Introducción a Programación con Python

Ivica Kalichanin Balich

9 de marzo de 2016

Índice general

Prefacio

Para Quien es Este Libro

Este texto tiene dos niveles introductorio y avanzado.

Las personas sin experiencia con la programación de las computadoras deben estudiar la primera parte detenidamente para adoptar los conceptos básicos y sobre todo la manera de pensar al crear un programa. Una vez que se domine la primera parte, se puede continuar al nivel avanzado, para comprender los niveles de una computadora y su interrelación y para dominar el lenguaje Python con profundidad de manera que todos los demás cursos relacionados con computación puede convertir en cursos prácticos, programando en Python todos los modelos que vaya aprendiendo.

Las personas que ya saben programar en algún lenguaje aprovecharan la parte introductoria para conocer los elementos básicos de Python y acostumbrarse a su sintaxis. Aunque sean experimentados programadores, pueden aprovechar los proyectos que se desarrollan a lo largo de libro para entender el funcionamiento de una computadora de una manera unificada. Esto no lo ofrece ningún otro curso de ciencias de computación ya que todos están fragmentados a diferentes aspectos de la computación.

Objetivos del Curso

El objetivo de curso es que sirva como el primer curso de computación y el punto unificador para todos los demás cursos o libros que pueda estudiar.

La primera parte es una introducción para principiantes en programación
que desean aprender Python como el primer lenguaje.
A los programadores se les recomienda dar una lectura rápida de esta primera parte,
ya que encontrarán algunos detalles útiles sobre el lenguaje Python.

El texto se dedica solamente al lenguaje; no discute otros temas importantes:
estructuras de datos, algoritmos, ingeniería de software, bases de datos, aplicaciones y otros.

El primer objetivo es conocer todos los aspectos y la filosofía de Python para que,
el estudiante sea capaz de aplicar el lenguaje Python para proyectos personales y
continuar con el estudio independiente de la documentación oficial de lenguaje Python y
de otros libros más especializados sobre lenguaje, librerías y aplicaciones Python.

El segundo objetivo es conocer algunos paquetes más frecuentemente usados
de la librería estándar de Python: math, random, date, time, abc, sys, os, y tkinter.

El tercer objetivo es conocer tres paradigmas de programación:

- estructurada,
- orientada a objetos,

- guiada por eventos.

El cuarto objetivo es interactuar con el sistema operativo desde Python
usando librerías "sys" y "os" y
aprovechando las capacidades avanzadas del hardware y del sistema operativo.

El quinto objetivo es aprender a desarrollar interfaces gráficas usando la librería "tkinter".

Lenguaje Python

Versiones de Python

El Python 3, versión 3.0, que se desarrollo al finales de año 2008,
ya no era compatible con la versión 2.6.

Por esta razón hoy tenemos dos lenguajes Python:

- Python 2 representado por versiones 2.6 y 2.7
- Python 3 representado por versiones 3.0, 3.1, ..., 3.5, ...

El Python 2.7, contiene aquellos desarrollos hechos en Python 3
los cuales son compatibles con la versión Python 2.6.

Es posible escribir código de manera que sea ejecutable por ambas versiones: 2 y 3.
Aunque posible, esto no es necesario ya que casi todo desarrollo nuevo se hace en el Python 3.

Es posible instalar Python 2 y Python 3 en la misma máquina independientemente y
ejecutar "python2" o "python3", según la necesidad.
Usualmente uno es instalado como parte de sistema operativo y el otro se puede instalar aparte.

¿Por qué Python?

Python es lenguaje interpretado, que tiene muchas ventajas en el desarrollo de software.
El único reclamo a Python es la velocidad de ejecución de programas Python.
Definitivamente no está hecho para diseñar sistemas operativos y software embebido,
donde la velocidad de ejecución es uno de los parámetros esenciales.
Cabe decir que Python es suficientemente rápido y es usado en varios tipos de proyectos
de robótica y control automático.

A principio, los programas en Python fueron 10, 20 o más veces lentos que por ejemplo C++.
Se han hecho muchas mejoras, como librerías numéricas hechas en C/C++, y otras,
de maneara que Python hoy compite con C++, Java
y lenguajes similares en cuanto la velocidad de ejecución.

Al inicio, las principales usos de Python fueron análisis de datos y aplicaciones en Internet.
Hoy en día el Python se usa para prácticamente todos tipos de aplicaciones de software:

- Programas con interfaces gráficas en variedad de plataformas, incluyendo móviles.
- Análisis numérico y presentación gráfica de datos.
- Manejo de archivos de datos procedentes de Internet, redes y bases de datos.

- Búsqueda de internet y recopilación de datos.
- Manejo de "big-data" y datos geo-espaciales con extensas cantidades de datos.
- Análisis de texto y procesamiento de lenguajes naturales
- Manejo, control y automatización de procesos en maquinas, y redes.
- "Hacking" y seguridad de maquinas, redes, servidores y conexiones a Internet.
- Programación de juegos en 2 dimensiones y 3 dimensiones.
- Algoritmos de inteligencia computacional y aprendizaje de máquinas.
- Aplicaciones para estadística, econometría y finanzas cuantitativas.

El uso de Python para desarrollo de aplicaciones tiene las siguientes ventajas:

- Es gratuito.
- Es portátil sobre múltiples plataformas.
- Tiene sintaxis simple,
- Es muy fácil de aprender, perfecto como el primer lenguaje de programación.
- Lenguaje de alto nivel, enfocado hacia las aplicaciones.
- Es interpretado, no hay necesidad de compilar cambios.
- Apoya tres paradigmas de programación: estructurada, orientada a objetos, guiada por eventos.
- Tiene "colector de basura" y todos los elementos de un lenguaje moderno.
- Está bien integrado con los sistemas operativos para el uso de hilos, programación paralela, comunicación entre procesos, etc.
- Desarrollo de aplicaciones toma, en promedio, 5 veces menos tiempo que en C++ o Java.
- Python puede usar código de C/C++ o Java, o ser embebido en software hecho en ellos.
- Tiene librerías para desarrollo "agile" y desarrollo basado en pruebas automatizadas.
- Tiene librería estándar extensa para aplicar ingeniería de software moderna y para desarrollo de una variedad de aplicaciones.
- Existen más de 60,000 librerías independientes para aplicaciones especializadas.

Intérpretes de Python y su Instalación

Python es un lenguaje interpretado.
El interpretador original y oficial está escrito en lenguaje C y se le llama CPython.
El CPython está hecho para varias plataformas: Linux, Mac, Windows, etc.

Existen otros interpretadores de lenguaje Python, como

- el JPython, para usar Java desde Python, o Python desde Java,
- el PyPy el cual es le intérprete de Python, escrito en Python,
- el IronPython para entorno .NET,
- el QPython para Android con librería gráfica Kivy multi-plataforma

- el Stackless, Cython y otros interpretadores.

No hay que confundir CPython con Cython.
El Cython es un intérprete de Python que permite mezclar lenguajes C/C++ y Python en el mismo archivo de código.

El intérprete original, CPython, no permite mezclar códigos de C/C++ y Python, en el mismo archivo de código, pero
permite usar librerías hechas en C/C++ desde código escrito en Python
y permite que funciones de Python sean llamadas desde código escrito para C/C++.

Para instalar CPython, hay que ir a la página

```
http://www.python.org
```

buscar descargas ("downloads") y descargar la versión correspondiente a su sistema operativo y al hardware de 32 o 64 bits. El de 32 bits puede correr sobre la arquitectura de 64 bits también.

Al instalar Python, se instala el intérprete, la librería estándar de Python y
en algunos casos también se instala un entorno de uso interactivo de Python y
desarrollo de aplicaciones en Python, llamado IDLE (Integrated DeveLopment Environment).

Todos los ejemplos en este texto son probados con el intérprete CPython, versión Python 3.4, en sistemas operativos Linux y Windows.

Para cualquier aclaración pueden comunicarse a correo "ivica1009@gmail.com".

Parte I.

Introducción a la Programación

1. Introducción

Nuestra cultura y modo de vivir están fuertemente ligados a usos de las computadoras.
Computadora es cualquier dispositivo programable que pueda.
almacenara, procesar y transmitir datos.

Tenemos las computadoras controlando nuestros teléfonos, vehículos, aparatos electro-domésticos, instrumentos musicales eléctricos, aparatos para climas, nuestras casas, oficinas y plantas industriales.

Al dominar un lenguaje de programación como Python podemos crear nuestros propios programas y
no estar limitados a solamente usar los programas hechos por otros.
Para crear programas complejos, profesionales se necesita más conocimientos que un lenguaje.
Se necesita dominar cierto nivel de las matemáticas, ciencias computacionales y otras.
Al mismo tiempo, conociendo un lenguaje de programación podemos escribir nuestros propios programas para controlar nuestra computadora, automatizar tareas que hacemos con ellas, diseñar algún juego que podemos jugar en la computadora u organizar una pequeña base de datos para nosotros.

1.1. Lenguajes de Programación

1.2. El Primer Modelo de Computadora

2. Tipos de Datos y Expresiones

En este capitulo empezaremos a trabajar con el lenguaje Python.
Tienen que tenerlo instalado en su computadora.
Si su computadora no lo tiene instalado,
sigan las instrucciones dadas en el capitulo 23 en la página 210.

2.1. Tipos de Datos y Expresiones

2.1.1. Tipos de Datos

Para programar en cualquier lenguaje, incluyendo Python, usamos distintos
tipos de datos: números enteros, números decimales, textos, imágenes y otros.
Cada uno de estos tipos de datos se codifica cada para ser memorizado y procesado
en una máquina hecha de elementos que tienen solamente dos estados estables: cero y uno.

Los Tipos de Datos Incorporados Básicos

Los Tipos de datos incorporados son aquellos que son parte del intérprete de Python y
no necesitan ser importados de las librerías de Python ni de otras librerías independientes.

Empezaremos el estudio de Python conociendo sus tipos de datos.
Cada uno de esos tipos de datos lo conoceremos en detalle, uno por uno en los primeros capítulos.
El objetivo aquí es nada más dar una primera idea, un poco abstracta.

El tipo de dato **genérico**, tiene solamente un valor: "None" (ninguno), pronunciado "non".
En el código Python, el valor None es representado por el literal: **None**.
Al "tipo de dato genérico" también se le dice "tipo de dato None",
ya que el None es el valor único de este tipo.

El tipo de dato lógico ("bool", "boolean") tiene sólo dos valores: "False" y "True".
El False (falso) se le pronuncia similar a "fols" y el True (cierto) se le pronuncia similar a "tru".
A este tipo de dato nos vamos a referir como tipo "**lógico**" y
a sus valores nos referimos con literales **False** y **True**.

El tipo de dato entero ("int", "integer") consiste de los números enteros.
En Python, el tamaño de un entero no está limitado por el intérprete.
Nos vamos a referir a este tipo de dato como "**enteros**" y
a los valores de este tipo con literales como: **-3, -2, -1, 0, 1, 2, 3, 4** y otros números enteros.

El tipo de dato punto flotante ("float", "floating point"), abarca los decimales de doble precisión,
definidos por el estándar IEEE 754 conocido como "double-precision floating point standard" y
equivale a tipo "double" en otros lenguajes de programación como C/C++, C#, o Java.

Nos vamos a referir a este tipo de dato como "**flotante**" y a los valores
de tipo flotante con literales: **0.123, 16.25**, o en la notación científica como: **1.45e-2**.

Usaremos el nombre "flotante", ya que estos números tienen características particulares,
que no corresponden a ningún conjunto de números comúnmente estudiados en álgebra,
como lo son: enteros, racionales (fracciones), reales (decimales), o complejos.
El flotante es un modelo de números reales, pero limitados a cierta cantidad de dígitos binarios.

El tipo de dato números complejos ("complex", "complex numbers")
es un modelo de los números complejos con las mismas limitantes que tienen los flotantes.
Los números complejos tienen la parte real "R" y la parte imaginaria "I" en forma: R + Ij,
donde los "R" e "I" son de tipo flotante y el "j" es la constante que representa la unidad imaginaria;
la raíz cuadrada de -1. Nos vamos a referir a estos números como "**complejos**" y
sus expresiones literales son de forma: **1 + 2j**, **3.4 - 5.8j**, **-1.3e-2 + 8.5e-3j**.

El tipo cadena de caracteres ("str", "string") son secuencias finitas de caracteres,
incluyendo letras, dígitos, símbolos y caracteres especiales invisibles codificados en UTF-8.
Nos vamos a referir a este tipo de dato como "**cadena**" ya que "cadena de caracteres" es muy largo.
Cada vez que mencionamos "cadena" se entiende que quiere decir "cadena de caracteres".
Los valores de tipo cadena son literales siempre escritos entre apostrofes dobles, o sencillos,
triples dobles, o triples sencillos. Las cuatro cadenas presentadas en continuación son equivalentes:

```
"cadena", 'cadena', """cadena""", '''cadena'''.
```

Por lo pronto usaremos las primeras dos formas.
Las segundas dos se usan para la documentación y para las cadenas de múltiples líneas de texto.
Vamos a ver los usos de todas estas formas de delimitadores en los ejemplos a lo largo de este libro.

En Python no hay tipo de dato "carácter" como en algunos otros lenguajes de alto nivel.
Aunque tenemos solamente un carácter entre comillas, se trata de una cadena.

Una característica común a estos tipos de datos básicos incorporados es que son "inmutables".
Este termino "inmutables" es muy importante y lo conoceremos al hablar del concepto "objeto".

Es IMPORTANTE tomar en cuenta que el intérprete de Python
distingue entre las letras minúsculas y mayúsculas al procesar código.
Por ejemplo, el "None" no es lo mismo que el "none", o el "NONE".

Los primeros tipos de datos que conocimos se pueden organizar en tres grupos:

Ninguno: None

Escalares: lógicos, enteros, flotantes, complejos

Colección: cadena

La cadena se considera una colección porque contiene caracteres como sus elementos (ítems).
El término "escalar" quiere decir que no se trata de una colección de elementos.

Otros Tipos de Datos Incorporados

Aparte de la cadena, vamos a estudiar dos colecciones inmutables más: **tupla** "tuple" y **bytes** "bytes".

También estudiaremos las dos colecciones mutables: **lista** "list" y **arreglo** de bytes "bytearray".

Estudiaremos dos tipos de conjuntos: **conjunto** "set" y conjunto **congelado** "frozenset".

Por fin estudiaremos un mapeo: **diccionario** "dict" ("dictionary").

