo en el sistema de archivos. La biblioteca gestiona el acceso al archivo, incluido un bloqueo para evitar corrupción o manipulación indebida cuando varios usuarios con la autorización de editar lo utilizan. La base de datos se crea la primera vez que se accede al archivo, pero la aplicación es responsable de gestionar las definiciones de tabla o esquema, dentro de la base de datos.

MySQL y SQLite están escritos en C, son de código abierto, forman parte integral del programa, ocupan poco espacio y memoria, son gratis, multiplataforma, eficientes y rápidos.

Será necesario descargar un programa que permita la visualización, manipulación y creación de la base de datos, para esto se utilizará DB Browser para SQLite, el cual puede ser descargado desde: https://sqlitebrowser.org

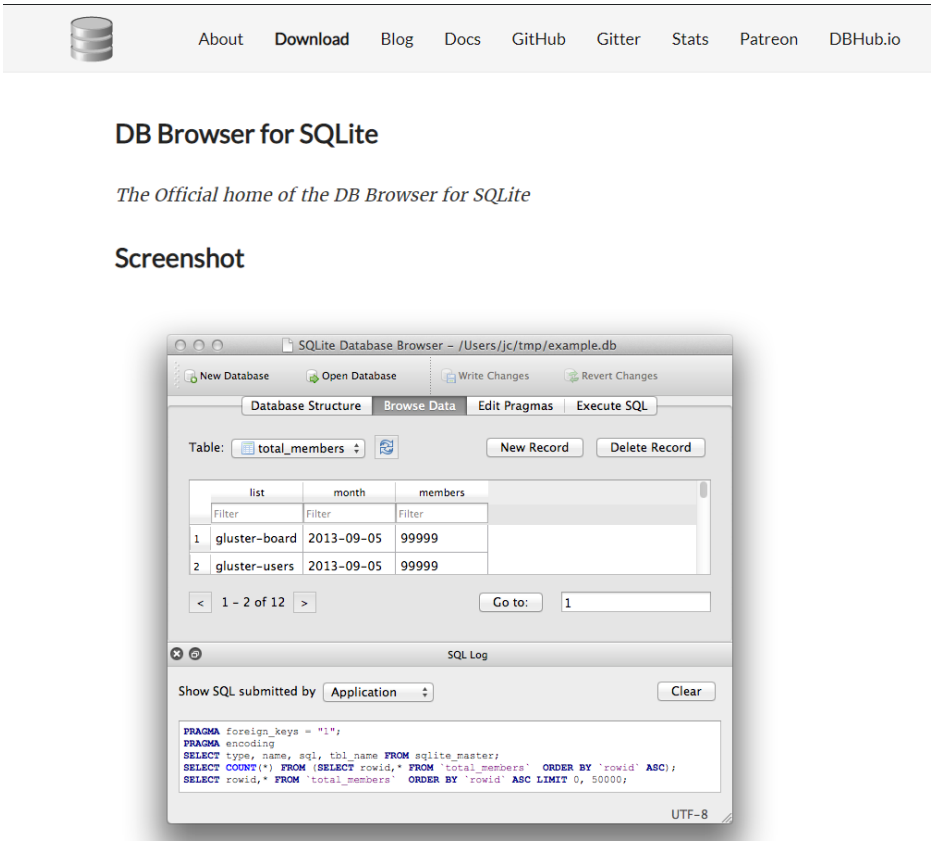


ILUSTRACIÓN 70. PÁGINA WEB DE DB BROWSER PARA SQLITE

Para descargarlo, se debe de desplazar la página hacia abajo y dar clic en la última versión:

ILUSTRACIÓN 71. ÚLTIMA VERSIÓN DISPONIBLE DEL PROGRAMA

Y seleccionar la opción de descarga adecuada para la versión de nuestro sistema operativo.

Version 3.12.1 released

3 min read

2020-11-09

This is the first bug fix release for our 3.12.x series.

There aren't any "super critical **must upgrade**" bugs fixed, so updating isn't urgent. 😉

Downloads

- DB.Browser.for.SQLite-3.12.1-win32-v2.msi - Standard (MSI) installer for Win32 and WinXP
- DB.Browser.for.SQLite-3.12.1-win32.zip - .zip (no installer) for Win32 and WinXP
- DB.Browser.for.SQLite-3.12.1-win64-v2.msi - Standard (MSI) installer for Win64
- DB.Browser.for.SQLite-3.12.1-win64.zip - .zip (no installer) for Win64
- DB.Browser.for.SQLite-3.12.1-v2.dmg - For macOS

The changes in this over the 3.12.0 release include:

ILUSTRACIÓN 72. ÁREA DE DESCARGAS E INFORMACIÓN SOBRE ÚLTIMA VERSIÓN

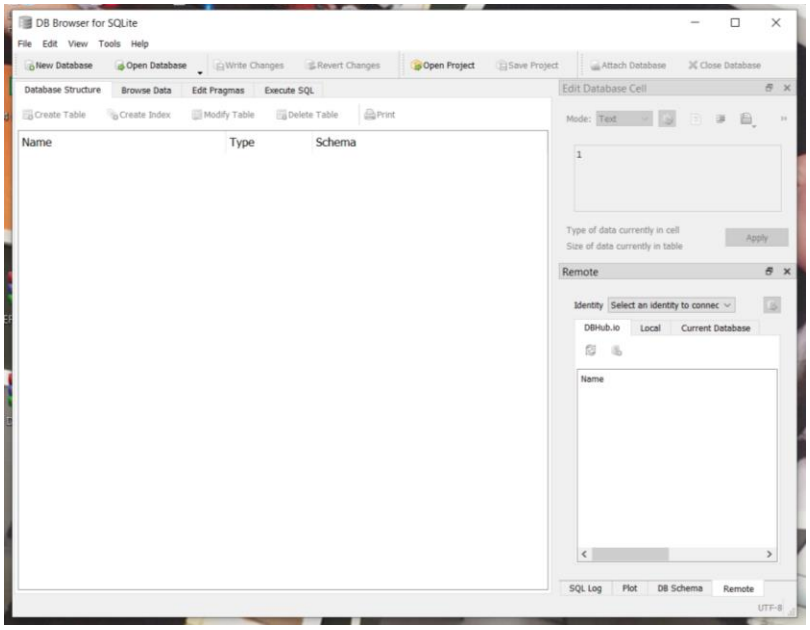


ILUSTRACIÓN 73. INTERFAZ DE PROGRAMA DB BROWSER PARA SQLITE

Para crear una base de datos

Traduciendo los pasos de creación de una base de datos a código, se apreciaría como lo siguiente:

1. Abrir-crear conexión

```
import sqlite3

miConexion = sqlite3.connect("PrimeraBBDD")
```

2. Crear un puntero y ejecutar el query

```
miCursor = miConexion.cursor()

miCursor.execute( "CREATE TABLE TABLA (NOMBRE_ARTICULO
VARCHAR(50), CLASIFICACION VARCHAR(50), USO VARCHAR(100))"
)
```

3. Manejar los resultados CRUD (Create-Read-Update-Delete)

   Para eso, deberá de ejecutarse el query con la instrucción correspondiente: `Update`, `delete`, `insert` o `select`. Esto puede realizarse con ayuda de DB Browser o con el programa mismo.