Estos son todos los tipos de datos incorporados que vamos a usar.
Para los propósitos de este curso, vamos a dejar como la referencia a tipos incorporados la tabla:

```
None                        # Genérico     estático  inmutable  NO-mapeable
bool, int, float, complex   # Escalares    estáticos inmutables mapeables
str, bytes, frozenset       # Colecciones  estáticos inmutables mapeables
tuple                       # Colección    estático  inmutable  AMBOS
list, bytearray, set, dict  # Colecciones  dinámicos mutables   NO-mapeables
```

No se preocupen, los vamos a ir conociendo poco a poco; primero unos después otros.
El significado de "estático", "mutable" y "mapeable" veremos en la sección 4.4 en la página 49.
Por lo pronto empezaremos con los más sencillos: genérico, los escalares y la cadena ("string").

Antes de que termine el curso, ya dominarán todos los objetos integrados y además
diseñarán y codificarán sus propios tipos de objetos y sus propias colecciones de objetos.

2.1.2. Expresiones Literales

Como se prometió, vamos a empezar el estudio de Pyton con los tipos de datos más sencillos:
genérico, escalares y la cadena, ya que estos los conocemos desde la educación primaria y secundaria.

La siguiente tabla presenta los ejemplos de las **expresiones literales validas**
de los mencionados tipos de datos ya que son nuestro punto de partida para estudiar Python.

```
# TIPO       # EXPRESIONES LITERALES

Genérico     None

Lógico       False   True

Entero       1234567890   2016   108   2   0  -3

Flotante     5.8e6   3.141592   3.   .2   1e-9   0.0   -9.8e-8   -1.23

Complejo     3.4 - 8.9j   0.0 + 0.2j   -5 + 0j   -4.29 - 1.87j

Cadena       "Hola"   'Soy un "string"'   "Soy una cadena de caracteres."
```

2.1.3. Ejercicios

2.2. Modo Interactivo

El interprete de Python se puede usar en **modo interactivo**, o
se pueden escribir archivos de texto con código Python, llamados **módulos**, para formar programas.
Este segundo modo es conocido como **programación**, o "**scripting**" en Python.

Vamos a trabajar en ambos modos, aunque el énfasis estará en la programación.
Durante la programación se usa el modo interactivo para probar ciertas partes de código.

Vamos a iniciar con el modo interactivo para aprender algunos conceptos básicos
los cuáles vamos a usar en el resto de este libro.

2.2.1. Iniciar el Modo Interactivo

Para trabajar en modo interactivo use su entorno de programación, como por ejemplo IDLE, o
si trabaja desde la consola (terminal, o línea de comandos), abra su consola e
inicie el Python 3 tecleando

```
$ python3
```

o

```
$ python
```

dependiendo como el Python 3 esté instalado en su sistema.

El "$" representa el **prompt de sistema operativo** en la consola.
En su computadora este prompt será más largo ya que aquí he borrado la primera parte de prompt
la cuál precede el símbolo "$". La parte importante es: después del prompt "$" hay que teclear

```
python3
```

El Intérprete de Python responderá con algo así:

```
$ python3
Python 3.4.3 (default, Oct 14 2015, 20:28:29)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Asegúrense que les esté respondiendo el Python 3; que no sea Python 2.6... o 2.7...

El **prompt de Python** ">>>", significa que entramos exitosamente al modo interactivo.

Existen más maneras de entrar al modo interactivo. Las aprenderemos después.

Salir de Modo Interactivo

Si desean abandonar la sesión interactiva pueden usar la función "**quit**", o la función "**exit**".
Una función se activa aplicando el **operador de ejecución ()** después de su nombre.

```
>>> quit()
$
```

o

```
>>> exit()
$
```

Esto cierra el intérprete de Python y regresa el control al sistema operativo.
Como dijimos, en su computadora, el prompt de sistema operativo "$" se va a ver diferente.

También se puede interrumpir el modo interactivo al mantener la tecla "control", [**Ctrl**], oprimida y oprimir y soltar la tecla "d": [**Ctrl-d**]. Claro, después soltamos la tecla [Ctrl] también.

En sistemas basados en "Unix", como son "Mac" y "Linux", el [**Ctrl-c**] termina un programa, mientras el [Ctrl-z] solamente suspende el programa, pero no lo termina.
En sistemas basados en "Microsoft Windows," el [**Ctrl-z**] termina un programa.
Estas combinaciones de teclas podemos necesitar cuando empezamos a programar.

Buscar Ayuda en Modo Interactivo

Para buscar ayuda o aclaración, se usa la función "**help**" (ayuda).
Cada vez que tecleemos algo a Python y deseamos que este responda,
tenemos que oprimir la tecla [**Enter**], o sea [**Return**].
Teclados en Español pueden llamar esta tecla [**Entrar**], o
puede tener flecha hacia abajo y a la izquierda.

Los ejemplos del uso de "help":

```
>>> help(None)
>>> help(False)
>>> help(True)
>>> help(int)
>>> help(float)
>>> help(str)
```

Con estos obtiene ayuda sobre el término que está en los paréntesis.

Para deslizarse hacia abajo en la pantalla de la ayuda que se abrirá,
opriman la tecla [**Entrar**]. Igual es posible navegar usando las teclas con flechas.

Para salir de la pantalla de ayuda y regresar al intérprete de Python, se oprime la tecla [**q**], "quit".

Si desean obtener ayuda sobre algún operador que está entre las palabras reservadas,
mostradas en la sección 22.1 en la página 204, tienen que poner el operador entre comillas:

```
>>> help("def")
```

Las ***palabras reservadas*** son parte de Python y son usadas para propósitos específicos.
No debemos usarlas para ningún otro propósito, excepto el especificado por Python.
Avanzando en este libro, aprenderemos todas las palabras reservadas y sus usos correctos.

2.3. Usando el Modo Interactivo

2.3.1. Evaluación de Expresiones

Vamos a usar el modo interactivo para empezar a explorar Python.

Expresión

Los tipos de datos que ya conocimos son representados por los literales, o por las variables.
Empezaremos usando solamente los literales e incluiremos las variables más adelante.

Una **expresión** es
una combinación de literales, variables, operadores, funciones y símbolos de agrupación,
la cuál el interprete puede evaluar, ejecutando las operaciones y las funciones si las hay,
para producir un resultado.

Un Literal es Una Expresión

Las expresiones más sencillas son las que consisten de solamente un literal.
Acuérdense de oprimir la tecla [**Entrar**] al completar el literal en los ejemplos:

```
>>> None
>>> False
False
>>> True
True
>>> 123
123
>>> 9.87
9.87
>>> "Hola"
'Hola'
```

Al recibir una expresión, el intérprete la evalúa y reporta el resultado de esta evaluación.
Observen que el None (ninguno) no es una expresión, el Python no respondió a este literal.
Observen que Python usa las comillas sencillas para representar las cadenas: 'Hola'.

El "Traceback" de la Excepción

Cuando nos equivocamos y el intérprete de Python no entiende que queremos que haga,
el intérprete "**levanta una excepción**". Así se dice "responde a un error" en Python.
El resultado usual de levantar una excepción es imprimir un reporte llamado "**Traceback**".
El "Traceback" quiere decir algo como: "rastrear hasta el punto del origen de error".
El "Traceback" es nuestro mejor amigo, que nos ayuda a identificar la ubicación y el tipo de error.

Veamos un ejemplo; provocaremos una excepción tecleando punto decimal dos veces:

```
>>> 1..23
  File "<stdin>", line 1
    1..23
       ^
SyntaxError: invalid syntax
```

Después de provocar una excepción, tecleando el punto decimal dos veces,
en lugar de recibir un resultado, recibimos un reporte que consiste de cuatro líneas de texto.
Tales reportes de Python es más practico leer **línea por línea, desde abajo hacia arriba**.

Analicemos que nos dice este "traceback":
La ultima, la cuarta línea "SyntaxError: invalid syntax" nos dice que hicimos un error sintáctico.
En Python no existen literales que usan dos puntos seguidos, y el intérprete levanta la excepción.

La tercera línea tiene el carácter ^ apuntando arriba al fin del literal en el cual se descubrió el error.

La segunda línea es la copia de la línea en la cuál se encuentra el error.

La primera línea nos dice en cuál archivo "File" y en cuál línea "line" de este archivo está el error.
Ahora estamos en modo interactivo y todavía no usamos ningún archivo con el código, así que
el intérprete reporta "<**stdin**>", lo que es abreviación de "**st**andar**d in**put" (entrada estándar).
La entrada estándar para el Python es el teclado. El reporte dice que el error llegó desde el teclado.

El número de línea reportado es 1, ya que nuestra expresión está en solamente una línea.
Pronto veremos que se pueden escribir expresiones de varias líneas.
En tales expresiones de varias líneas, el error puede estar en la línea anterior a la línea de código
reportada por el "traceback". Por ejemplo, un paréntesis no cerrado en la línea anterior, mientras
la línea reportada no completa el paréntesis, sino empieza una expresión nueva.

Aparte de tener la entrada estándar "<stdin>", la cual es le teclado de la computadora,
el Python también tiene "<**stdout**>", "**st**andar**d out**put" (salida estándar) el cual es la pantalla.
El tercero es el "<**stderr**>", "**st**andar**d err**or". También se refiere a la pantalla de la computadora.
Si la computadora tiene dos pantallas una puede servir como "<stdout>" para resultados y
la otra puede servir como el "<stderr>" para reportar los errores.
Cada uno de estos: "<stdin>", "<stdout>" y "<stderr>", lo podemos redirigir hacia los archivos,
para recibir datos de un archivo y/o reportar resultados y errores a los archivos..
La comunicación se puede dirigir inclusive a las otras computadoras, o a la impresora de la red.
Esto se estudia al nivel avanzado. A este nivel usaremos archivos de otra manera, más sencilla.

Vamos a provocar un error más, tecleando espacio en lugar de punto decimal.

```
>>> 1 23
  File "<stdin>", line 1
    1 23
       ^
SyntaxError: invalid syntax
```

Analicen este reporte de error leyendo linea por linea desde abajo hacia arriba,
como lo hicimos en el ejemplo de error anterior.
Vamos a usar las palabras "error" y "excepción" como sinónimos.

2.3.2. Las Operaciones con Números

En el primer nivel de las expresiones, cada expresión fue simplemente un literal.
En el siguiente nivel, el que vamos a empezar a practicar ahora,
las expresiones son combinaciones de operadores y literales como sus operandos.

Practicaremos las operaciones aritméticas aplicadas a literales numéricas.
Las **operaciones** básicas son: **suma**, **resta**, **multiplicación**, **división**, y **potenciación**.
Sus **operadores** correspondientes son los: +, **-**, *, /, **.
La operación de potenciación sirve tanto para calcular las **potencias** como para calcular las **raíces**,
ya que la raíz n-esima es lo mismo que la potencia con exponente (1 / n).

```
>>> 2 + 3              # Suma
5
>>> 2 - 3              # Resta
-1
>>> 2 * 3              # Multiplicación
6
>>> 2 / 3              # División
0.6666666666666666
>>> 2 ** 3             # Potencia
8
>>> -8 ** (1 / 3)      # Raíz
-2.0
```

Cómo podemos ver al ejecutar "2 / 3", la división de enteros no produce entero sino un flotante.
La llamamos "división flotante" por el tipo de dato de su resultado.

Si no deseamos resultado de división en forma de un flotante, o sea no deseamos decimales,
también existe la división de enteros, la cual consiste de dos operaciones:
el **cociente** con el operador "división entera" // y
el **residuo** con el operador "modulo": %.

```
>>> 7 // 3
2             # Cociente
>>> 7% 3
1             # Residuo
```

El cociente 2 nos dice que el divisor 3 cabe 2 veces dentro del 7.
El residuo 1 nos dice que después de que el 3 quepa 2 veces, todavía falta 1 para llegar al 7.
O sea, 7 = 3 * 2 + 1.

Regresemos a las expresiones.

También tenemos a nuestra disponibilidad dos operadores unarios: + y -.
El operador + no afecta en número, mientras el operador - le cambia el signo:

```
>>> +(1)
1
>>> -(1)
-1
>>> -(-1)
1
```

Los paréntesis son los símbolos de agrupación.
Nos permiten controlar el orden de ejecución de las operaciones.
El Python respeta la precedencia de los operadores matemáticos.
Primero se ejecuta la potenciación, después multiplicaciones y divisiones y al fin sumas y restas.
Se puede ver el orden de precedencia de todos los operadores de Python
en la sección 22.2 en la página 204.

Por ejemplo en la expresión: 5 + 4 * 3 ** 2,
primero se ejecuta la potencia y esto resulta en: 5 + 4 * 9.
Después se ejecuta la multiplicación, lo que resulta en: 5 + 36.
Al fin se hace la suma y el resultado es 41.

```
>>> 5 + 4 * 3 ** 2
41
```

Si deseamos alterar el orden de la ejecución de los operandos usamos los paréntesis.
lo que está en paréntesis tiene la precedencia más alta.

Ejecutando: (5 + 4) * 3 ** 2.
Primero se calcula lo dentro de la paréntesis: 9 * 3 ** 2.
Después sigue la potenciación: 9 * 9 y al fin la multiplicación: 81.

```
>>> (5 + 4) * 3 ** 2
81
```

¿Cuál será el resultado de: ((5 + 4) * 3) ** 2)?

Expresiones Largas

En lugar de teclear la expresión ((5 + 4) * 3) ** 2, podemos hacerlo parte por parte, así:

```
>>> 5 + 4
9
>>> _ * 3
27
>>> _ ** 2
729
```

El símbolo de subrayado _ representa el resultado de la última evaluación.
Nos permite completar la expresión en el siguiente prompt.

La otra manera de manejar expresiones largas es no cerrando los paréntesis.
Si no cerramos el paréntesis, el intérprete empezará la segunda linea con prompt distinto: "..." y esperará la continuación de la expresión en esta nueva línea:

```
>>> (1 + 2 + 3 +
... 4 + 5)
15
```

Abrir los paréntesis y no cerrarlos hasta que la expresión esté completa,
es una técnica de escribir expresiones y declaraciones a través de varias líneas.
Esto lo podemos usar tanto en modo interactivo como en los archivos de código Python al programar.

Existe otra manera, algo anticuada y no muy recomendada, para señalar a Python
que la expresión o declaración no está completa y continuará en la siguiente línea.
Esta manera es basada en el uso del símbolo \ al final de la línea no completada.

Usando este símbolo, el último ejemplo también puede escribirse así:

```
>>> 1 + 2 + 3 + \
... 4 + 5
15
```

Esta manera de continuar una expresión también es valida.

Existen otros pequeño conjunto de los operadores para trabajar con los números.
Los estudiaremos en el siguiente capitulo.

2.3.3. Las Operaciones con Cadenas

También podemos procesar las cadenas de caracteres, o sea, los literales alfanuméricos:
Recuerden que las cadenas de caracteres se escriben entre comillas dobles, o sencillas.
El interprete siempre responde usando las comillas sencillas.

```
>>> "Hola. "
'Hola. '
```

Suma de dos cadenas genera una cadena nueva, la cual es la concatenación de las cadenas.

```
>>> “Hola. “ + “¿Cómo está? ”
'Hola. ¿Cómo está? '
```

Al multiplicar una cadena por un entero,
la cadena se repite, o sea se concatena a si misma para aparecer dada cantidad de veces:

```
>>> 'Hola. ' * 2
'Hola. Hola. '
```

Y que pasa si no usamos las comillas:

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

El “Traceback” es la manera en la que el intérprete de Python nos reporta que hay un error.
La última línea nos dice: “ErrorDeNombre: el nombre 'Hola' no está definido”.
La penultima línea nos dice que el error llegó desde el teclado: “<stdin>”.
La antepenultima, o sea la primera línea nos dice que se trata de un “Traceback”.

En este caso, en la línea

```
>>> Hola
```

la que para el interprete es la primera linea “line 1” ya que las anteriores respondió,
a la palabra Hola le faltaron comillas dobles, o simples.
Sin comillas, el intérprete entiende que Hola es un nombre.

Existen más operaciones con las cadenas. Las estudiaremos en el siguiente capitulo.

2.3.4. Las Funciones

Aparte de los operadores, también se usan las **funciones** para trabajar con los datos de Python.

Cada función tiene su **nombre** que la identifica.
Para decir al intérprete que deseamos que ejecute una función su nombre debe ser seguido
por el “**operador de ejecución**” el cual son un par de paréntesis ( ).
Dentro de las paréntesis se “mandan” a la función los **argumentos**,
o sea los datos sobre los cuales va a trabajar.

Hay funciones que no necesitan argumentos, entonces se aplica un par de **paréntesis vacíos**.
Los ejemplos de tales funciones son “quit” y “exit” que usamos para salir de modo interactivo.

Veamos ejemplo de una función que recibe dos argumentos y efectúa la división de enteros.

La función "divmod" (división entera, modulo) la cuál recibe dos argumentos: dividendo y divisor y nos **devuelve** dos números: el cociente y el residuo.
Por ejemplo:

```
>>> divmod(7, 3)
(2, 1)
```

En la división flotante esto es:

```
>>> 7 / 3
2.3333333333333335
```

Observen el último dígito en el resultado de la división flotante. Es un 5. Raro, ¿no? Debería ser un 3.
Vamos a tener mucho que hablar sobre los números de tipo flotante y su uso en la computación; en particular sobre su representación en la computadora, la precisión de los resultados y la comparación de dos flotantes para saber si son iguales, o distintos.

Conoceremos una variedad de funciones a lo largo de este libro,
hasta aprenderemos a escribir nuestras funciones propias.

El Python ofrece un conjunto de **funciones incorporadas** y otras funciones de sus librerías.
La lista de las funciones incorporadas está en la sección 22.3 en la página 206.
Poco a poco, aprenderemos a usar todas estas funciones en los ejercicios de programación.
No hay necesidad de memorizar estas funciones ahora. Las vamos a introducir una por una.

2.4. Asignación de Objetos a Variables

A veces necesitamos guardar el resultado de una expresión en la memoria,
para poder reusarlo en otras expresiones.

Python nos permite esto a través de uso de las variables.
Cada variable se define al ser asignada el resultado de una expresión.

La forma de hacer esto es la siguiente:

```
<nombre_de_la_variable> = <expresión>
```

Seguiremos usando el modo interactivo, pero ahora también iniciaremos con el modo "script", o sea, programación.

Una "expresión" tiene que estar escrita de manera sintácticamente correcta,
para que el intérprete puede efectuar las operaciones y producir el resultado.

Una "expresión" se evalúa y produce resultado el cual es un "objeto" de algún tipo de datos.

Los Objetos y sus Variables

Un dato, de algún tipo reconocido por el intérprete de Python, es un "objeto".
El concepto de objeto abarca muchas más cosas que datos.

Por lo pronto podemos decir que cualquier información que se guarda en la memoria
durante la ejecución de un programa Python, es un objeto.
Inclusive partes de código, como:
una expresión, una declaración, una función, una clase, o un modulo,
son objetos.

Los objetos se guardan en la memoria RAM, o en la memoria cache.
El acrónimo RAM viene de "Random Access Memory".

A veces queremos guardar el resultado de una expresión, o sea un objeto,
en la memoria para el uso posterior. Esto se logra usando declaración de asignación:

```
<variable> = <expresión>
```

Lo que está a la izquierda de = es un "lvalue" ("left-side value")
Lo que está a la derecha de = es un "rvalue" ("right-side value")

Observen que el símbolo = no representa igualdad, o ecuación, como en las matemáticas.
El símbolo = es el "operador de asignación". Esto significa que
el objeto que resulta de evaluar la expresión se guarda en un lugar en la memoria,
y a la variable se le asigna el número identificador, el domicilio, de este lugar en la memoria.

por ejemplo:

```
>>> edad = 23
```

A la izquierda de = son siempre una o varias variables. Aquí, la variable tiene nombre "edad".
A la derecha de = es siempre una expresión. En este caso, la expresión consiste de un literal: 23.

La expresión se evalúa y esto produce un objeto: entero 23.
El operador de asignación, el signo =, indica que el objeto se va a guardar en la memoria.
El nombre de la izquierda de =, el "edad", se guarda en otra parte de la memoria,
en una tabla llamada "espacio de nombres, "namespace",
donde el nombre es relacionado con el número identificador del domicilio de la memoria,
donde se guardó y donde empieza la escritura/lectura del objeto: entero 23.
Se dice que ahora la variable "edad" está "referenciando" el objeto entero 23.

El nombre que usamos como la variable le sirve al intérprete para leer el objeto de la memoria y
sirve a los programadores para entender que representa el objeto "entero 23":
en "edad = 23", el entero 23 representa la edad de alguien y
nombramos la variable "edad" para que nos recuerde a esto.
Hay que evitar usar nombres sin significado claro, como "pgf_3_x" y los nombres de este estilo.

Nombres Válidos

Hasta cierto punto estamos libres de elegir los nombres de las variables.
Para que un nombre sea aceptado por parte del intérprete tiene que cumplir con lo siguiente:

1. El primer símbolo tiene que ser ya sea una letra, o el símbolo de subrayado _.
2. Los demás símbolos aparte de letras y _, también pueden ser dígitos de sistema decimal.

Aparte de esta regla sintáctica de Python
el estándar PEP8 recomienda que nombres de variables sean escritos en minúsculas y
si hay varias palabras, que se separen por el símbolo de subrayado _.

Por ejemplo:

```
nombre = ...
nombre_de_objeto = ...
```

Si tenemos intención que un objeto no se modifique, que quede constante,
su nombre se escribe en mayúsculas, con palabras separadas por el subrayado _.

```
CONSTANTE_DE_PROPORCIÓN = ...
```

El intérprete de Python no maneja constantes, ni garantiza que se mantendrán constantes.
El escribir un nombre en mayúsculas es solamente para programadores que van a reusar el código.
Si el programador desea modificar la supuesta constante, el intérprete no lo va a impedir.
Escribir en mayúsculas sirve para mostrar la intención de usar esta variable como constante, y
ayuda a los programadores, incluyendo a nosotros mismos, comprender el código.
Es importante usar mayúsculas para las variables que representan el valor que no debe cambiar.

2.4.1. Definición de Variable

Un literal que deseamos solamente imprimir, no necesitamos guardar en la memoria:

```
print(23)
```

pero si deseamos usar este objeto entero 23 en varios lugares de código,
lo guardamos en la memoria usando asignación a una variable y
después podemos acceder a este objeto por su nombre y usarlo,.
Esto le explicaríamos al intérprete de Python así:

```
edad = 23
print(edad)
```

En la primera declaración el objeto 23 es guardado en la memoria bajo el nombre "edad".
En la segunda declaración el objeto es recuperado por su nombre y mandado a la impresión.
Lo que se imprime no es el nombre "edad", sino el 23.

A veces queremos definir un nombre de variable,
pero todavía no queremos referenciar un objeto.
Para este propósito se usa el objeto tipo genérico: None.

```
edad = None
```

Vamos a ver usos prácticos de tipo genérico, None, en futuros ejemplos.

2.4.2. Uso de la Memoria

El Uso y la Re-asignación de Variable

Después de ser definida, la variable puede ser usada para leer el objeto de la memoria.
Una variable también puede ser reasignada a otro objeto.

```
edad = 23        # Guarda 23 en la memoria, referenciada por “edad”.
print(edad)      # Nombre “edad” es sustituido por entero 23, y se imprime.
edad = edad + 1  # “edad = 23 + 1”. El “edad” ahora referencia a entero 24.
print(edad)      # Nombre “edad” es sustituido por entero 24, y se imprime.
```

En el tercer renglón primero se evalúa la expresión de lado derecho de =, la expresión “edad + 1”.
Podemos ver que esta expresión contiene variable, operador y literal.
El intérprete primero sustituye la variable en la expresión por el objeto que representa.
En seguida se evalúa la expresión “23 + 1” y se produce el objeto: entero 24, como el resultado.
Declaración de asignación hace que el entero 24 se guarde en un domicilio en la memoria,
y ahora este domicilio queda referenciado por el nombre “edad”.

El lugar que contiene el entero 23 queda des-referenciado y la memoria que está ocupando
es recuperada automáticamente por un proceso de intérprete llamado “colector de basura”.

Domicilio de Memoria como el Identificador de Objeto

Los domicilios en la memoria se identifican por enteros desde 0, 1, 2, hasta el ultimo domicilio.
Cuál es el último número identificador, depende del tamaño de la memoria.

Cada objeto guardado en la memoria tiene su identificador,
el cual es el número de domicilio en la memoria donde empieza la lectura del objeto.

Si deseamos saber el numero identificador, o sea el domicilio de la memoria,
donde empieza la lectura de un objeto, usamos la función “id”.

Ejecute el siguiente código como script, o en modo interactivo:

```
edad = 23
print(edad)
print(id(edad))
edad = edad + 1
print(id(edad))
print(edad)
```

Observe que la variable “edad” cambia de identificador en el momento de ejecutarse la asignación

```
edad = edad + 1
```

Antes de esto la variable “edad” reverenciaba el objeto 23 en algún domicilio,
después, referencia el objeto 24 en algún otro domicilio.

Los Objetos “mutables” e “inmutables”

Los objetos que guarda el intérprete de Python en la memoria se dividen en dos grupos:
objetos inmutables y objetos mutables.

Todos los tipos de datos que conocemos hasta ahora:
genérico, lógico, entero, flotante y cadena son objetos inmutables.

A los objetos inmutables no se les puede modificar el valor en el lugar de memoria donde están.
Por ejemplo, si tenemos asignación

```
a = 9
```

El objeto 9 es un entero y por lo tanto inmutable.
el "a" es la variable que está referenciando el objeto 9.

Si después reasignamos "a":

```
a = 8
```

El domicilio de la memoria donde está el objeto 9 no va a ser modificado para que sea 8.
Así sucedería con un objeto mutable, pero objeto tipo entero es inmutable.

Como en todas asignaciones, primero se evalúa la expresión de lado derecho de =,
lo cual produce objeto entero con valor 8. Este objeto se guarda en la memoria y
el identificador de este lugar en la memoria se asocia con la variable "a" en la tabla de nombres.

En la tabla de nombres ("namespace"), no puede haber dos entradas "a", así que
el único "a" de la tabla ahora está referenciando el objeto 8 y el objeto 9 queda des-referenciado.
La memoria que es usada para el objeto 9 es recuperada por el "colector de basura" y
está disponible para guardar un nuevo objeto que pueda aparecer en el resto del código.

Las variables en Python no se usan como las variables en otros lenguajes de programación.
Una variable en lenguajes compilados seria lo que en Python es un nombre junto con su objeto.

La primera diferencia entre variable en Python y en C++ por ejemplo,
es que variable en C++ tiene tipo de dato.
En Python la variable en sí no tiene tipo de dato, el tipo de dato lo tiene el objeto referenciado.

Así que en Python es valido lo siguiente

```
dato = 12
dato = "cadena"
```

En C++ una variable se declara e inicializa definiendo su tipo de dato:

```
int dato = 12
```

y siendo esta variable de tipo entero ("int") no se le puede asignar otro tipo de dato.

La segunda diferencia entre variable en Python y variable en C++
es que en Python existen objetos inmutables.

La asignación:

```
dato = dato + 1
```

en C++ causaría que el objeto 12 en la memoria se sobre-escriba con el valor nuevo: 13.
En Python enteros son inmutables y un entero no se sobre-escribe, se crea nuevo objeto: entero 13
y en la tabla de nombres se modifica la variable "dato" para referenciar este nuevo objeto.
La memoria que guardaba el objeto anterior, entero 12, es recuperada por el "colector de basura".

El "Colector de Basura"

Cuando un nombre cambia la referencia a otro lugar en la memoria RAM,
y el lugar anterior no tiene ningún otro nombre que lo mantiene referenciado,

hay una parte de intérprete de Python llamada "colector de basura"
que se encarga de liberar el lugar anterior para que pueda ser reusado para objetos nuevos.
El colector de la basura es un proceso independiente del intérprete y
puede tardar entre milésimas de segundo hasta un par de segundos para liberar el lugar.

Si desea conocer más sobre el colector de basura y modificar su funcionamiento
hay que estudiar el modulo "gc" ("garbage collector") de la librería estándar de Python.
https://docs.python.org/3/library/gc.html

2.5. Modo Programación "Scripting"

2.5.1. La Sintaxis y el Estilo

Los programas de Python consisten de uno o varios archivos de texto llamados "módulos.
El texto es código escrito en el lenguaje Python para que le intérprete lo pueda ejecutar.
Al escribir un modulo tenemos que pensar en dos lectores:

- el intérprete de Python que necesita ejecutar el código y
- los programadores que necesitarán entender y usar o modificar el código.

Para que el intérprete ejecute el modulo necesitamos respetar la sintaxis de lenguaje,
o sea el uso correcto de los símbolos, elementos, expresiones y declaraciones de lenguaje Python.

La sintaxis de Python todavía deja mucha libertad en cuanto el estilo de organización de código.
Para facilitar entendimiento del código a los programadores (incluyéndonos a nosotros mismos),
tenemos que adoptar un estilo de organización de código y apegarnos a el.

Un estilo, prácticamente "estándar" para los programadores Python,
está definido en el documento "Python Enhancement Proposal 0008", PEP-8.
Este estilo es el más usado y es recomendado por profesionales en lenguaje Python.

Todos los ejemplos en este código están respetando las recomendaciones PEP8.

Al avanzar con el lenguaje Python, se van a presentar las recomendaciones de PEP-8,
en cuadros como el que sigue, según estemos listos para entenderlos.

Las recomendaciones de PEP8 tienen como objetivos
promover entendimiento fácil de código Python 3 y
lograr la consistencia de estilo entre los proyectos.

Consistencia con las recomendaciones de este documento es importante
Consistencia dentro de un proyecto es más importante.
consistencia dentro de un módulo es todavía más importante.

Lo más importante es: saber cuando NO hay que ser consistentes.

A veces las recomendaciones PEP8 simplemente no aplican.
Cuando estén en duda, usen lo mejor de su juicio y
consulten el asunto con sus colegas que trabajan en el mismo proyecto.

Dos buenas razones para romper con alguna recomendación:
(1) Cuando aplicación de la recomendación hace que código sea difícil de comprender,
aún para alguien con experiencia en Python y las recomendaciones de PEP8.
(2) Cuando es necesario ser consistente con el código ya existente.
Aunque esta es buena oportunidad para limpiar este código viejo para que cumpla con PEP8

En cuanto a escribir módulos (archivos de código) Python, aplican las siguientes recomendaciones.

El código Python no debe sobrepasar 79 caracteres por línea,
más el ultimo carácter invisible, "nueva línea", '\n', como 80-avo.

Hay que usar letra de tipo "typewriter", "mono" o
cualquier tipo de letra que tenga todos los caracteres de misma anchura.

El tabulador '\t' debe ser equivalente a 4 caracteres, o sea 4 espacios.
No hay que mezclar tabuladores con espacios. Lo más recomendable es no usar tabulador.
En varios editores de texto es posible usar la opción "sustituir tabulador con espacios"
de manera que todos los tabuladores '\t' son convertidos automáticamente en 4 espacios " ".

No hay que dejar espacios vacíos al final de las líneas, ni líneas vacías al final del texto.

Hay que guardar el texto en el formato "utf-8".

Los nombres de módulos Python se escriben en letras minúsculas,
con palabras separadas por subrayado "_" y con extensión ".py".

Existe un programa "pep8" que revisa si el código está respetando las recomendaciones del PEP8.
Se le puede instalar de Internet y usar desde línea de comandos de su sistema operativo como:

```
pep8 nombre_del_modulo.py
```

Existen más programas para evaluar código Python, como son "flake8", "pylint" y otros.
Para nuestros propósitos es recomendable el más sencillo: "pep8"

Si no desean instalar el programa pep8, existe una página de internet:
http://pep8online.com/
donde pueden copiar el texto de su modulo y
oprimir el boton "Check code" para que el pep8 verifique so código.
Modifiquen su código y verifíquenlo otra vez hasta que el pep8 ya no reporte errores.
Entonces pueden copiar su código limpio de la página de Internet, atrás a su editor de texto.

La filosofía de los "pythonistas", está resumida en un muy corto documento: PEP-20.
Busque el PEP-20 "El zen de Python" (The Zen of Python) en el Internet y lea lo.

2.5.2. La Salida Estándar: la Pantalla

Para que un programa sea útil debe de hacer algo.
En gran mayoría de programas se escribe algo en la pantalla para el usuario.
Es una acción común a muchos programas, así que empezaremos con programación
escribiendo programas que simplemente presentan datos en la pantalla.

La Función "print"

Para imprimir datos en la pantalla, se usa la función "print".
El siguiente programa imprime literales de varios tipos de datos.
Los literales los conocemos en las matemáticas como constantes numéricas y alfanuméricas.

```
# Imprimiendo literales de distintos tipos de datos:
print()                           # Imprime '\n': o sea, "nueva linea"
print(None)
print(True, False)
print(321)
print(0.123)
print("Soy una cadena con gato #.")
print()

# Todos en una sola declaración:
print(None, True, False, 321, 0.123, "Soy una cadena con gato #.")
print()
```

Al encontrar el símbolo "gato", #, que no es parte de alguna cadena, o sea, no está entre comillas,
el intérprete de Python brinca a leer la siguiente línea. Lo escrito después del símbolo #
son los comentarios que se usan para facilitar entendimiento del código a los programadores.

Usamos los comentarios para avisar otros lectores de nuestro código, los colegas programadores,
sobre nuestras intenciones, asunciones, problemas encontrados, detalles importantes y similar.

La palabra "print" es el nombre de la función de impresión en la salida estándar, la pantalla,
y esta función está definida dentro del intérprete de Python.
Después aprenderemos a diseñar nuestras propias funciones.

Los paréntesis () son el operador de ejecución.
Al poner los paréntesis después del nombre de una función, como en la línea

```
print()
```

le estamos dando la instrucción al intérprete que ejecute la función "print".

El interior de los paréntesis se puede usar para pasar a la función los datos
con los cuales va a trabajar. En caso de la función "print", los datos se presentarán en la pantalla.
Los datos pasados a una función se les dice "argumentos".

Escriban este código en un editor de texto de su preferencia asegurando se de que cada carácter, incluyendo espacios y líneas vacías, esté tecleado exactamente como lo muestra el ejemplo. Guarden el modulo con el nombre "función_print.py" en una carpeta, o sea directorio, en el cual van a guardar todos los ejemplos de este texto. Este directorio es su "directorio de trabajo".

Para formar nombres de los módulos usamos palabras en letras minúsculas y
si son varias palabras, las separamos con subrayado _.
Al final del nombre de modulo se escribe la extensión ".py".

Las palabras reservadas no deben ser usadas como nombres de módulos.
La lista de palabras reservadas se puede consultar
en la sección 22.1 en la página 204.

2.5.3. Ejecutar un "Script" Python

Los archivos de texto en los que guardamos nuestros códigos de Python, con extensión ".py", los llamamos "módulos". Al querer ejecutar uno de los módulos, este lo llamamos el "script".

Existen varias maneras de ejecutar un script de Python.
Si deseamos ejecutar el script desde línea de comandos,
("terminal" en Linux y Mac, "DOS-prompt" en Windows)
debemos movernos al directorio (carpeta) en la cual está guardado el archivo y teclear:

```
python3 función_print.py
```

En caso de usar IDLE, el script se puede ejecutar usando el menu:

```
Run ---> Run module
```

El modulo debe ser guardado en el disco para que el IDLE lo ejecute,
de otra manera, IDLE va a pedir que el archivo sea guardado.

Si usan algún otro entorno de desarrollo, consulten el manual sobre como ejecutar su programa.

Para quedar dentro del modo interactivo despues de ejecutar el script usan la opción "-i":

```
python3 -i función_print.py
```

2.5.4. Ejercicios

Escriba un programa que ...

1) presenta su nombre, edad, altura en metros, en la pantalla, un dato por línea.

2) presenta su nombre, edad, altura en metros, en la pantalla un dato por línea,
de manera que cada línea empieza con el símbolo gato y un espacio: "# ".

3) presenta su nombre, edad, altura en metros, todos en una misma línea.

2.6. Librería Estándar

La librería estándar de Python (PSL, "Python Standard Library") ofrece muchas componentes:
paquetes, módulos, clases, funciones, variables, constantes,
para ser usadas y reusadas en los proyectos hechos en Python.

En las secciones llamadas "Librería Estándar" vamos a ver algunos de estos componentes reusables,
para poder hacer programas con más capacidad y más sencillos, ya que
no necesitamos programar lo que ya está hecho para nosotros.

2.6.1. El Modulo "sys", Sistema e Intérprete

El intérprete contiene información sobre el lenguaje Python, tipos de datos incorporados y
también ofrece un conjunto de funciones como son "print", "format" y otras por conocer.

Además de lo que nos ofrece el intérprete mismo, también tenemos a nuestra disposición
la librería estándar de Python: "Python Standard Library" PSL.
https://docs.python.org/3/library/

La librería estándar normalmente está instalada junto con el intérprete.

La PSL es muy extensa y puede tomar hasta años para dominar todos sus paquetes y módulos.
Mayoría de los módulos está especializada para ciertos tipos de aplicaciones.
Son suficientes unos cuantos módulos, más usados, que vamos a conocer en este texto,
para empezar a programar en Python. No hay necesidad de conocer todos los módulos.

El primer módulo que vamos a empezar a conocer se llama "sys", abreviado de "system".
https://docs.python.org/3/library/sys.html

Cada computadora es controlada por su sistema operativo y cuando el intérprete de Python
está ejecutando código Python, lo hace en colaboración con el sistema operativo.
El modulo "sys" nos ofrece información sobre el intérprete de Python y
su relación con el sistema operativo. Lo vamos a estudiar a lo largo del texto.
Después añadiremos el modulo "os", abreviado de "operating system",
para usar los servicios de sistema operativo desde nuestros programas.
Por ahora vamos a ver algunos servicios que el "sys" nos ofrece.

Para usar un modulo de la librería estándar, primero lo tenemos que importar.
Esto se logra usando la declaración "import":

```
>>> import sys
```

Cuando queremos usar alguna información, o función, de "sys", esta siempre lleva el prefijo "sys.".

En Modo Interactivo

El "sys.executable" nos dice donde en nuestra computadora está guardado
el intérprete con el cual estamos trabajando.

```
>>> sys.executable
'/usr/bin/python3'
```

Desde un script debemos usar

```
print(sys.executable)
```

Más información sobre el intérprete con el cual trabajamos, se puede obtener usando:

```
>>> sys.version
'3.4.3 (default, Oct 14 2015, 20:28:29) \n[GCC 4.8.4]'
>>> sys.version_info
sys.version_info(major=3, minor=4, micro=3, releaselevel='final', serial=0)
>>> sys.implementation
namespace(_multiarch='x86_64-linux-gnu',   cache_tag='cpython-34',
          hexversion=50594800, name='cpython', ...
```

Para saber sobre cual plataforma se está ejecutando el intérprete usamos:

```
>>> sys.platform
'linux'
```

Las posibles respuestas son:
Linux: 'linux', Windows: 'win32', Windows/Cygwin: 'cygwin', Mac OS X: 'darwin'.

Los que trabajan en Windows pueden usar "sys.getwindowsversion()",
para averiguar cual es la versión de Windows que tienen.

También podemos indagar sobre los tipos de datos incorporados:

```
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

Si queremos saber cuantos bytes de la memoria va a ocupar un dato,
podemos usar la función "getsizeof":

```
>>> sys.getsizeof("mi nombre")
58
>>> sys.getsizeof(123)
28
>>> sys.getsizeof(1.23)
24
```

2.7. Tareas

Escriba un programa que ...

1) asigna su primer nombre a la variable "mi_nombre" e imprime su nombre usando la variable.

2) asigna su edad a la variable "edad" e imprime su edad usando la variable.

3. No Hay Civilización Sin las Matemáticas

En este capitulo nos recordaremos de las matemáticas aprendidas en la secundaría.
Necesitamos repasar estos conceptos para entender la programación de computadoras.

3.1. Números Enteros

Conjuntos de números

división entera

sistemas de números: binario, octal, hexadecimal

3.2. Fracciones

Clases de fracciones

Porcentajes

denominador común

Fracciones a binario

3.3. Reales

división decimal

Precisión notación científica

3.4. Números Complejos

Tal vez en su educación todavía no han visto números complejos.
Aquí vamos a estudiar una introducción elemental a los números complejos,
ya que los necesitamos para completar todos los tipos de números que usa el lenguaje Python.

De todos modos, a nivel que los vamos a estudiar, los números complejos son muy sencillos.

Ejercicios

4. Entrada, Procesamiento, Salida

mencionar condiciones, "if" y abstracción con el concepto de función

4.1. La Entrada Estándar: el Teclado

Ya aprendimos a crear variables y asignar algún objeto a ellas. También aprendimos a formatear e imprimir diferentes tipos de literales y variables usando las funciones "format" y "print".

Lo siguiente es aprender como obtener datos desde el usuario del programa,
a través de la entrada estándar: el teclado.

4.1.1. Entrada de Una Cadena

Para recibir datos desde el teclado se usa la función "input":

```
input()
print("Gracias.")
```

Si ejecute programa con esta expresión, la función "input" va a parar la ejecución del programa y esperar una cadena de caracteres, una cadena, desde el teclado.
Para señalar el fin de la cadena de entrada, se oprime la tecla "Enter", o sea "Return".

Cando el intérprete llegue a la declaración "input()" va a parar el programa y
el usuario puede estar sorprendido sin saber de que se trata.

Para que el usuario sepa por que se detuvo la ejecución del programa, la función "input"
puede escribir un mensaje en la pantalla. Por ejemplo:

```
input("Presione la tecla [Enter], o sea [Return], para continuar. ")
print("Gracias.")
```

Cuando el intérprete lea esta expresión, va a llamar la función "input".
La función input va a escribir el mensaje dado en la pantalla
y va a leer la entrada de caracteres desde el teclado,
hasta que se oprima la tecla "Enter", o sea "Return".
Después de esto la ejecución de programa continúa.

A diferencia de "print", la función "input" recibe como argumento solamente una cadena y
no tiene los parámetros "sep" y "end" para formatear lo que va a imprimir.

La función "input" es una expresión que produce un objeto de tipo cadena,
y esta cadena contiene el texto recibido del teclado.
Si deseamos guardar el texto recibido del teclado en la memoria,

debemos usar la declaración de asignación en la misma línea,
como lo muestra el siguiente ejemplo.

Ejecute el siguiente código:

```
nombre = input("¿Cómo te llamas? ")
print("Hola, ", nombre, ".", sep="")
```

La función "input" recibe el texto tecleado en una memoria llamada "buffer".
La declaración de asignación =, causa que este texto se guarda como un objeto en la memoria y este objeto queda referenciado por la variable "nombre".

La función "print" imprime tres objetos: "Hola", el nombre y el ".".

4.1.2. Conversión a Otros Tipos de Datos

El objeto producido por la función "input" es siempre de tipo cadena.
Para convertir esta cadena a otro tipo de dato, se usan las funciones de conversión como son: bool(), int(), float(), y otras que vamos a aprender, entre cuales str().