4. Adicional a cada cambio, siempre se debe incluir la instrucción:

```
miConexion.commit()
```

5. Se cerrará el puntero y finalizará la conexión con la BBDD:

```
miConexion.close()
```

Una base de datos creada con Python en SublimeText puede apreciarse de la siguiente manera:

```python
#Automatización de claves

import sqlite3

miConexion = sqlite3.connect("INCOMPATIBLES")

miCursor = miConexion.cursor()

#generación automática de claves
miCursor.execute('''
    CREATE TABLE INCOMPATIBLES_3 (
    ID INTEGER PRIMARY KEY AUTOINCREMENT,
    FARMACO VARCHAR (50),
    CLASIFICACION VARCHAR (100),
    INCOMPATIBILIDADES_FF VARCHAR (100),
    INCOMPATIBILIDADES_FE VARCHAR(150))
''')

incompatibilidades_A = [
    ("APAP", "AINE", "Ivermectina, ibuprofeno, insulina", "CMC"),
    ("Alopurinol", "Antiarritmico", "Fentanilo, Ivermectina, Antiarritmicos", "CMC")

]
#NULL se usa para gestionar automaticamente el campo clave
miCursor.executemany("INSERT INTO INCOMPATIBLES_3 VALUES (NULL,?,?,?,?)", incompatibilidades_A)

miConexion.commit()

miConexion.close()
```

ILUSTRACIÓN 74. ESTRUCTURA PROPUESTA PARA BASE DE DATOS

Siguiendo los pasos mencionados anteriormente se puede elaborar una base de datos sencilla (personalizable) pero funcional. Primero se importa el paquete de SQLite, se abre la conexión y se crea el puntero. En el ejemplo de arriba se nombra a la base de datos como “INCOMPATIBLES” (el nombre es de elección libre) al cual, se le puede añadir la extensión “.db” para poder ejecutarla y abrirla automáticamente con DB Browser. Al ejecutar el cursor se creará la estructura de la base de datos, la cual posee un nombre “INCOMPATIBLES_3”, un generador automático de claves ascendentes ID (donde se especifica el tipo de variable integer), posteriormente las columnas creadas por nosotros: FARMACO, CLASIFICACION, INCOMPATIBILIDADES_FF, INCOMPATIBILIDADES_FE y podrían haber muchas más y ser de cualquier tema que sea de la elección del usuario; hay que especificar el tipo de variable que se incluirá dentro de cada columna y la longitud de caracteres o información que esta admitirá. Es importante cuidar la estructura de la tabla, ya que, de no crearse la tabla al ejecutar el código, no se podrá ejecutar nada más. Dentro del ejemplo también se muestra la creación de una lista con múltiples datos a insertar en la base de datos, siendo una forma de hacerlo el indicar al cursor el executar varios elementos con el método

`executemany` indicando el nombre de la tabla donde insertar, los valores a tomar automatizados (`NULL,?,?,?,?`) incluyendo un signo de interrogación por cada dato adicional a la clave autogenerada (`NULL`) y el objeto de donde se tomarán los valores a insertar (la lista). El formato en mayúsculas no es obligatorio y existen diversas maneras de crear y manipular tablas, la elección será libre y se puede consultar más información en la documentación oficial de `sqlite3` de Python

Para las operaciones CRUD, una opción al programar la base de datos pudiera ser las siguientes: BÚSQUEDA/LECTURA O READ

```
#BUSQUEDA O READ:
import sqlite3

miConexion = sqlite3.connect("INCOMPATIBLES_3") #aqui debe ir la tabla en la que buscamos

miCursor = miConexion.cursor()

miCursor.execute("SELECT * FROM INCOMPATIBLES_3 WHERE FARMACO = 'APAP'") #aqui va el criterio que buscamos

busqueda = miCursor.fetchall()

print(busqueda)




miConexion.commit()

miConexion.close()

#Ejecutamos y se observa lo que coincide
```

ILUSTRACIÓN 75. EJEMPLO DE BÚSQUEDA DE DATOS CON SQLITE EN BASE DE DATOS

UPDATE O ACTUALIZAR

```
#UPDATE:

import sqlite3

miConexion = sqlite3.connect("INCOMPATIBLES_3")

miCursor = miConexion.cursor()

miCursor.execute("UPDATE INCOMPATIBLES_3 SET CLASIFICACION ='ANTIINFLAMATORIO' WHERE FARMACO = 'APAP'")

#Hay que especificar toda la ruta y lo que se desea cambiar


miConexion.commit()

miConexion.close()
```

ILUSTRACIÓN 76. EJEMPLO DE ACTUALIZACIÓN DE DATOS CON SQLITE EN BASE DE DATOS

DELETE O ELIMINAR

```
#DELETE:

import sqlite3

miConexion = sqlite3.connect("INCOMPATIBLES_3")

miCursor = miConexion.cursor()

miCursor.execute("DELETE FROM INCOMPATIBLES_3 WHERE ID = 1")


miConexion.commit()

miConexion.close()
```

ILUSTRACIÓN 77. EJEMPLO DE ELIMINACIÓN DE DATOS CON SQLITE EN BASE DE DATOS

Capítulo XVIII. Ejecutables

Existen muchas maneras de crear ejecutables con ayuda de Python, sin embargo, se pueden obtener resultados satisfactorios con una de las maneras más simples de hacerlo.

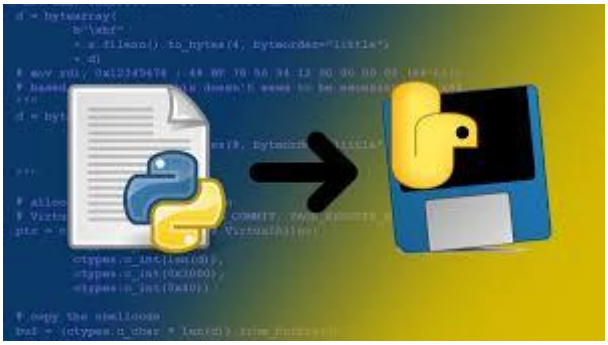


ILUSTRACIÓN 78. CREACIÓN DE ARCHIVOS EJECUTABLES: ÍCONO DE EJECUTABLE PYTHON

El crear ejecutables con Pyinstaller resulta muy sencillo y rápido. Puede realizarse en tan solo unos cuantos pasos:

1. Se necesita poseer la estructura del futuro programa ejecutable en un archivo en formato *.py
2. Abrir el símbolo del sistema (cmd) desde la barra de búsqueda Windows. Introducir el siguiente comando y presionar la tecla enter: `pip install pyinstaller`
3. En caso de ser necesario, actualizarlo tal como indica la interfaz.
4. Se deberá de buscar el archivo que se busca convertir empleando la ruta que le corresponda, ubicándonos en la carpeta que lo contiene.
5. Una vez hallado el archivo, se debe anteponer el comando `pyinstaller` seguido del nombre del archivo con su debida extensión, como a continuación:

```
C:\Users\MaJii\OneDrive\Escritorio\Tesis_PC\Python\Graficos
>pyinstaller Calculadora.py
```

6. Para encontrar el ejecutable se deberá revisar la carpeta que contiene el archivo Python que acaba de ser convertido, ubicar la carpeta dist y buscar dentro el ejecutable:

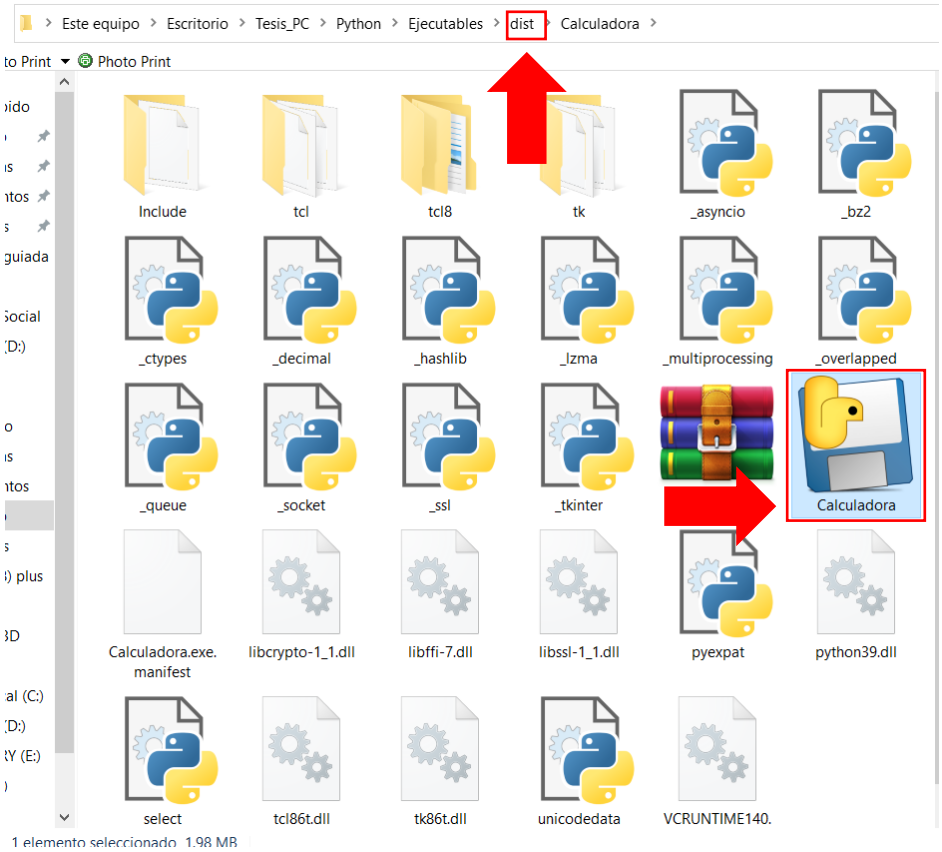


ILUSTRACIÓN 79. UBICACIÓN DE ARCHIVO EJECUTABLE EN CARPETA DIST

El resultado final es variable y en función al programa que se haya diseñado. Sin embargo, lo que se puede observar suele no resultar estético, ya que, posee una ventana adicional negra que se observa descuadrada. También puede resultar tedioso o desagradable tener que realizar una búsqueda entre tantas carpetas para llegar al ejecutable, incluso puede personalizarse el ícono que aparecerá, así como en la mayoría de los programas que existen. Para poder personalizar, mejorar la apariencia y facilidad de acceso a nuestro programa, existen diversos comandos sencillos que pueden emplearse:

a) Todo en un solo lugar:

--onefile

b) Ícono:

--icon=./nombre.ico

Nota: el ícono deberá estar dentro de la misma carpeta que el archivo a convertir para simplificar la ruta. Muchas veces los errores de ejecución de los programas se deben a la adición de un ícono, ya sea dentro y/o fuera del programa.

c) Ajustado al tamaño especificado (modo ventana):

--windowed

Se pueden emplear los 3 al mismo tiempo para obtener un ejecutable más agradable al usuario final:

```
C:\Users\MaJii\OneDrive\Escritorio\Tesis_PC\Python\Graficos>pyinstaller --windowed -- onefile --icon=./nombre.ico Calculadora.py
```

Hay que tener en cuenta que si se presentan errores al momento de ejecutar, puede ser debido a alguna línea dentro del código, alguna falta de paquetes, incompatibilidades de librerías incluidas en el programa (consultar documentación correspondiente, en este caso: `tkinter` y `pyinstaller`) o cosas tan simples como el ícono. Para resolverlo, se pueden hacer pruebas previas del programa (como se hace en la práctica de documentación), eliminar los íconos, intentar la conversión de archivos con otro paquete o consultar en la web.

Capítulo XIX. Desarrollo de un software auxiliar en el área de desarrollo de medicamentos humanos y veterinarios.

Para culminar este manual, se desarrollará una aplicación sencilla, pero útil. La aplicación estará dirigida al área industrial y/o de investigación y desarrollo de la rama farmacéutica y más específicamente: a la etapa de preformulación de medicamentos innovadores o genéricos de uso humano o veterinario.

La preformulación consiste en la caracterización fisicoquímica del principio activo y de los excipientes la cual se realiza primero por revisión bibliográfica y, posteriormente, pueden realizarse pruebas específicas marcadas en bibliografía tal como las farmacopeas internacionales y pruebas directas dentro de una formulación para conocer el comportamiento real. La obtención de información acerca de estos activos o excipientes en ocasiones es complicada, debido a la dispersión de esta; no siempre se encuentra fácilmente disponible (ya sea en la web o en lugares físicos como las bibliotecas universitarias) debido a sus altos costos y, la información disponible, muchas veces puede ser insuficiente. Con el desarrollo de un software que logre agrupar la mayor cantidad de información y se presente de manera amigable y gratuita a los usuarios, se estaría buscando tener un impacto positivo en el sector farmacéutico desde niveles de enseñanza hasta la industria e investigación.