Lo siguiente no va a funcionar:

```
edad = input("¿Cuántos años tienes? ")
edad = edad + 1
print("El siguiente año tendrás", edad, "años.")
```

A la variable "edad" en la primera declaración se le asigna el objeto tipo "cadena".
La segunda declaración trata de sumar entero 1 a esta cadena, lo que no es valido.
Para lograr lo intencionado, hay que convertir el resultado de "input" a un entero,
antes de intentar a sumarle un 1:

```
edad = int(input("¿Cuántos años tienes? "))
edad = edad + 1
print("El siguiente año tendrás", edad, "años.")
```

El objeto que resulta de la función "input" es entregado a la función "int"
para que lo convierta en un objeto te tipo "entero".
Si lo tecleado no es un entero valido, la función "int" reportará una excepción al intérprete.

Si esperamos recibir un número de punto flotante, aplicaríamos la función "float":

```
edad = float(input("¿Cuántos años tienes? "))
```

Resumido: para recibir datos del teclado se usa la función "input",
la cual está opcionalmente acompañada con una función de conversión de tipo de datos y está acompañada con la declaración de asignación para guardar lo recibido del teclado.

En el proceso de depurar el programa de algún error, podemos usar "Input" sin asignación:

```
input("Presione Enter para continuar. ")
```

ya que no nos interesa recibir ni guardar nada de teclado.
El "input" fue usado solamente para interrumpir ejecución.

Como vimos en el ejemplos anteriores, no se pueden sumar datos tipo entero con datos tipo cadena.
En la continuación estudiaremos cuales operaciones de Python son validas
para cada tipo de datos y cuando y como se pueden mezclar
distintos tipos de datos en la misma expresión.

4.1.3. Ejercicios

Escriba un programa que ...

1) asigna su edad a la variable "edad", le suma 1 a la variable y usando la variable imprime

```
El próximo año tendré ?? años.
```

donde en lugar de ?? es el número de años que usted tendrá en próximo año.

4.2. Procesamiento de Datos

4.2.1. Cadenas

Aquí se presenta solamente una introducción al tipo de dato "cadena".

Operaciones comunes a todas las colecciones:

```
x in s              True if an item of s is equal to x, else False
x not in s          False if an item of s is equal to x, else True
s + t               The concatenation of s and t
s * n or n * s      Eequivalent to concatenating s to itself n times
s[i]                The i-th item of s, origin 0 (3)
s[i:j]              Slice of s from i to j
s[i:j:k]            Slice of s from i to j with step k
len(s)              Length of s
min(s)              Smallest item of s
max(s)              Largest item of s
s.index(x[,i[,j]])  Index of the first occurrence of x in s
s.count(x)          Counts appearances of x in s
```

Conversión a Tipo Cadena

La función "str" ("string") convierte otro tipo de dato a una cadena.

```
str(123)        # produce cadena '123'
str(9.8)        # produce cadena '9.8'
```

También se puede usar la función "format" que conocimos en formateo de datos para imprimir.

```
format(<entero>, 'Nd')
format(<float>, 'N.Df')
```

La función "format" la usamos solamente dentro de la lista de argumentos,
para la función "print".

La Operación Suma + de Cadenas

La operación "suma" de dos cadenas, con operador +,
produce una cadena nueva, la cual es la concatenación de las dos cadenas originales.
O sea, las dos cadenas se pegan juntos para producir una nueva.

```
"Juan" + "Perez"           # da resultado "JuanPerez".
"Juan" + " " + "Perez"     # da resultado "Juan Perez".
```

La Operación Multiplicación * de Cadena y Entero

La operación "multiplicación" de una cadena y un entero N, con operador *,
produce una cadena nuevo que consiste de N cadenas originales concatenados.

```
"cadena " * 3              # da resultado "cadena cadena cadena ".
3 * "cadena "              # da resultado "cadena cadena cadena ".
"Juan" + 3 * " " + "Perez" # da resultado "Juan   Perez".
'-' * 12                   # da resultado '------------'.
```

Aplicaciones a la Función "input"

La función "input" recibe solamente un argumento tipo cadena.
Si deseamos imprimir varias cadenas con un "input", tenemos que concatenarlos.

Lo siguiente no funcionará:

```
nombre = input("¿Cual es su nombre? ")
edad = int(input("¿Cual es su edad, ", nombre, "? "))
```

La segunda declaración produce una excepción, ya que pasamos tres argumentos a "input".

La solución es concatenar las cadenas en una sola.

```
nombre = input("¿Cual es su nombre? ")
edad = int(input("¿Cual es su edad, " + nombre + "? "))
```

Se pueden concatenar las cadenas antes de llamar "input":

```
nombre = input("¿Cual es su nombre? ")
pregunta = "¿Cual es su edad, " + nombre + "? "
edad = int(input(pregunta))
```

Operación Índice []

Dada una cadena, la operación "índice", con operador [],
extrae un carácter de la cadena y forma nueva cadena de un solo carácter.

El primer carácter de la cadena tiene índice 0, el segundo 1 y así sucesivamente.

```
s = "cadena"
print(s[0])            # imprime "c"
print(s[1])            # imprime "a"
print(s[2])            # imprime "d"
```

También se pueden usar índices negativos, los cuales son relativos al fin de la cadena:
el último carácter de la cadena tiene índice -1, penúltimo -2 y así sucesivamente.

```
print("cadena"[-1])    # imprime “a”
print("cadena"[-2])    # imprime “n”
print("cadena"[-3])    # imprime “e”
```

Si el índice esta fuera de string se produce una excepción.

```
>>> "cadena"[6]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Operación Sub-Cadena [:] o [::]

La operación “sub-cadena”, con operador [:] u operador [::],
produce una nueva cadena el cual es un sub-cadena de la cadena original.

Observe que en la expresión [i:j]
i es el índice del primer carácter incluido en resultado
j es el índice del primer carácter no incluido en resultado.

```
# la expresión      # resulta en

"cadena"[1:4]       # “ade”
"cadena"[2:]        # “dena”
"cadena"[:3]        # “cad”
"cadena"[:]         # “cadena”
```

En la expresión [i:j:k], k es el tamaño de paso de avance.

```
>>> "cadena"[1:5:2]
’ae’
>>> "cadena"[1::2]
’aea’
>>> "cadena"[:-1:2]
’cdn’
>>> "cadena"[1::3]
’an’
>>> "cadena"[::-1]
’anedac’
```

El último resultado es la “cadena” invertida.

El operador [i:j:k] se puede sustituir por la función [slice(i, j, k)]

```
"cadena"[slice(None, None, -1)]      # es equivalente a “cadena”[::-1]
```

Funciones "len", "ord" y "chr"

La función "len" ("length") nos dice cuantos caracteres hay en una cadena.
Esto no es lo mismo que el número de bytes, ya que caracteres pueden usar
desde uno hasta cuatro bytes para su representación numérica en UTF-8.

```
>>> len("")
0
>>> len("Hola.")
5
>>> len(" _ ")
3
```

intérprete Python guarda las cadenas en la memoria RAM usando formato UTF-8.
A cada carácter de la cadena le corresponde un número entero ordinal como su código.

Con la función "ord" podemos obtener este número ordinal del carácter:

```
ord('a')     # produce entero 97
```

Con la función "chr" podemos obtener el carácter que corresponde a un código dado:

```
chr(97)      # produce cadena 'a'
```

4.2.2. Números

Conversiones Entre Tipos Numéricos

Las funciones "int", "float" y "complex" convierten otros tipos de datos a numérico correspondiente.

El uso de estas funciones ya vimos en la sección de entrada de datos desde teclado.

Operaciones Aritméticas

Las operaciones sobre los datos numéricos se dividen en: unarias y binarias.

Unarias son: + y -.

Binarias son: +, -, *, /, **, //, y %

Operaciones Sobre los Bits

4.2.3. Ejercicios

Escriba un programa que ...

1) le pide a usuario nombre de estado donde nació y un entero entre 3 y 7 inclusivos.
después imprime en la pantalla el nombre de estado,
repetido las veces que indica el entero proporcionado, separados con un espacio.
Por ejemplo:

```
Ingrese el nombre de estado donde nació o "extranjero": Jalisco
Proporcione un entero [3 a 7]: 5

Jalisco Jalisco Jalisco Jalisco Jalisco
```

2) le pide a usuario escribir nombre de país que le gustaría visitar y
en la pantalla imprime la primero y el último carácter del texto recibido.

3) le pide a usuario nombre de la ciudad donde la gustaría vivir e
imprime en la pantalla las letras en lugares impar dentro de la palabra y
en la siguiente línea imprime cada segunda letra del fin de la palabra, hacia adelante.

```
¿En cuál ciudad le gustaría vivir? Zapopan

Zppn
nppZ
```

4) le pide a usuario que proporcione nombre de su color favorito e
imprime cuantos caracteres fueron recibidos del teclado y en la siguiente línea
imprime el nombre del color al revés.

5) le pide a usuario proporcionar una letra de abecedario e
imprime el número ordinal UTF-8 de esta letra en la pantalla.

6) pide un número al usuario y lo escribe con signo opuesto.

7) pide dos números al usuario y reporta resultado de cuatro operaciones:
suma, resta, multiplicación y división, uno por linea. Por ejemplo

```
Proporcione el primer número: 8
Proporcione el segundo número: 2

8 + 2 = 10
8 - 2 = 6
8 * 2 = 16
8 / 2 = 4.0
```

8) pide un número positivo [0.0 a 100.0] al usuario y reporta cuatro resultados
con 3 dígitos decimales y alineados como en el ejemplo:

```
proporcione un número [0.0, 100.0]: 64

Raíz cubica:         4.000
Raíz cuadrada:       8.000
cuadrado:         4096.000
cubo:           262144.000
```

9) pide un entero al usuario y reporta el cociente y el residuo de división con 3.

4.3. Salida de los Datos

Para imprimir los datos en la pantalla, la función "print"
primero convierte cada dato que recibe a una cadena y

concatena (junta) todas estas cadenas en una sola “cadena de impresión”.
Inclusive un dato de tipo cadena se puede modificar para la impresión.

Es posible controlar como una función “print” prepara la “cadena de impresión”.
Este es el tema de la sección presente.

4.3.1. Control del “sep” y del “end”

La función “print” tiene dos parámetros: “sep” y “end”,
cuyos argumentos, o sea valores asignados son
un espacio “ “ y el carácter especial para el fin de línea “\n” respectivamente.
De esta manera el

```
print(0, 1, 2)
```

equivale a

```
print(0, 1, 2, sep=” “, end=”\n”)
```

El parámetro “sep” tiene asignada la cadena que separa los datos al imprimirlos.
El parámetro “end” tiene asignada la cadena que se imprime después del último dato.

A los parámetros “sep” y “end” les podemos reasignar los argumentos a los que deseamos.

Si al sep se le asigna la cadena vacía: sep=””, el “print” no insertará un espacio entre los datos.
Si al end se le asigna la cadena vacía: end=””, el “print” no se moverá a nueva linea;
continuará con la siguiente cadena de impresión donde el último impreso ha terminado.

O sea, el código

```
print(0, 1, 2, sep=””, end=””); print(3, 4, 5)
```

imprimirá en la pantalla:

```
0123 4 5
```

Si tenemos varias declaraciones en misma linea de código, las separamos con el símbolo ;.
En Python preferimos poner solamente una declaración por línea y no usar el símbolo ;.

En el ejemplo anterior, el segundo “print”, regresa automáticamente a sep=” “ y end=”\n”,
ya que nosotros no hemos asignado argumentos a estos parámetros.

Ejecute y entienda los resultados de los siguientes ejemplos, y experimente con algunos más.

Escriba el modulo que contenga el siguiente código.
Guarde lo con el nombre que desee y ejecútelo.

```
print(0, 1, 2, end=’END ’)
print(0, 1, 2, sep=’ | ’, end=’   ||\n’)
print(0, 1, 2, sep=””, end=”|”)
print(“Cantidad: $”, 123, “ pesos.”, sep=””)
```

También puede practicar ejecutando el “print” dentro de una sesión interactiva.

Por ejemplo:

```
>>> 0 + 1 + 2
3
>>> print(0, 1, 2, sep=" + ", end=" = 3 \n")
0 + 1 + 2 = 3
```

Así puede experimentar, interactuando directamente con el intérprete de Python.

Aparte de la separación entre datos "sep" y la parte final "end" de la "cadena de impresión", también se puede controlar el formato de impresión de cada dato pasado a la función "print". Una manera de hacer esto es usando la función "format".
Existe otro sistema de formateo de impresión, más complejo, el cual estudiaremos después.

La función "format" recibe dos argumentos: el dato y el formato para su impresión:

```
format(<dato>, '<formato>')
```

Su uso para varios tipos de datos se explica en continuación.

4.3.2. Formateo de las Cadenas

Formateo de Ancho de Campo y de Posición de la Cadena

Una cadena se puede formatear definiendo el mínimo ancho de campo de impresión, expresado como la cantidad mínima de N símbolos que se tiene que contener.