Código:

```
"""
Creado en 2020-2021

Autora: María José Fernández Hernández

Universidad Nacional Autónoma de México
Facultad de Estudios Superiores Cuaitlán
Licenciatura en Farmacia

"""

from tkinter import *
from tkinter import messagebox
import sqlite3
```

```
#----------------------------Funcionalidades-------------------------------------#
#-------------------------------------------------------Menú base de datos
# Creación de base de datos

def conexionBBDD():

    miConexion = sqlite3.connect("RegistroDeFarmacos.db")

    miCursor = miConexion.cursor()

    try:

        miCursor.execute ('''
            CREATE TABLE DATOS_FARMACOS (
            ID INTEGER PRIMARY KEY AUTOINCREMENT,
            NOMBRE_FÁRMACO VARCHAR (50),
            NOMBRE_EXCIPIENTE VARCHAR(50),
            CLASIFICACIÓN_FARMACOLÓGICA VARCHAR (50),
            USO_HUMANO VARCHAR (100),
            USO_ANIMAL VARCHAR (100),
            FUNCIONES VARCHAR (150),
            INCOMPATIBILIDADES VARCHAR (200),
            PROPIEDADES_FQ VARCHAR (200),
            PROPORCIONES_USO VARCHAR (250))
            ''')

        messagebox.showinfo("BBDD", "Base de datos creada con éxito")

    except:

        messagebox.showwarning("Intento fallido", "La base de datos ya ha sido creada")


# Salir de la aplicación

def salirAplicacion():

    pregunta = messagebox.askquestion("Salir", "¿Deseas salir de la aplicación?")

    if pregunta == "yes":
        root.destroy()

#-------------------------------------------------------Menú borrar
#Borrar campos

def borrarCampos():

    miID.set("")
```

```
    miNombre.set("")
    miExc.set("")
    miClas.set("")
    cuadroHum.delete(1.0, END)
    cuadroAnim.delete(1.0, END)
    cuadroFun.delete(1.0, END) #para indicar qué caracteres borrar
    cuadroFF.delete(1.0, END)
    cuadroFE.delete(1.0, END)
    cuadroProp.delete(1.0, END)

    messagebox.showinfo("Limpieza de campos","Campos borrados, realice la siguiente
acción.")

#-----------------------------------------------------Menú CRUD
#CREAR

def crear():
    try:
        miConexion = sqlite3.connect("RegistroDeFarmacos.db")

        miCursor = miConexion.cursor()

        registro = (cuadroNombre.get(), cuadroExc.get(), cuadroClas.get(),
cuadroHum.get(1.0, END), cuadroAnim.get(1.0, END), cuadroFun.get(1.0, END),
cuadroFF.get(1.0, END), cuadroFE.get(1.0, END), cuadroProp.get(1.0, END))

        print(registro)

        miCursor.execute("INSERT INTO DATOS_FARMACOS VALUES (NULL,?,?,?,?,?,?,?,?,?)",
registro)

        miConexion.commit()

        messagebox.showinfo("BBDD", "Registro realizado exitosamente")

        miConexion.close()
    except Exception as e:
        print(e)

#LEER/READ
def leer():
    miConexion = sqlite3.connect("RegistroDeFarmacos.db")

    miCursor = miConexion.cursor()

    miCursor.execute("SELECT * FROM DATOS_FARMACOS WHERE ID = " + miID.get() )

    elFarmaco = miCursor.fetchall()
```

```
    for Farmaco in elFarmaco:

        miID.set(Farmaco[0])
        miNombre.set(Farmaco[1])
        miExc.set(Farmaco[2])
        miClas.set(Farmaco[3])
        cuadroHum.delete(1.0, END)
        cuadroHum.insert(1.0, Farmaco[4])
        cuadroAnim.delete(1.0, END)
        cuadroAnim.insert(1.0, Farmaco[5])
        cuadroFun.delete(1.0, END)
        cuadroFun.insert(1.0, Farmaco[6])
        cuadroFF.delete(1.0, END)
        cuadroFF.insert(1.0, Farmaco[7])
        cuadroFE.delete(1.0, END)
        cuadroFE.insert(1.0, Farmaco[8])
        cuadroProp.delete(1.0, END)
        cuadroProp.insert(1.0, Farmaco[9])

    miConexion.commit()

#ACTUALIZAR/UPDATE
def actualizar():
    miConexion = sqlite3.connect("RegistroDeFarmacos.db")

    miCursor = miConexion.cursor()

    miCursor.execute("UPDATE DATOS_FARMACOS SET NOMBRE_FÁRMACO = '" + miNombre.get() +
        "', NOMBRE_EXCIPIENTE = '" + miExc.get() +
        "', CLASIFICACIÓN_FARMACOLÓGICA = '" + miClas.get() +
        "', USO_HUMANO = '" + cuadroHum.get(1.0, END) +
        "', USO_ANIMAL = '" + cuadroAnim.get(1.0, END) +
        "', FUNCIONES = '" + cuadroFun.get(1.0, END) +
        "', INCOMPATIBILIDADES = '" + cuadroFF.get(1.0, END) +
        "', PROPIEDADES_FQ = '" + cuadroFE.get(1.0, END) +
        "', PROPORCIONES_USO = '" + cuadroProp.get(1.0, END) +
        "' WHERE ID = " + miID.get())


    miConexion.commit()

    messagebox.showinfo("BBDD", "Actualizado correctamente")


#BORRAR/DELETE
def eliminar():
    miConexion = sqlite3.connect("RegistroDeFarmacos.db")
```

```
    miCursor = miConexion.cursor()

    miCursor.execute("DELETE FROM DATOS_FARMACOS WHERE ID = " + miID.get())

    miConexion.commit()

    messagebox.showinfo("BBDD", "Registro borrado correctamente")

#------------------------------Menú ayuda-----------------------------------------#
def contacto():
    messagebox.showinfo("Dudas, preguntas y comentarios", "Mandar correo a
majofarma@comunidad.unam.mx")

def licencia():
    messagebox.showinfo("Licencia", "Este programa es de licencia libre")

def AcercaDe():
    messagebox.showinfo("Acerca de...", "Año 2020-2021 \n \n Versión 1.0.0")

#------------------------ Personalización de ventana----------------------------------#
root = Tk()
root.title("Inserción de datos sobre fármacos: búsqueda y actualización")
root.config(bg = "dark turquoise", bd = 15, relief = "sunken", cursor = "hand2")

#--------------------------- Menú de opciones-------------------------------------#
barraMenu = Menu(root)
root.config(menu = barraMenu, width = 800, height = 800)

#-------Menú de bases de datos
bbddMenu = Menu(barraMenu, tearoff = 0)
bbddMenu.add_command(label = "Conectar", command = conexionBBDD)
bbddMenu.add_command(label = "Salir", command = salirAplicacion)

#-----Menú borrar
borrarMenu = Menu(barraMenu, tearoff = 0)
borrarMenu.add_command(label = "Borrar campos", command = borrarCampos)

#-----Menú CRUD
crudMenu = Menu(barraMenu, tearoff = 0)
crudMenu.add_command(label = "Crear", command = crear)
crudMenu.add_command(label = "Leer", command = leer)
crudMenu.add_command(label = "Actualizar", command = actualizar)
crudMenu.add_command(label = "Borrar", command = eliminar)

#-----Menú ayuda
helpMenu = Menu(barraMenu, tearoff = 0)
helpMenu.add_command(label = "Licencia", command = licencia)
```

```
helpMenu.add_command(label = "Acerca de...", command = AcercaDe)
helpMenu.add_command(label = "Contacto", command = contacto)

#----Opciones desplegables de menú
barraMenu.add_cascade(label = "BBDD", menu = bbddMenu)
barraMenu.add_cascade(label = "Borrar", menu = borrarMenu)
barraMenu.add_cascade(label = "CRUD", menu = crudMenu)
barraMenu.add_cascade(label = "Ayuda", menu = helpMenu)


#-------------------------Creación del primer Frame ------------------------------#

miFrame = Frame(root)
miFrame.pack()

#Para poder modificar los datos de nuestras bases de datos, se asigna a cada entry una
función de variable tipo string:
miID = StringVar()
miNombre = StringVar()
miExc = StringVar()
miClas = StringVar()
#miHum
#miAnim
#miFun
#miFF
#miFE
#miProp


#A las funciones que se consideren texto no es necesario asignar estas variables

cuadroID = Entry(miFrame, textvariable = miID)
cuadroID.grid(row = 0, column = 1, padx = 10, pady = 10)
cuadroID.config(font = ("Calibri", 11))

cuadroNombre = Entry (miFrame, textvariable = miNombre)
cuadroNombre.grid(row = 1, column = 1, padx = 10, pady = 10)
cuadroNombre.config(fg = "OrangeRed3", justify = "center", font = ("Calibri", 11))

cuadroExc = Entry(miFrame, textvariable = miExc)
cuadroExc.grid(row = 2, column =1, padx =10, pady = 10)
cuadroExc.config(fg = "OrangeRed3", justify = "center", font = ("Calibri", 11))

cuadroClas = Entry(miFrame, textvariable = miClas)
cuadroClas.grid(row = 3, column = 1, padx = 10 , pady = 10)
cuadroClas.config(font = ("Calibri", 10))

cuadroHum = Text(miFrame, width = 40, height = 4)
```

```
cuadroHum.grid(row = 4, column = 1, padx = 10, pady = 10)
cuadroHum.config(font = ("Calibri", 10))

cuadroAnim = Text(miFrame, width = 40, height = 4)
cuadroAnim.grid(row = 5, column = 1, padx = 10, pady = 10)
cuadroAnim.config(font = ("Calibri", 10))

cuadroFun = Text(miFrame, width = 40, height = 4)
cuadroFun.grid(row = 6, column = 1, padx = 10, pady = 10)
cuadroFun.config(font = ("Calibri", 10))

cuadroFF = Text(miFrame, width = 40, height = 4)
cuadroFF.grid(row = 7, column = 1, padx = 1, pady = 10)
cuadroFF.config(font = ("Calibri", 10))

cuadroFE = Text(miFrame, width = 40, height = 4)
cuadroFE.grid(row = 8, column = 1, padx = 10, pady = 10)
cuadroFE.config(font = ("Calibri", 10))

cuadroProp = Text(miFrame, width = 40, height = 4)
cuadroProp.grid(row = 9, column = 1, padx = 10, pady = 10)
cuadroProp.config(font = ("Calibri", 10))

#---------------scrolls
#Uso Humano

scrollHum = Scrollbar(miFrame, command = cuadroHum.yview)
scrollHum.grid(row = 4, column = 2, sticky = "nsew")

cuadroHum.config(yscrollcommand = scrollHum.set)

#Uso Vet

scrollAnim = Scrollbar(miFrame, command = cuadroAnim.yview)
scrollAnim.grid(row = 5, column = 2, sticky = "nsew")

cuadroAnim.config(yscrollcommand = scrollAnim.set)

#Funciones

scrollFun = Scrollbar(miFrame, command = cuadroFun.yview)
scrollFun.grid(row = 6, column = 2, sticky = "nsew")

cuadroFun.config(yscrollcommand = scrollFun.set)

#FF/EE

scrollFF = Scrollbar(miFrame, command = cuadroFF.yview)
```

```
scrollFF.grid(row = 7, column = 2, sticky = "nsew")

cuadroFun.config(yscrollcommand = scrollFun.set)

#FE

scrollFE = Scrollbar(miFrame, command = cuadroFE.yview)
scrollFE.grid(row = 8, column = 2, sticky = "nsew")

cuadroFE.config(yscrollcommand = scrollFE.set)

#Proporciones

scrollProp = Scrollbar(miFrame, command = cuadroFun.yview)
scrollProp.grid(row = 9, column = 2, sticky = "nsew")

cuadroProp.config(yscrollcommand = scrollProp.set)


#-----------------------Creación de Labels--------------------------------#
IDLabel = Label(miFrame, text = "ID de producto: ")
IDLabel.grid(row = 0, column = 0, sticky = "e", padx = 10, pady = 10)

NombreLabel = Label(miFrame, text = "Nombre de fármaco:")
NombreLabel.grid(row = 1, column = 0, sticky = "e", padx = 10, pady = 10)

ExcLabel = Label(miFrame, text = "Nombre del excipiente: ")
ExcLabel.grid(row = 2, column = 0, sticky = "e", padx = 10, pady = 10)

ClasLabel = Label(miFrame, text = "Clasificación:")
ClasLabel.grid(row = 3, column = 0, sticky = "e", padx = 10, pady = 10)

HumLabel = Label(miFrame, text = "Uso en humanos:")
HumLabel.grid(row = 4, column = 0, sticky = "e", padx = 10, pady = 10)

AnimLabel = Label(miFrame, text = "Uso en animales:")
AnimLabel.grid(row = 5, column = 0, sticky = "e", padx = 10, pady = 10)

FunLabel = Label(miFrame, text = "Funciones:")
FunLabel.grid(row = 6, column = 0, sticky = "e", padx = 10, pady = 10)

FFLabel = Label(miFrame, text = "Incompatibilidades: ")
FFLabel.grid(row = 7, column = 0, sticky = "e", padx = 10, pady = 10)

FELabel = Label(miFrame, text = "Datos fisicoquímicos: ")
FELabel.grid(row = 8, column = 0, sticky = "e", padx = 10, pady = 10)

PropLabel = Label(miFrame, text = "Proporciones de uso:")
```

```
PropLabel.grid(row = 9, column = 0, sticky = "e", padx = 10, pady = 10)


#----------------------Botones (Frame 2)----------------------------------------#
miFrame2 = Frame (root)
#miFrame2 = Frame(root, width=18000, height=18000)
miFrame2.pack()

botonC = Button(miFrame2, text = "Crear", command = crear)
botonC.grid(row = 1, column = 0, sticky = "e", padx = 10, pady = 10)

botonR = Button(miFrame2, text = "Leer", command = leer)
botonR.grid(row = 1, column = 1, sticky = "e", padx = 10, pady = 10)

botonU = Button(miFrame2, text = "Actualizar", command = actualizar)
botonU.grid(row = 1, column = 2, sticky = "e", padx = 10, pady = 10)

botonD = Button(miFrame2, text = "Borrar", command = eliminar)
botonD.grid(row = 1, column = 3, sticky = "e", padx = 10, pady = 10)


#---------------------Ejecución de programa ----------------------------------#
root.mainloop()
```