```
format('Texto', 'Ns')   # N es la cantidad de caracteres.
```

Ejecute lo siguiente y experimente

```
>>> print("|", format("Texto", '3s'), "|", sep="")
|Texto|
>>> print("|", format("Texto", '6s'), "|", sep="")
|Texto |
>>> print("|", format("Texto", '9s'), "|", sep="")
|Texto    |
```

Si la cadena tiene N o más caracteres, va a ser escrita completa tal como es.
Si la cadena tiene menos de N caracteres, el resto de campo de ancho N se llena con espacios, o con el carácter de relleno que puede ser definido distinto de espacios.

Aparte de tamaño mínimo del campo,
también se puede controlar la posición de la cadena dentro del campo usando:

```
'<Ns'     # justificado a la izquierda
'^Ns'     # centrado
'>Ns'     # justificado a la derecha
```

Ejecute lo siguiente y experimente:

```
>>> print("|", format("Texto", '<9s'), "|", sep="")
|Texto    |
>>> print("|", format("Texto", '^9s'), "|", sep="")
|  Texto  |
>>> print("|", format("Texto", '>9s'), "|", sep="")
|    Texto|
```

Aparte de la función "format", se puede controlar la impresión y definir el carácter de relleno con funciones de formateo de las cadenas como sigue:

```
str.ljust(<ancho>, <carácter>)    # justificado a la izquierda
str.center(<ancho>, <carácter>)   # centrado
str.rjust(<ancho>, <carácter>)    # justificado a la derecha

str.zfill(<ancho>)                # rellenar con ceros a la izquierda.
```

El significado del operador punto estudiaremos después.

Ejecute las siguientes declaraciones en modo interactivo, o como un script:

```
print('01234'.ljust(9, '*'))
print('01234'.center(9, '_'))
print('01234'.rjust(9, '_'))

print('01234'.zfill(9))
print('Texto'.zfill(9))
```

Caracteres Especiales en las Cadenas

Cada carácter tiene un código que lo representa en la máquina.
No todos códigos representan caracteres visibles.
Algunos códigos son caracteres que no se ven impresos, pero llevan instrucciones sobre impresión.

Tales caracteres especiales se representan visualmente empezando con el símbolo \.

Los caracteres especiales ("escape sequences") más usados y lo que representan
están dados en la siguiente tabla:

```
CÁRACTER:     REPRESENTA:

\"          # "
\'          # '
\t          # tabulador
\n          # nueva línea

\0          # carácter nulo (no señala el final de la cadena como en C++)
\a          # sonido audible (audible) conocido como "beep"

\b          # regresar un espacio a la izquierda (backspace)
\f          # completar formato, usualmente página (form feed)
\r          # regresar al inicio de la línea (return)
\v          # tabulador vertical (vertical tab)

\N{id}      # identificador de base de unicode
\oOO \oOOO  # O son dígitos octales (2 o 3 dígitos)
\xHH        # H son dígitos hexadecimales (2 dígitos)
\uHHHH      # Unicode hexadecimal 2 bytes (4 dígitos)
\uHHHHHHHH  # Unicode hexadecimal 4 bytes (8 dígitos)
```

Cuando se desea, imprimir literalmente \t en lugar de carácter especial \t,
el cual imprime tabulador, se usa \\t.

También se puede evitar interpretación de símbolo \ como carácter especial
si se separan los caracteres con un espacio: \ t, en caso que no es necesario que estén juntos.

La tercera manera de evitar el uso de caracteres especiales en una cadena,
es usando la cadena rea, "raw string", el cual elimina la interpretación de caracteres especiales.
La cadena rea, no-refinada, se forma con letra r justo antes de la cadena.

Por ejemplo:

```
r"C:\nuevos\texto_3.txt"
```

es lo mismo que

```
"C:\\nuevos\\texto_3.txt"
```

Ambas maneras evitan la interpretación de símbolo \ como carácter especial.

Para entender mejor los caracteres especiales, intenta ejecutar y experimente con:

```
print("  \z  \\z  \  \\  \'  \\'  \"  \\")
print(r"  \z  \\z  \  \\  \'  \\'  \"  \\")
```

donde el \z no es uno de los caracteres especiales.

4.3.3. Formateo de Enteros

Formateo de un entero es similar al caso de formateo de una cadena, solamente que usamos letras "d", "b", "o", "x" para sistemas decimal, binario, octal o hexadecimal, respectivamente.

La declaración

```
print(format(<entero>, 'Nd'))    # N es el número
```

imprime entero en sistema decimal sobre el campo mínimo de N caracteres.

```
>>> print("|", format(12345678, '12d'), "|", sep="")
|    12345678|
```

El uso de los siguientes símbolos modifica la forma de presentación de resultado:

```
'+d'     # imprimir el signo siempre (aunque el número sea positivo).
',d'     # separar cada tres dígitos con comas para facilitar comprensión.


>>> print("|", format(12345678, '+,d'), "|", sep="")
|+12,345,678|
```

Las modificaciones se pueden combinar con el especificación de mínimo ancho de campo.

```
'+Nd'       # imprimir el signo siempre (aunque sea positivo).
'N,d'       # separar cada tres dígitos con comas.
```

```
>>> print("|", format(12345678, '+12,d'), "|", sep="")
| +12,345,678|
>>>
```

Con el ancho de campo especificado se pueden usar otros dos modificadores:

```
'0Nd'       # rellenar con ceros.
'c=Nd'      # rellenar con carácter "c" entre el signo y los dígitos.
    '0=Nd'  # es equivalente a '0Nd', o sea "c" es 0.
    '=Nd'   # es equivalente a ' =Nd', o sea "c" es espacio " ".
```

Ejecute los siguientes ejemplos y verifique si se obtienen los resultados indicados:

```
print(format(12345, '09d'))    # imprime '000012345'
print(format(12345, '+9d'))    # imprime '   +12345'
print(format(12345, '*=9d'))   # imprime '****12345'
print(format(12345, '=9d'))    # imprime '    12345'
print(format(12345, '9,d'))    # imprime '   12,345'
```

Ejecute estos ejemplos que muestran como combinar los símbolos:

```
print(format(12345, '+09d'))    # imprime '+00012345'
print(format(12345, '=+9d'))    # imprime '+   12345'
print(format(12345, '09,d'))    # imprime '0,012,345'
print(format(12345, '+09,d'))   # imprime '+0,012,345'
```

También existen funciones bin(), oct(), hex() para imprimir enteros
en los sistemas binario, octal o hexadecimal respectivamente.

Ejecute:

```
>>> print(12, bin(12), oct(12), hex(12))
12 0b1100 0o14 0xc
```

El resultado quiere decir:

```
12 en decimal es:
0b (en binario)     1100 = 1 x 8 + 1 x 4 + 0 x 2 + 0 x 1
0o (en octal)         14 =                 1 x 8 + 4 x 1
0x (en hexadecimal)    c =                         12 x 1
```

Todo esto se puede combinar con opciones '^', '<', y '>'
para que el número esté centrado o justificado a la izquierda o a la derecha,
como en el caso de controlar la impresión de una cadena.

Ejecute como script, o de manera interactiva:

```
print(format(12345, '<9d'))
print(format(12345, '^9d'))
print(format(12345, '>9d'))
```

4.3.4. Formateo de flotantes

El formateo de punto flotante se logra usando símbolos “e”, “f”, “g”, o “ %”
para formato científico, numero decimal, automático, o porcentaje, respectivamente.
El “automático” quiere decir que se imprima la cadena más corta,
entre las cadenas producidas por formatos “e” y “f”.

Ejecute como script, o de manera interactiva:

```
print(format(0.123456789, 'e'))
print(format(0.123456789, 'f'))
print(format(0.123456789, 'g'))
print(format(0.123456789, '%'))
```

Observe cuantos dígitos se imprimen y si el valor es redondeado, o truncado.

La cantidad de dígitos impresos se puede controlar. La declaración

```
print(format(<número>, 'N.Df'))   # N y D son números
```

imprime número sobre un campo mínimo de N caracteres,
con D dígitos después del punto decimal.

Ejecute como script, o de manera interactiva:

```
print(format(0.123456, '10.6f'))
print(format(0.123456, '8.6f'))
print(format(0.123456, '6.6f'))
print(format(0.123456, '4.6f'))

print(format(0.123456, '10.8f'))
print(format(0.123456, '10.6f'))
print(format(0.123456, '10.4f'))
print(format(0.123456, '10.3f'))

print(format(1234.5, '9f'))
print(format(1234.5, '9.2f'))
print(format(1234.5, '9,.2f'))
```

Este formato se puede combinar con los símbolos “<”, “^” y “>”,
los que ya usamos para modificar la impresión de cadenas y de enteros.

Ejecute el siguiente ejemplo y experimente algunos más:

```
print('|', format(1234.5, '<9.2f'), '|')
```

Existe otro sistema de formateo de impresión, más complejo, el cual estudiaremos después.

4.3.5. Ejercicios

Escriba un programa que ...

1) presenta su nombre, edad, altura en metros, todos en una misma línea, y
entre datos escriba un gato # y al fin de la linea tres gatos: ### y nueva linea.

2) presenta su nombre, edad, altura en metros, un dato por linea,
sobre campo de 25 caracteres y todos los datos alineados a la derecha.

3) presenta las siguientes cadenas, una por línea en la pantalla:

```
What's that? "A sorprise"?
C:\noticias\tramitar\averiguar
```

4) presenta entero 98765 en los siguientes formatos, uno por línea

```
+0,098,765
?????98765
+    98765
0000098765
```

5) presenta número 15 en sistemas: decimal, binario, octal y hexadecimal.

6) presenta número 3.456 en forma científica y en forma decimal.

7) produce los siguientes números flotante, con dos dígitos decimales y alineados a la derecha:

```
23400.00
  876.00
 1230.00
```

4.4. Los Objetos y la Vista al Futuro

La palabra "objeto" es la palabra más usada
en todos los lenguajes de programación "orientados a objetos".
Es todavía más usada en Python que esta hecho de objetos y para trabajar con objetos.

En esta sección de una vez vamos a aclarar el concepto de "objeto".
Vamos a aprender diferentes clasificaciones de objetos y tenerlos aquí como referencias, y
vamos a conocer algunos tipos de objetos que estudiaremos en detalle a lo largo de este libro.

Todos los datos que usamos en Python son objetos; pero objetos son más que solamente datos.
Los objetos son también los archivos, y las componentes de código, como:
expresiones, declaraciones, funciones, clases y módulos.
El intérprete mismo de Python es un objeto.

La Eliminación de Objeto

La **eliminación** de cualquier objeto de la memoria y liberación de esta memoria para otros usos,
se logra usando la declaración "del" ("delete", borrar):

```
del <nombre_de_objeto>
```

Aunque esto es posible, el uso de la declaración "del" en código Python es muy raro, ya que
el proceso llamado "gc" ("garbage collector", colector de basura)
se encarga de recuperar la memoria de cualquier objeto que ya no está en uso,
durante la ejecución del programa y sobre todo cuando el programa termine.

4.4.1. Identidad, Tipo y Valor

Cada objeto tiene su **identidad**, su **tipo** y su **valor**.

La **identidad** de un objeto se puede entender como el número de **domicilio en la memoria**, donde inicia la lectura de esta objeto. La identidad de un objeto nunca cambia durante su vida. La función "id" nos proporciona ese número identidad.

```
id(<objeto>)
```

La identidad de un objeto nunca cambia, aunque este objeto puede ser movido
de una posición de la memoria RAM a la otra, o de la memoria RAM a la memoria cache y atrás. Por esto se dice que la identidad "no es exactamente", pero "se puede entender" como el domicilio.

El **tipo** de objeto determina su codificación, o sea su **representación binaria** en la computadora y
determina las operaciones aplicables a este objeto. El tipo de un objeto no cambia durante su vida. La función "type" nos dice cual es el tipo de un objeto.

```
type(<objeto>)
```

El **valor** de un objeto es lo que es guardado en la memoria en el domicilio del objeto:
es una cantidad para escalares, o una secuencia de identidades para las colecciones.

Algunos tipos de objetos tienen su valor modificable y
en otros objetos no cambian su valor durante su vida.

Esto produce varias divisiones de objetos, lo cuál aclaramos en continuación.

4.4.2. Los Objetos Dinámicos, Mutables y Mapeables ("Hashables")

Los Tipos de Objetos Dinámicos

Con respecto a la cantidad de memoria que usan los objetos durante su vida,
los dividimos en dos tipos:

Estáticos usan la constante cantidad de memoria durante su vida.

Dinámicos cambian la cantidad de memoria que necesitan durante su vida.

Con respecto a su dinámica, los objetos integrados se dividen en los siguientes grupos:

```
None                            # Genérico es estático
bool, int, float, complex       # Escalares son estáticos
str, tuple, bytes, frozenset    # Colecciones estáticas
list, bytearray, set, dict      # Colecciones dinámicas
```

Tal vez confunde que una colección como cadena "str" puede ser estática.
Lo que pasa es si una variable referenciando una cadena, le intentamos de cambiar el valor, agregando, o eliminando, caracteres a la cadena, no se modifica la misma cadena, sino
se produce una cadena nueva en otra parte de memoria y la variable se reasigna a la nueva cadena. De esta manera los objetos estáticos no cambian de tamaño en la memoria donde están asignados.

Los Tipos de Objetos Mutables

Los objetos se dividen en dos grupos con respeto a posible cambio de valor durante su vida:

Inmutables No pueden cambiar su valor de manera directa durante su vida.

Mutables pueden cambiar su valor directamente durante su vida.

La inmutabilidad de los objetos integrados está relacionada con su tipo:

```
None                            # Genérico es inmutable
bool, int, float, complex       # Escalares son inmutables
str, tuple, bytes, frozenset    # Colecciones inmutables
list, bytearray, set, dict      # Colecciones mutables
```

Como podemos ver, entre los tipos de objetos integrados,
los tipos inmutables son estáticos y los tipos mutables son dinámicos.

Existen ciertas sutilezas en la relación entre el valor y la mutabilidad.
Una colección inmutable, como tupla "tuple" puede contener un elemento mutable, como lista "list".
El contenido de la tupla son las referencias a sus elementos, sus ítems,
entre cuales está la referencia a la mencionada lista. La "referencia" es la identidad de la lista.

Siendo esta lista objeto mutable, es posible que lo modifiquemos. Esto no cambia su identidad.
O sea, la tupla que contiene las identidades de sus elementos no ha cambiado, es inmutable.
Pero por otro lado, tenemos que decir que el valor de la tupla ha cambiado, porque
el valor de uno de sus elementos ha cambiado. Así que, un objeto inmutable puede cambiar valor.
No lo puede cambiar directamente, pero lo puede cambiar indirectamente.

Los dos párrafos anteriores se van a aclarar cuando estudiaremos las tuplas y veamos los ejemplos;
aquí quedan como una referencia para cuando regresen a revisar el concepto "objeto".

Lo anterior introduce otra importante división de los objetos:
los objetos mapeables ("hashables") y los objetos no-mapeables.

Los Tipos de Objetos Mapeables ("hashables")

Una de las acciones frecuentes del intérprete de Python es mapear objetos a un número entero.
Se aplica un mapeo, una función, "hash-function" la cuál usa el valor de un objeto,
y a base de este calcula un número entero "hash-code" que le corresponde.
Este entero se usa después para comparar e identificar los objetos.

El mapeo funciona solamente con los objetos que no cambian su valor durante su vida.
Esto produce una división de objetos en dos tipos:

Mapeables los que no pueden cambiar su valor, directamente ni indirectamente, durante su vida.

No-Mapeables los que pueden cambiar su valor, directamente o indirectamente, durante su vida.

Los objetos integrados de tipo **mapeable** son:
los escalares: lógicos, enteros, flotantes y complejos, ya que son inmutables y
las colecciones inmutables mapeables: cadenas, tuplas mapeables y conjunto congelado.