Programa en ejecución:

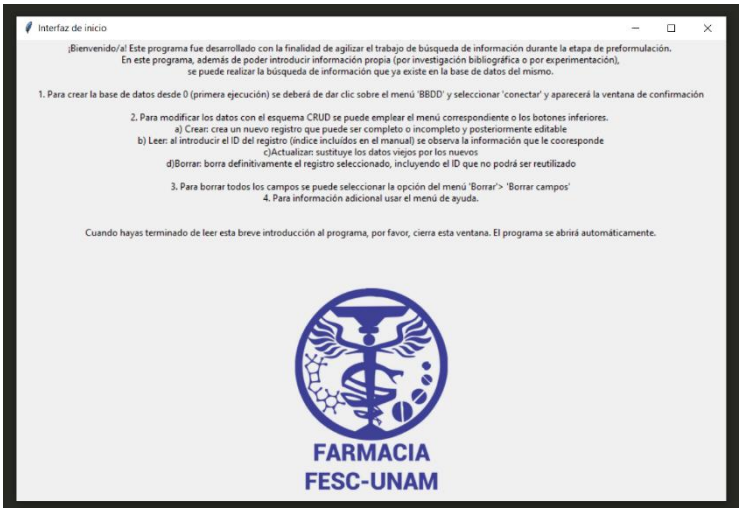


ILUSTRACIÓN 80. INTERFAZ DE BIENVENIDA

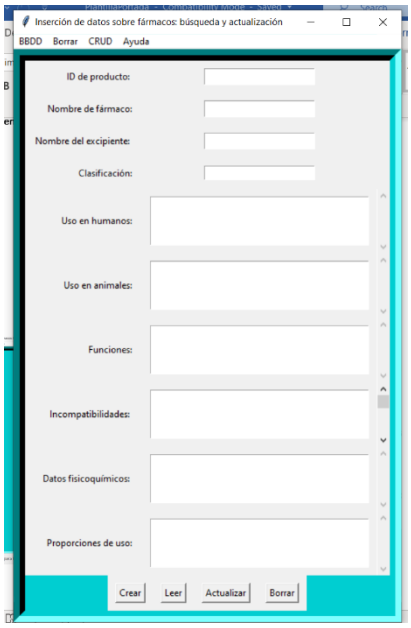


ILUSTRACIÓN 81. ÁREA DE MANIPULACIÓN Y BÚSQUEDA DE INFORMACIÓN

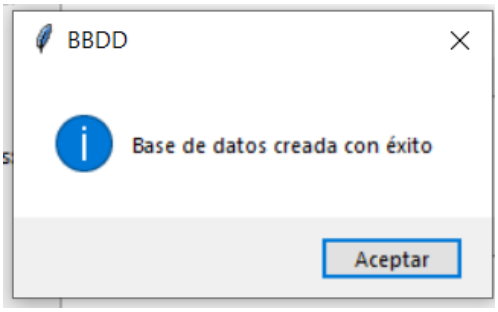


ILUSTRACIÓN 82. MENSAJES EMERGENTES INCLUIDOS EN EL PROGRAMA

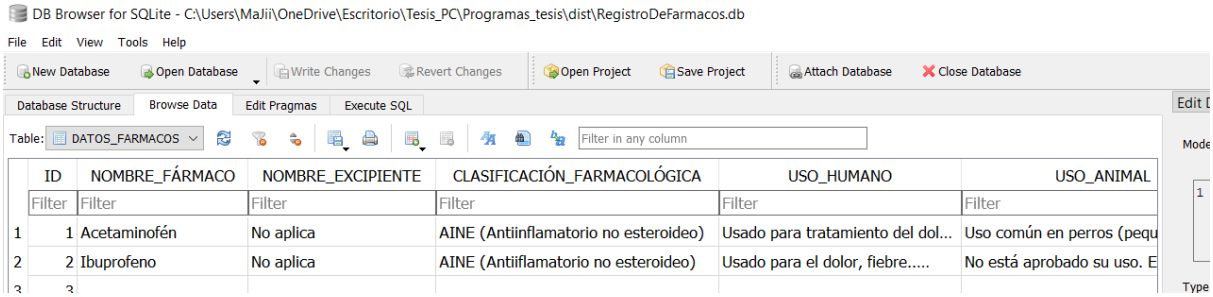


ILUSTRACIÓN 83. EJEMPLO DE REGISTRO EN BASE DE DATOS

El desarrollo del programa se lleva a cabo en poco menos de 400 líneas de código, incluidas separaciones por bloques y comentarios, además de ser convertido a un ejecutable para mejorar su manejo. La interfaz es sencilla, pero completa. Este programa puede ampliarse tanto en características mostradas como en el número de compuestos con los que se trabaja. Sin embargo, es una muestra de la aplicación de los conocimientos básicos de programación en Python y la utilidad que estos pueden llegar a tener en el sector farmacéutico y en cualquier otro que se deseen implementar dichos conocimientos.

Glosario

1. ADN/DNA(Ácido desoxirribonucleico: molécula de doble hélice formada por dos hebras entrelazadas unidas por enlaces débiles entre pares de bases nucleotídicas. Molécula que tiene codificada la información genética.
2. Algoritmos: es una forma ordenada de describir los pasos para resolver problemas. Es una manera abstracta de reducir un problema a un conjunto de pasos que le den solución.
3. API (Active Pharmaceutical Ingredient): es el ingrediente de un medicamento farmacéutico o pesticida que es biológicamente activo.
4. Archivo binario: Un file object capaz de leer y escribir objetos tipo binarios.
5. Archivo HTML(HyperText Markup Language): es el estándar para la creación de sitios web. Los navegadores web comprenden este lenguaje y pueden interpretar su codificación en diferentes textos, colores, formatos (cabeceras, párrafos, citas y otras semánticas) e hiperenlaces, así como insertar imágenes y audio mediante la incorporación de URL
6. Argumento: Un valor pasado a una función (o método) cuando se llama a la función.
7. Atributo: un valor asociado a un objeto que es referencias por el nombre usado expresiones de punto. Por ejemplo, si un objeto o tiene un atributo a sería referenciado como o.a.
8. Base nucleotídica: son las unidades y productos químicos que se unen para formar los ácidos nucleicos, principalmente ARN y ADN.
9. Base de datos:
10. Big data: es un término que hace referencia a conjuntos de datos tan grandes y complejos que precisan de aplicaciones informáticas no tradicionales de procesamiento de datos para tratarlos adecuadamente.
11. Booleano: el tipo de dato lógico o booleano en programación que puede representar valores de lógica binaria, esto es 2 valores, que normalmente representan falso o verdadero.
12. Bucle: un bucle o ciclo, en programación, es una secuencia de instrucciones de código que se ejecuta repetidas veces, hasta que la condición asignada a dicho bucle deja de cumplirse.
13. Clase: una plantilla para crear objetos definidos por el usuario.
14. Codón: unidad del código genético formada por tres bases de nucleótidos en una molécula de ADN o ARN que especifica la síntesis de un aminoácido específico.
15. Diccionario: un arreglo asociativo, con claves arbitrarias que son asociadas a valores.
16. Ejecutable, programa:
17. FASTA, formato: es un archivo de texto que contiene secuencias de nucleótidos o aminoácidos.

18. FASTQ: ha surgido como un formato de archivo común para compartir secuenciación de datos leídos combinando tanto la secuencia como una puntuación de calidad por base asociada, a pesar de carecer de una definición formal hasta la fecha y existir en al menos tres variantes incompatibles.
19. Filogenómica: consiste en el estudio del genoma desde una perspectiva evolutiva con el objetivo de entender los mecanismos de cambio que operan sobre el genoma, así como su interrelación con la evolución de los organismos y sus características fenotípicas.
20. Función: una serie de sentencias que retornan un valor al que las llama.
21. Gap: separaciones añadidas artificialmente a una secuencia para maximizar su alineamiento con otras. Pueden darse por inserción o deleción y ser resultado de mutaciones.
22. Gen: unidad física y funcional de la herencia. Secuencia de nucleótidos ubicados en una posición determinada dentro del genoma que codifica un producto funcional específico (proteína o molécula de ARN).
23. Genoma: el conjunto completo de ADN (material genético) en un organismo.
24. Genómica: conjunto de disciplinas relacionadas con el estudio de los genomas y sus aplicaciones en terapia génica, biotecnología, etc.
25. Isla CpG: son regiones de ADN y conforman aproximadamente un 40% de promotores de los genes de mamíferos. Son regiones donde existe una gran concentración de pares de citosina y guanina enlazados por fosfatos. La "p" en CpG representa que están enlazados por un fosfato.
26. Iteración (ciclo o bucle): ejecución de una sentencia o conjunto de sentencias, mientras una variable booleana sea verdadera.
27. Iterador: un objeto que representa un flujo de datos.
28. Lista: es una sequence Python incorporada.
29. Máquina Virtual: una computadora definida enteramente por software. En informática, una máquina virtual es un software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora real. Este software en un principio fue definido como "un duplicado eficiente y aislado de una máquina física".
30. Medicamento:
31. Método: una función que es definida dentro del cuerpo de una clase.
32. Módulo: un objeto que sirve como unidad de organización del código Python.
33. Mutación: es cualquier cambio en la secuencia de nucleótidos del ADN.
34. Objeto: Cualquier dato con estado (atributo o valor) y comportamiento definido (métodos).
35. OPP/POO: Programación Orientada a Objetos

36. **Paquete:** Un module Python que puede contener submódulos o recursivamente, subpaquetes.
37. **Parámetro:** Una entidad nombrada en una definición de una función (o método) que especifica un argumento (o en algunos casos, varios argumentos) que la función puede aceptar.
38. **PATH, Python:** El sistema operativo tiene una variable de entorno llamada PATH, cuya función es conocer la ruta a todos los ejecutables al que el sistema o el shell pueden acceder directamente.
39. **PCR (Reacción en Cadena de la Polimerasa):** Técnica que permite la amplificaciónexponencial de un fragmento de ADN
40. **PEP:** Propuesta de mejora de Python, del inglés *Python Enhancement Proposal.* Un PEP es un documento de diseño que brinda información a la comunidad Python, o describe una nueva capacidad para Python, sus procesos o entorno. Los PEPs deberían dar una especificación técnica concisa y una fundamentación para las capacidades propuestas.
41. **RNA:** es un ácido nucleico formado por una cadena de ribonucleótidos. Esta presente tanto en las células procariotas como en las eucariotas, y es el único material genético de ciertos virus (virus ARN). El ARN celular es lineal y de hebra sencilla, pero en el genoma de algunos virus es de doble hebra.
42. **Sanger, método:** método de secuenciación de ADN que se basa en el empleo de dideoxinucleótidos que carecen del grupo hidroxilo del carbono 3', de manera que cuando uno de estos nucleótidos se incorpora a una cadena de ADN en crecimiento, esta cadena no puede continuar elongándose.
43. **Secuencia:** ejecución de una sentencia tras otra.
44. **Secuenciación:** determinación del orden de los nucleótidos en una molécula de ARN o ADN.
45. **Selección o condicional**: ejecución de una sentencia o conjunto de sentencias, según el valor de una variable booleana.
46. **Sentencia:** Una sentencia es parte de un conjunto (un bloque de código).
47. **Símbolo de sistema:**
48. **Sistema operativo:** Un sistema operativo es el software principal o conjunto de programas de un sistema informático que gestiona los recursos de hardware y provee servicios a los programas de aplicación de software, ejecutándose en modo privilegiado respecto de los restantes.
49. **String:** Son un tipo de secuencia de símbolos.
50. **Software:**
51. **Tándem, repetición:** Una repetición en tándem es una secuencia de dos o más pares de bases de ADN que se repite de tal manera que las repeticiones se encuentran uno

al lado del otro en el cromosoma. Repeticiones en tándem están generalmente asociadas con el ADN no codificante. Repeticiones en tándem se refiere a un fenómeno de la secuencia.

52. Tipo: El tipo de un objeto Python determina qué tipo de objeto es; cada objeto tiene un tipo.
53. Unicode: es un estándar de tecnología de la información (TI) para la codificación, representación y manejo consistentes de texto expresado en la mayoría de los sistemas de escritura del mundo. Sirve para poder mejorar el intercambio de información alrededor del mundo y manejo de informática mejorada.
54. Variable de clase: corresponden a una clase determinada y son accesibles sólo invocando la clase.
55. Variable de entorno: Una variable de entorno es una variable dinámica que puede afectar al comportamiento de los procesos en ejecución en un ordenador.
56. Zen de Python: Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje.

6. Análisis de resultados

La decisión de la realización de un manual interactivo enfocado a un nivel licenciatura y con ejemplos dirigidos a áreas químico-biológicas con un enfoque en áreas de posibilidad en la especialidad de farmacia es debido al constante crecimiento y avance tecnológico que se ha observado con el paso de los años en el diseño de fármacos, modelado de proteínas, análisis de secuencias genómicas, administración de almacenes u hospitales, etc. Además, se puede considerar su posterior uso como un material auxiliar dentro de carreras como la Licenciatura en Farmacia. Los diecinueve capítulos incluidos dentro del manual fueron planeados y jerarquizados de acuerdo con las necesidades de usuarios que no poseen o poseen bajos conocimientos en lenguajes de programación.

Actualmente se cuenta con más de 600 lenguajes de programación, donde algunos son muy populares desde hace algunas décadas y otros, aunque ya existían, han ganado popularidad lentamente dentro de las industrias, la investigación y el desarrollo de proyectos innovadores. Python es uno de los lenguajes que ha ido ascendiendo en su nivel de importancia en todas las áreas anteriores, siendo uno de los más valorados hoy en día. Existen lenguajes de programación de diversos niveles, los que se emplean para crear aplicaciones y programas son de alto nivel, tal como lo es Python, por lo que las posibilidades se vuelven sumamente extensas. Aunado a lo anteriormente mencionado, la elección del lenguaje Python y no otro es por su facilidad de aprendizaje, la diversidad de herramientas adicionales que se pueden utilizar o crear. Otro punto a favor es la potabilidad del lenguaje: puede emplearse en OSX, Linux, Windows e incluso se está trabajando su uso en Android para brindar la posibilidad de programar desde la comodidad del celular. Sin embargo, la mayoría de los usuarios emplea el sistema operativo Windows (Ramírez, 2019) y en muchas ocasiones, los tutoriales, manuales y libros están dirigidos a sistemas operativos diferentes que están pensados para la programación o de manejos más especializados, por lo cual, este manual fue pensado para dichos usuarios, demostrando la facilidad de uso sin que el sistema operativo sea una barrera para el aprendizaje de habilidades como la programación.

Es muy amplia el área de aplicación de los conocimientos que pueden ser adquiridos con el uso de este material didáctico, lo cual es apreciable con los ejemplos que son mostrados dentro del mismo, lo cual permite a estudiantes y profesionales de las diversas áreas, no sólo complementar los conocimientos adquiridos previamente, sino generar nuevos por medio de la creación de programas que dan solución a problemas propios del estudiante como de un área dentro de la carrera que este cursa e incluso pudiendo generar materiales que pueden ser empleados para la obtención de remuneraciones monetarias o del ahorro de capital por creación de software para las escuelas, laboratorios y universidades. La posibilidad de creación de software de utilidad queda demostrada con la creación de un software ejecutable de extensión *.exe de interfaz amigable para el usuario; el programa que concentra, almacena y maneja la información recopilada de diversos medios acerca de principios activos y excipientes con su respectiva información fundamental para formulación de medicamentos veterinarios y humanos puede ser de utilidad en materias tales como Tecnología Farmacéutica I y II, especialidad de Desarrollo Farmacéutico Veterinario, LEM Farmacia, Biofarmacia e incluso como auxiliar en manejos de inventario si se le añadiesen funciones adicionales.

7. Conclusiones

- Se llevó a cabo la recopilación de recursos electrónicos tales como secuencias genómicas, imágenes, monografías y la búsqueda de áreas de oportunidad tales como manejo de inventarios, manejo de farmacias, aplicaciones biológicas como bioinformática, aplicaciones matemáticas y planeación de formulaciones farmacéuticas.

- Se estructuró un manual y se obtuvo un formato final con diecinueve capítulos interactivos estructurados jerárquicamente sobre el aprendizaje del lenguaje de programación Python 3 con suficiente contenido para guiar al futuro usuario a la creación de software personalizado.

- Finalmente, se logró llevar a cabo la estructuración y puesta en funcionamiento de un software ejecutable de extensión *.exe de interfaz amigable al usuario que concentra, almacena y maneja la información recopilada de diversos medios acerca de principios activos y excipientes con su respectiva información fundamental para formulación de medicamentos veterinarios y humanos.

- El futuro de las áreas de investigación y desarrollo como las que competen a las áreas de las ciencias químico-biológicas y de la salud se encuentra en proyectos ejecutados vía *in silico* los cuales requieren del conocimiento de lenguajes de programación tales como Python 3.