```
None                            # Genérico    NO-mapeable
bool, int, float, complex       # Escalares   mapeables
```

```
str, bytes, frozenset             # Colecciones mapeables
tuple                             # Colección
list, bytearray, set, dict        # Colecciones NO-mapeables
```

Como podemos ver, la división de tipos de objetos a mapeables y no-mapeables es casi idéntica a la división a los inmutables y mutables. Solamente hay dos excepciones: None y tupla.

El None, aunque es inmutable, no es mapeable ya que no contiene ningún valor.

La mapeabilidad de la tupla depende de sus ítems:
Si todos sus ítems son mapeables, entonces la tupla misma también es mapeable.
Sí al menos uno de sus ítems no es mapeable, entonces la tupla misma no es mapeable.

4.4.3. Objeto y Sus Atributos

Cuando un objeto dado es una colección y contiene otros objetos como sus elementos, o ítems, a estos objetos contenidos los llamamos los "atributos" del objeto dado. Vamos a aclarar esto.

Objetos en la Naturaleza

Módulos como Objetos

Funciones como Objetos

Clases Como Objetos y Moldes para Objetos

4.5. Librería Python

4.5.1. El Modulo "sys", Sistema e Intérprete

En Programación

Para terminar nuestro primer encuentro con el módulo "sys", vamos a ver que nos ofrecen la variable "sys.argv" y la función "sys.exit()". Estas dos son útiles en los scripts.

Guarden en su computadora el modulo "sys_argv_exit.py" con el contenido:

```
import sys

print("El vector de argumentos contiene:")
print(sys.argv)
print("Esto es todo.")
sys.exit()
print("Esta línea no se va a ejecutar.")
```

Ejecuten este programa desde la linea de comandos usando:

```
python3 sys_argv_exit.py datos.txt resultados.txt
```

En su computadora el prompt del sistema operativo se puede ver diferente:

```
$ python3 sys_argv_exit.py datos.txt resultados.txt
El vector de argumentos contiene:
['sys_argv_exit.py', 'datos.txt', 'resultados.txt']
Esto es todo.
$
```

El "sys.argv", donde "argv" es abreviado de "argument vector," es una variable que contiene los argumentos que se teclearon después de "python3" en la línea de comandos.

El primer argumento es el nombre del modulo.
Los siguientes dos argumentos por momento no los usamos más que imprimirlos, pero en futuro los usaremos para nombres de archivos de datos de entrada y de salida.

La función "sys.exit()" provoca que el el sistema operativo termine el proceso que está ejecutando nuestro script. Ninguna declaración después de esta ya no se va a ejecutar.

Esta sección sobre "sys" es solamente informativa. Lo que aprendimos de "sys" aquí, lo vamos a repasar y usar más adelante en capítulos avanzados.

4.5.2. El Modulo "math", Funciones Matemáticas

Contenido del Modulo "math"

El modulo "math" nos ofrece constantes y funciones para procesar los datos tipo entero y flotante.
https://docs.python.org/3/library/math.html

Las funciones para procesar los números complejos están en el modulo "cmath".
https://docs.python.org/3.5/library/cmath.html

Funciones en el Modulo "math"

Las únicas funciones en el modulo "math" que devuelven resultado de tipo entero son:

```
# FUNCIÓN    # RESULTADO
ceil(x)      # El entero más pequeño, que es mayor o igual a x.
floor(x)     # El entero más grande, que es menor o igual a x.
trunc(x)     # El entero de x truncado, o sea redondeado hacia 0.
```

Abandonaremos el modulo "math" por momento.
Para completar los cuatro tipos de redondeo, el intérprete tiene la función

```
round(x, D)
```

la cual redondea un número x de tipo flotante a D dígitos después del punto decimal.
Para D = 0, la función redondea x al entero más cercano y lo regresa como tipo flotante.
Si se desea resultado como tipo entero, hay que usar "int", la función de conversión:

```
entero = int(round(x, 0))
```

Regresemos al modulo "math".
La función para la raíz cuadrada principal es:

```
# FUNCIÓN    # RESULTADO
sqrt(x)      # La raíz cuadrada positiva de x.
```

Las funciones logarítmicas y exponenciales son:

```
# FUNCIÓN    # RESULTADO
log10(x)     # Logaritmo base 10 de x.
log2(x)      # Logaritmo base  2 de x.
log(x, b)    # Logaritmo base  b de x, donde b es cualquier flotante.
log(x)       # Logaritmo base  e de x
exp(x)       # Base e al exponente x.
```

Las funciones para conversiones de unidades de ángulo son:

```
# FUNCIÓN    # RESULTADO
radians(x)   # Conversión de x grados a radianes
degrees(x)   # Conversión de x radianes a grados
```

Las funciones trigonométricas y trigonométricas inversas son:

```
# FUNCIÓN    # RESULTADO
sin(x)       # Seno de ángulo x medido en radianes.
cos(x)       # Coseno de ángulo x medido en radianes.
tan(x)       # Tangente de ángulo x medido en radianes.

asin(x)      # Arco seno de x, reportado en radianes.
acos(x)      # Arco coseno de x, reportado en radianes.
atan(x)      # Arco tangente de x, reportado en radianes.
atan2(y, x)  # atan(y / x), con el ángulo en el cuadrante correcto.
```

Hay más funciones en el modulo "math":
hiperbólicas, especiales, para trabajo con triángulos rectángulos y otras.
Las mencionadas son suficientes para nuestros propósitos en este texto.

Constantes en el Modulo "math"

El modulo "math" nos ofrece varias constantes matemáticas como son:
el número "math.e": la base de logaritmo natural, y
el número "math.pi": la razón entre la circunferencia y el diámetro de cualquier círculo.

El siguiente ejemplo imprime estas dos constantes.

```
import math

print(m.e, "El número \"e\", de Euler.")
print(m.pi, "El número \"pi\".")
```

Uso de Modulo "math"

Ejecute el siguiente ejemplo de uso de la constante "math.pi".

```
import math as m

# Ejemplo de uso de la constante "pi" en la geometría de círculo
radio_m = 3.0
print("Radio de círculo:", format(radio_m, 'E'), "metros.")

circunferencia_m = 2 * m.pi * radio_m
print("Circunferencia:", format(circunferencia_m, 'E'), "metros.")

área_m2 = m.pi * radio_m ** 2
print("Área:", format(área_m2, 'E'), "metros cuadrados.")
```

Al importar el modulo "math" se introdujo un nombre sustituto más corto: "m".

```
import math as m
```

Después de esto, la constante "math.pi" se usa como "m.pi".

En las variables "radio_m", "circunferencia_m" y "área_m2",
la letra "m" se refiere a metros y "m2" se refiere a metros cuadrados.

4.5.3. Modulo "stats"

4.5.4. Ejercicios

Escriba un programa que ...

1) permite ver su versión de Python en lo que imprime.

2) permite ver si su intérprete es CPython en lo que imprime.

3) permite ver cuál es su sistema sobre cual se ejecuta el intérprete.

4) permite ver cuales son las características de dato tipo entero en su sistema.

5) permite ver cuales son las características de dato tipo flotante en su sistema.

6) recibe argumentos de la línea de comandos y los escribe todos en la pantalla.

Escriba un programa que ...

1) pida un número [-10.0, 10.0] al usuario y reporte cuatro números ENTEROS,
resultados de las funciones: floor, trunc, round, ceil, en este orden.

2) pida un número x de intervalo [1.0, 10.0] y calcule cuatro resultados:
logaritmo natural de x, logaritmo base 2 de x, 2^x y e^x.

3) pida un ángulo a, entero entre 1 y 89 grados, reporte la medida de ángulo en radianes y
los resultados de las seis funciones: seno, coseno, tangente, cotangente, cosecante y secante.

4) pida la coordenada x y la coordenada y de un punto (x, y) en el plano cartesiano y
reporte el ángulo en grados y redondeado a tres decimales, donde el ángulo está
entre el vector que va desde origen (0, 0) al punto (x, y), y el lado positivo de la eje x.
La respuesta debe ser en el intervalo [0.000, 359.999].

4.6. Recomendaciones y Ejemplos

Vamos a presentar algunos ejemplos de código Python y practicar ejercicios.

Algunos consejos antes de empezar.

Interrupción de Un Programa Activo

Si necesita interrumpir un programa, puede hacerlo desde teclado:
En sistemas basados en "Microsoft Windows" pueden interrumpir programas con: [Ctrl][z],
En sistemas basados en "Unix", como son "Mac" y "Linux" el [Ctrl][c] termina el programa,
mientras el [Ctrl][z] solamente suspende el programa, pero no lo termina.

Al mantener la tecla "control", [Ctrl], oprimida y oprimir la tecla "d": [Ctrl][d], para un programa significa que no hay más datos que esperar en la entrada y que continúe.
Esta combinación [Ctrl][d] también termina la sesión interactiva.

Errores Más Frecuentes

Los errores más frecuentes al principio es olvidar a cerrar unas comillas, o un paréntesis.
Los van a identificar por recibir uno de los siguientes excepciones por parte de "Traceback":

```
SyntaxError: EOL while scanning string literal

SyntaxError: unexpected EOF while parsing
```

donde EOL es "End Of Line" y EOF es "End Of File".

4.6.1. Recomendaciones PEP8

Se recomienda instalar y usar el programa pep8 para verificar
si su código esté conforme con el PEP8 ("Python Enhancement Proposal 0008").

En continuación se dan algunas de las recomendaciones PEP8 para respetarlas en los ejercicios.

Corte de Una línea Larga

Si una línea de código es más larga de 79 caracteres visibles,
es posible continuar la misma declaración en la línea que sigue.

El intérprete sabe que la expresión no termina hasta que el paréntesis este cerrado,
así que busca la continuación de la expresión en la siguiente línea.

```
print(dato_1, dato_2, dato_3, dato_4,
      dato_5, dato_6)
```

La continuación está alineada con el paréntesis abierto en la primera línea.

La primera línea de una declaración, si no es completa, debe terminar con el operador y la siguiente línea de la misma declaración empieza con variable o literal.

```
NO:
print(dato_1, dato_2, dato_3, dato_4
      , dato_5, dato_6)
```

Comentarios

Comentario no debe decir lo obvio. No debe decir "que" se hace, sino "por que" se hace.

```
SI:                                 NO:
# Compensando el margen             # Incrementando x
x = x + 3                           x = x + 3
```

```
# Comentario que contradice el código es peor que no tener comentario.
# Modifiquen comentarios inmediatamente al modificar el código comentado.

# Comentarios deben ser escritos usando oraciones completas y entendibles.
# Cada comentario y oración deben empezar con mayúscula,
# excepto si la primera palabra es un identificador usado en el código comentado.
# Los identificadores hay que escribir igual como están en el código.

# En comentarios cortos de una línea, la puntuación al final puede ser omitida

# Comentario de múltiple línea debe ser escrito usando oraciones completas. Las oraciones
# deben terminar con signos de puntuación. Hay que usar un espacio entre las oraciones.
#
# El texto de comentario debe empezar al menos un espacio después del símbolo #
```

Comentario "en línea" es comentario en la misma línea y a la derecha de una declaración.
Comentarios "en línea" no se recomiendan. En caso de usar el comentario "en línea",
el signo # del comentario debe estar al menos dos espacios después de la declaración.

```
SI:                                 EVITAR:
# Compensando el margen             x = x + 3   # Compensando el margen
x = x + 3
                                    NO:
                                    x = x + 3 # Compensando el margen
```

Vamos a presentar algunos ejemplos de código Python.

Algunos consejos antes de empezar.

Interrupción de Un Programa Activo

Si necesita interrumpir un programa, puede hacerlo desde teclado:
En sistemas basados en "Microsoft Windows" pueden interrumpir programas con: [Ctrl][z],

En sistemas basados en "Unix", como son "Mac" y "Linux" el [Ctrl][c] termina el programa, mientras el [Ctrl][z] solamente suspende el programa, pero no lo termina.

Al mantener la tecla "control", [Ctrl], oprimida y oprimir la tecla "d": [Ctrl][d], para un programa significa que no hay más datos que esperar en la entrada y que continúe.
Esta combinación [Ctrl][d] también termina la sesión interactiva.

4.6.2. Recomendaciones PEP8

Se recomienda instalar y usar el programa pep8 para verificar
si su código esté conforme con el PEP8 ("Python Enhancement Proposal 0008").

En continuación se dan algunas de las recomendaciones PEP8 para respetarlas en los ejercicios.

Corte de Una línea Larga

Si una línea de código es más larga de 79 caracteres visibles,
es posible continuar la misma declaración en la línea que sigue.

La primera opción, anticuada, es terminar la línea con \ y continuar en la siguiente.
Para que los programadores humanos identifiquen que se trata de una continuación,
la siguiente línea con sangría de 8 espacios (dos tabuladores).

```
resultado = número_1 + número_2 + número_3 + \
        número_4 + número_5
```

La segunda opción, la recomendada, es poner toda la expresión en paréntesis.
El intérprete sabe que la expresión no termina hasta que el paréntesis este cerrado,
así que busca la continuación de la expresión en la siguiente línea.

```
resultado = (número_1 + número_2 + número_3 +
        número_4 + número_5)
```

La continuación tiene una sangría de 8 espacios, (2 tabuladores), o
alineada con el paréntesis abierto en la primera línea:

```
resultado = (número_1 + número_2 + número_3 +
             número_4 + número_5)
```

El PEP8 recomienda alinear con el paréntesis abierto, como en este último ejemplo, pero
si el paréntesis abierto está demasiado a la derecha, pueden usar 8 espacios (2 tabuladores).

La primera línea de una declaración, si no es completa, debe terminar con el operador y
la siguiente línea de la misma declaración empieza con literal, nombre de variable, o función, o
nombre de cualquier objeto que sigue en la expresión.

```
NO:
resultado = (número_1 + número_2 + número_3
             + número_4 + número_5)
```

Identificadores (Nombres)

En código Python usamos tres estilos para formar los identificadores (nombres):

```
minúsculas_y_subrayado_1    # variable, función (método), modulo
MAYÚSCULAS_Y_SUBRAYADO_2    # constante
EstiloDeCamello             # clase (excepción),  paquete
```

En cada una de estas se pueden usar los dígitos 0 a 9 también como palabra separada por _.

¡No hay que distorsionar las palabras!

Por ejemplo, si desean usar nombre "print" el problema es que el "print" ya es usado.
El conflicto se evita agregando un subrayado: "print_", no recortando la palabra: "prnt".
La lectura de código para programadores y usuarios debe ser fluyente.

```
SI:                             NO:
print_                          prnt
```

¡NUNCA use identificadores que empiezan y terminan con doble subrayado: __nombre__!
Este formato lo usa el lenguaje Python para nombres con significado especial.

Identificadores empezando por un subrayado "_nombre" o dos subrayados "__nombre",
usaremos sólo para propósitos específicos que aprenderemos más adelante. No los use por ahora.

```
SI:                             USOS ESPECÍFICOS:       NO:
nombre                          _nombre                 __nombre__
nombre_                         __nombre
nombre_de_varias_palabras
nombre_de_varias_palabras_
```

Las abreviaciones se escriben en mayúsculas.