8. Referencias

- Baranov, P. V., Henderson, C. M., Anderson, C. B., Gesteland, R. F., Atkins, J. F., & Howard, M. T. (2005). Programmed ribosomal frameshifting in decoding the SARS-CoV genome. Virology, 332(2), 498-510. https://doi.org/10.1016/j.virol.2004.11.038
- Bassi, S. (2017). Python for Bioinformatics (2.a ed.). Amsterdam University Press. http://englishonlineclub.com/pdf/Python%20for%20Bioinformatics%20(Second%20Edition)%20[EnglishOnlineClub.com].pdf
- Brea, A. (2020, 15 septiembre). Python: el lenguaje de programación más popular del mundo. Baética. https://baetica.com/python-lenguaje-programacion/?cli_action=1607567290.606
- Cañedo, R. (2004). Bioinformática: en busca de los secretos moleculares de la vida. ACIMED, 12(6), 1. http://scielo.sld.cu/scielo.php?script=sci_arttext&pid=S1024-94352004000600002&lng=es&tlng=es
- Cañedo, R., Ramos, R., Guerrero, J. (2005). La Informática, la Computación y la Ciencia de la Información: una alianza para el desarrollo. Centro Nacional de Información de las Ciencias Médicas-Infomed. Recuperado de: http://scielo.sld.cu/pdf/aci/v13n5/aci07505.pdf
- Cock, P. J. A., Antao, T., Chang, J. T., Chapman, B. A., Cox, C. J., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F., Wilczynski, B., & de Hoon, M. J. L. (2009). Biopython: freely available Python tools for computational molecular biology and bioinformatics. Bioinformatics, 25(11), 1422-1423. https://doi.org/10.1093/bioinformatics/btp163
- Del Río, L. (2002). Estudio de preformulación para el desarrollo de comprimidos de indometacina como sustancia policristalina. Ars Farmacéutica 43:1-2, (147-148). Recuperado de: https://farmacia.ugr.es/ars/pdf/231.pdf
- Ekmekci, B., McAnany, C. E., & Mura, C. (2016). An Introduction to Programming for Bioscientists: A Python-Based Primer. PLOS Computational Biology, 12(6), e1004867. https://doi.org/10.1371/journal.pcbi.1004867
- Galvez, F. (2009). Ácidos Nucleicos. Universidad Politécnica de Valencia. Valencia, España. https://www.uv.es/tunon/pdf_doc/Acidos%20Nucleicos_09.pdf
- Hunt, J. (2019). A Beginners Guide to Python 3 Programming. Springer Publishing. https://doi.org/10.1007/978-3-030-20290-3
- Hunt, J. (2019b). Advanced Guide to Python 3 Programming (Undergraduate Topics in Computer Science) (1st ed. 2019 ed.). Springer. https://doi.org/10.1007/978-3-030-25943-3
- Lancaster, A., & Webster, G. (2019). Python for the Life Sciences: A Gentle Introduction to Python for Life Scientists (English Edition) (1.a ed.). Apress. https://doi.org/10.1007/978-1-4842-4523-1
- Langtangen, H. P. (2016). A Primer on Scientific Programming with Python (Texts in Computational Science and Engineering, 6) (5th ed. 2016 ed.). Springer. https://doi.org/10.1007/978-3-662-49887-3

- Lee, K. (2019). Why we should learn Python. PharmaSUG. https://www.pharmasug.org/proceedings/2019/AD/PharmaSUG-2019-AD-136.pdf
- Lewis, H. R., & Papadimitriou, C. H. (1998). Elements of the Theory of Computation (2.a ed.). Prentice Hall. https://awa2el.net/sites/default/files/nzry_hsbt_ltb_lthny.pdf
- Lewis, H. R., & Papadimitriou, C. H. (1998). Elements of the Theory of Computation (2.a ed.). Prentice Hall. https://awa2el.net/sites/default/files/nzry_hsbt_ltb_lthny.pdf
- Mancilla, M. (2008). Predicción bioinformática e identificación de islas genómicas en *Brucella abortus*: inestabilidad genómica conduce a cambios fenotípicos. Universidad Austral de Chile. https://www.conicyt.cl/bases/catalogo/tesis/html/8/0738.html
- Mitra, A. (2020, 10 octubre). Pharmacy management using Python. Skyfi Labs. Recuperado de: https://www.skyfilabs.com/project-ideas/pharmacy-management-using-python
- Mullis, K. (1990). The unusual origin of the polymerase chain reaction. Scientific American 56-65. https://cs.brown.edu/courses/csci1810/resources/pcr%20origin.pdf
- National Center for Biotechnology Information. (2020). BLAST Topics. A. Query Input and database selection. https://blast.ncbi.nlm.nih.gov/Blast.cgi?CMD=Web&PAGE_TYPE=BlastDocs&DOC_TYPE=BlastHelp
- National Center for Biotechnology Information. (2020). Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome. https://www.ncbi.nlm.nih.gov/nuccore/1798174254
- Papich, M. G. (2015). Saunders Handbook of Veterinary Drugs: Small and Large Animal. Saunders. https://www-sciencedirect-com.pbidi.unam.mx:2443/book/9780323244855/saunders-handbook-of-veterinary-drugs#book-info
- Pinzón, A. (2007). Introducción al diseño "*in silico*" de primers. Universidad Nacional de Colombia. Bogotá, Colombia. http://bioinf.ibun.unal.edu.co/cbib/estudiantes/1-07/primerDesign.pdf
- Python Software Foundation. (2020). Documentación de Python 3.8.5: glosario. https://docs.python.org/es/3/glossary.html
- Python Success Stories. (s. f.). Python.org. https://www.python.org/about/success/astra/
- Riquelme, A., Pinto, M. (2005). Introducción a la genómica en la vid. Universidad de Chile, Facultad de Ciencias Agronómicas. Santiago, Chile. http://www.gie.uchile.cl/pdf/Manuel%20Pinto/Introducci%F3n%20a%20la%20gen%F3mica%20en%20vid.pdf?fbclid=IwAR3nHcAeCIF14D2PmTIOXYz2C4oNem0wuzmhU_4Hs0cOqayf5_jV3NtYMJs
- Robertson, M. P., Igel, H., Baertsch, R., Haussler, D., Ares, M., & Scott, W. G. (2004). The Structure of a Rigorously Conserved RNA Element within the SARS Virus Genome. PLoS Biology, 3(1), e5. https://doi.org/10.1371/journal.pbio.0030005

- Rowe, R. C., Sheskey, P. J., Quinn, M. E., & American Pharmacists Association. (2009). Handbook of Pharmaceutical Excipients (6th ed.). Amsterdam University Press. https://www.pdfdrive.com/handbook-of-pharmaceutical-excipients-d42475804.html
- Schubert, B., Walzer, M., Brachvogel, H.-P., Szolek, A., Mohr, C., & Kohlbacher, O. (2016). FRED 2: an immunoinformatics framework for Python. Bioinformatics, 32(13), 2044-2046. https://doi.org/10.1093/bioinformatics/btw113
- Williams, G. D., Chang, R.-Y., & Brian, D. A. (1999). A Phylogenetically Conserved Hairpin-Type 3′ Untranslated Region Pseudoknot Functions in Coronavirus RNA Replication. Journal of Virology, 73(10), 8349-8355. https://doi.org/10.1128/jvi.73.10.8349-8355.1999
- Wu,F., Zhao,S., Yu,B., Chen,Y.M., Wang,W., Song,Z.G., Hu,Y., Tao,Z.W., Tian,J.H., Pei,Y.Y., Yuan,M.L., Zhang,Y.L., Dai,F.H., Liu,Y., Wang,Q.M., Zheng,J.J., Xu,L., Holmes,E.C. and Zhang,Y.Z. (2020). A new coronavirus associated with human respiratory disease in China. Nature, 579 (7798), 265-269. https://www.nature.com/articles/s41586-020-2008-3
- Zapata-Ros, M. (2015). Pensamiento computacional: Una nueva alfabetización digital. Revista de Educación a Distancia (RED), 46, 2-24. https://doi.org/10.6018/red/45/4

9. Anexos

a) Índice de ilustraciones

b) Índice de tablas