```
SI:                             NO:
ErrorDeServidorHTTP             ErrorDeServidorHttp
código_ASCII                    código_ascii
formato_UTF8                    formato_Utf8
```

NO USE letras: O (o mayúscula), I (i mayúscula), l (L minúscula), como identificadores; parecen a 0 y 1.

Casos para NO Usar Espacios

NO USE ESPACIOS, " ", ni tabuladores, "\t", al final de una línea, ni dentro de una línea "vacía".

NO USE ESPACIOS por dentro de paréntesis, corchetes y llaves al iniciarlos o cerrarlos.

```
SI:                             NO:
función(a[0], {llave: valor})   función( a[ 0 ], { llave: valor } )
```

NO USE ESPACIOS inmediatamente antes de la coma, punto y coma y dos puntos:

```
SI:                                 NO:
y, x = x, y                         y , x = x ,y
a = 2; b = 3                        a = 2 ; b = 3
if a == 2 and b == 3:               if a == 2 and b == 3 :
    print(x, y)                         print(x , y)
```

NO USE ESPACIOS entre el nombre de función y el paréntesis,
ni alrededor de parámetro con argumento asignado.

```
SI:                                 NO:
print(a, b, sep="", end="|\n")      print (a, b, sep = "", end = "|\n")
función(parámetro=argumento)        función (parámetro = argumento)
```

NO USE ESPACIOS justo antes de operador de índice, o sub-cadena (sub-lista),
ni alrededor de colon en el operador de sub-cadena (sub-lista).

```
SI:                                 NO:
cadena[-1]                          cadena [-1]
lista[1:-2]                         lista [1 : -2]
```

Casos para SI Usar Espacios

USE EXACTAMENTE UN ESPACIO de cada lado de operador de asignación =,
con excepción de caso "parámetro=argumento" mencionado anteriormente.

```
SI:                                 NO:
x = 1                               x   = 1
y = 2                               y   = 2
val = 3                             val = 3
función(parámetro=argumento)        función (parámetro = argumento)
```

SIEMPRE USE UN ESPACIO de cada lado de cada operador binario.

```
SI:                                     NO:
i = i + 1                               i = i+1
contador += 1                           contador +=1
y += 2 * x + 1                          y+= 2*x + 1
hipotenusa = (a * a + b * b) ** 0.5     hipotenusa = (a*a + b*b)**0.5
```

DEJE LÍNEA VACÍA al final del archivo, que no contenga espacios ni tabuladores.
O sea, el archivo termina con el carácter nueva línea: "\n".

4.6.3. Ejemplos

01) Escriba un programa que le pide a usuario su primer nombre y después su apellido paterno y presenta en la pantalla: apellido paterno, espacio y primer nombre.

4.7. Tareas

Escriba un programa que ...

1) pida un texto y lo imprime al revés.

2) reciba precios entre \$1.00 y \$99,999.00 de tres artículos y
los presenta en la pantalla junto con el total, con dos dígitos decimales y alineados verticalmente.
En la pantalla se debe producir el formato siguiente:

```
Este programa calcula el total de tres precios.

Proporcione el precio 1 (1 a 99,999): 23400
Proporcione el precio 2 (1 a 99,999): 876
Proporcione el precio 3 (1 a 99,999): 1230
    23400.00
      876.00
     1230.00
------------
    25506.00
```

3) obtiene un precio del usuario y en la pantalla presenta el precio con descuento de 15 % y
la cantidad de ahorro, con dos dígitos decimales y alineados verticalmente así:

```
Este programa calcula el precio con descuento 15% y su ahorro.

¿Cuál es el precio regular (1 a 9,999)? 200
 $170.00  Precio con descuento
  $30.00  Su ahorro
```

4) haga lo siguiente. Editorial vende libros a las librerías con 40 % de descuento.
El costo de envío es \$10 para el primer libro más \$1 por cada libro adicional.
Escriba un programa que reciba precios de tres libros y calcule
el precio total (libros y envío) de compra.

5) pida el radio de esfera y calcule su superficie y volumen:

$$s = 4 \cdot \pi \cdot r^2 \qquad v = \frac{4}{3} \cdot \pi \cdot r^3.$$

6) pida dos números positivos y calcule las tres medias:
media aritmética, media geométrica y media harmónica.

$$ma = \frac{x+y}{2} \qquad mg = \sqrt{x \cdot y} \qquad mh = \left(\frac{1}{x} + \frac{1}{y}\right)^{-1}.$$

7) use la constante de gravedad G = -9.806 m/s^2,
pida el usuario altura inicial y_0, velocidad vertical inicial v_0 y tiempo de caída t, y
después calcule la altura final y velocidad final:

$$y = \frac{1}{2} \cdot G \cdot t^2 + v_0 \cdot t + y_0 \qquad v = G \cdot t + v_0.$$

8) pida un entero al usuario y reporta cuantos dígitos tiene el entero en sistema decimal y
cuantos en el sistema binario.

5. No Hay Vida Sin las Matemáticas

5.1. Lógica

5.2. Conjuntos

5.3. Flotantes

Parte II.

Programación Estructurada

6. Programación Estructurada

La programación estructurada es un paradigma de programación,
cuyo objetivo es claridad, calidad y rapidez de desarrollo de programas, y
el cual está basado en uso de los siguientes elementos:

- bloques de declaraciones de ejecución secuencial (una tas otra en orden escrito),
- ejecución condicionada de cierta declaración o bloque (ramificación),
- ciclos de ejecución que permiten repetir cierta declaración o bloque varias veces y
- funciones que se definen una vez y se ejecutan las veces que sea necesario.

Este capitulo está dedicado al estudio de los primeros tres de estos elementos
Las funciones se explican en el siguiente capitulo.

6.1. Objetos Lógicos y Expresiones Lógicas

Una expresión lógica es aquella que evaluada resulta en un objeto de tipo lógico (boolean).

Un objeto de tipo lógico es el que tiene uno de dos posibles valores:
0, el cual es usado en código como False (falso) y convertido a cadena como "False".
1, el cual es usado en código como True (cierto) y convertido a cadena como "True".

6.1.1. Conversión a Tipo Lógico

En Python, cualquier tipo de datos se puede convertir en tipo lógico usando la función "bool".

Para los tipos de datos básicos: genérico, entero, punto flotante y cadena, que ya conocemos,
aplican las siguientes reglas de conversión a tipo lógico:

bool(genérico) resulta en False.

```
bool(None)   # resulta en False
```

bool(entero) resulta en False si entero es 0, y en True para cualquier otro valor.

```
bool(0)      # resulta en False
bool(1)      # resulta en True
bool(-2)     # resulta en True
```

bool(punto_flotante) resulta en False si el punto_flotante es 0.0, y en True para cualquier otro valor.

```
bool(0.0)         # resulta en False
bool(3.141592653589793)     # resulta en True
bool(1e-307)      # resulta en True
```

bool(cadena) resulta en False, si la cadena es vacía, "", y en True para cualquier otro valor.

```
bool("")     # resulta en False
bool("0")    # resulta en True
bool(" ")    # resulta en True
```

6.1.2. Operadores de Comparación

Los operadores de comparación: <, <=, ==, !=, >=, >, is,
se usan para formar expresiones lógicas.

Las Seis Comparaciones: <, <=, ==, !=, >=, >

Dada como el punto de referencia una constante A, y la variable x,
podemos formar seis expresiones lógicas, usando las seis comparaciones:

```
x < A,   x <= A,   x == A,   x != A,   x >= A,   x > A
```

Dadas como los puntos de referencia dos constantes A y B tales que A < B, y la variable x,
podemos formar cuatro expresiones de comparación de ambos lados:

```
A < x < B,   A <= x <= B,   A <= x < B,    A < x <= B,
```

Cada expresión se puede escribir también al revés. Por ejemplo:
x < A es lo mismo que A > x.
A < x < B es lo mismo que B > x > A.

Los operandos A, B, de comparación pueden ser: constantes, variables, expresiones, ...
Por ejemplo:

```
y <= (2 * x + 1)
y > función(x)
```

En resumen, un operando puede ser cualquier expresión que produce un objeto comparable.

Ejecute estas declaraciones en modo interactivo e invente algunas más:

```
x = 3              # No es una expresión lógica, es una asignación.

print(x < 4)
print(x == 4)
print(x != 4)
print(4 < x)
print(2 < x < 4)
```

El Operador de Comparación "is"

Los objetos comparables se pueden comparar de dos maneras:

x == y
es True (cierto) si estas dos variables referencian objetos que tienen
el mismo valor, o contenido. Por ejemplo, el mismo número, o la misma cadena.

x is y
es True (cierto) si estas dos variables referencian objetos que tienen
el mismo domicilio en la memoria. O sea, si referencian el mismo objeto.

Si "x is y" es cierto, entonces "x == y" es también cierto, ya que se trata del mismo objeto.
Si "x == y" es cierto, no necesariamente es cierto "x is y", aunque puede que lo sea.

La función "id" nos dice cual es el domicilio de la memoria
donde empieza la lectura del objeto referenciado,
o sea, nos dice cual es el número identificador asociado a la variable
en la tabla de nombres ("namespace").

```
x is y          # es equivalente a  id(x) == id(y)
```

El otro uso del operador "is" es para comparar un objeto con constantes: None, False, y True.
No hay que comparar una variable con estas constantes usando las seis operadores de comparación:
<, <=, ==, !=, >=, >.

```
SI:                             NO:
x is None                       x == None
x is False                      x == False
x is True                       x == True
```

La Memoria RAM y la Memoria Cache

El intérprete optimiza el uso de la memoria, de manera que algunos de objetos inmutables:
genérico (None), lógicos (False, True), enteros, punto flotante y cadenas cortas
están guardados, si se puede, directamente en la memoria más rápida, la memoria "cache".

Durante la ejecución de un programa suceden intercambios de bloques de objetos,
entre la memoria RAM y la memoria cache.

Al asignar, por ejemplo x = 3, puede que el objeto "entero 3" queda en el RAM, o en el cache.
Y sin importar donde queda inicialmente, pronto puede ser intercambiado.

Si después sigue otra asignación al mismo valor, por ejemplo y = 3,
y el objeto 3, ya guardado con el nombre x está en el cache,
entonces, el nombre "y" referenciará el mismo entero 3.

El intérprete no revisa la memoria RAM para encontrar objetos ya existentes al asignar uno nuevo;
solamente revisa la parte de la memoria cache que le pertenece.

Así que, no se sorprenda si obtenga el siguiente resultado:

```
x = 3
y = 3
print(y is x)    # imprime False
z = 3
print(z is x)    # imprime True
```

Podemos intencionalmente lograr que dos variables tengan referencia el mismo objeto.
Esto se logra asignando una variable a la otra:

```
x = 3
y = x
print(y is x)   # imprime True
```

Esto funciona sin importar si el entero 3 esté en la memoria RAM, o en el cache.

El hecho de que tanto "x" como "y" referencian el mismo objeto inmutable no es problema, ya que al reasignar cualquiera, digamos "y = 4", el nombre "y" va a cambiar su referencia a algún otro domicilio de la memoria donde va a referenciar el objeto "entero 4", sin afectar el domicilio anterior donde el nombre "x" todavía sigue referenciando el "entero 3".

La importancia de la función "id" y del operador "is" se va a notar cuando empecemos trabajar con objetos mutables. Por ahora, todos nuestros objetos son inmutables.

6.1.3. El Operador de Inclusión "in"

El operador "in" compara un elemento, o una sub-colección, con una colección de elementos y determina si este elemento, o está sub-colección, esté incluida en la colección.

Por ejemplo, cadena es uno de tipo de datos básicos, pero al mismo tiempo una cadena no vacía es una colección de caracteres y una colección de sub-cadenas.

Así que, podemos usar el operador "in" para saber si un carácter, o una cadena particular, está dentro de una cadena dada.

Ejecute estas declaraciones en modo interactivo e invente algunas más:

```
a = "El texto de cadena."

print("e" in a)
print("T" in a)
print("." in a)
print(" " in a)
print('texto' in a)
print("Cadena" in a)
```

6.1.4. Los Operadores Lógicos "not", "and", "or"

Los operadores de comparación se pueden combinar usando los operadores lógicos: "not", "and", "or", para formar expresiones lógicas más complejas.

Funcionamiento de los Operadores Lógicos

Sean L1 y L2 valores lógicos: cada uno es True, o False. De momento no sabemos sus valores.

La expresión lógica: "not L1"
es cierta cuando L1 es falso y es falsa cuando L1 es cierto.

La expresión lógica: "L1 and L2"
es cierta solamente cuando ambas L1 y L2 son ciertas, de otra manera es falsa.

La expresión lógica: "L1 or L2"
es falsa solamente cuando ambas L1 y L2 son falsas, de otra manera es cierta.

Ejecute las siguientes declaraciones en modo interactivo e invente algunas más:

```
x = 3                   # No es una expresión lógica, es una asignación.

print(not (x < 4))      # equivalente a print(x >= 4)
print(not (x <= 4))     # equivalente a print(x > 4)
print(x > 2 and x < 4)  # equivalente a print(2 < x < 4)
print(2 < x and x > 4)  # equivalente a print(x > 4)
print(x <= 3 or 5 <= x) # equivalente a print(not (3 < x < 5))
print(x < 3 or 3 < x)   # equivalente a print(x != 3)

x = 3
y = 3
print(x == y);   print(x != y)
print(x is y);   print(x is not y)
z = x
print(x == z);   print(x != z)
print(x is z);   print(x is not z)
```

Optimización de Los Operadores Lógicos "and" y "or" (Shortcutting)

Sean L1 y L2 valores lógicos: cada uno es True, o False. De momento no sabemos sus valores.

La expresión lógica: "L1 and L2"
es cierta solamente cuando ambas L1 y L2 son ciertas, de otra manera es falsa.

En la expresión

```
L1 and L2
```

si L1 es falso, ya no se evalúa el L2; ya se sabe que la expresión completa "L1 and L2" es False.

La expresión lógica: "L1 or L2"
es falsa solamente cuando ambas L1 y L2 son falsas, de otra manera es cierta.

En la expresión

```
L1 or L2
```

si L1 es cierto, ya no se evalúa el L2; ya se sabe que la expresión completa "L1 or L2" es True.

Lo mismo sucede con los operadores repetidos.

La expresión

```
L1 and L2 and L3 and L4
```

se evalúa a False con primer valor L que se encuentre falso, o
se evalúa a True cuando todos los valores L estén evaluados a True.

La expresión

```
L1 or L2 or L3 or L4
```

se evalúa a True con el primer valor L que se encuentre cierto, o
se evalúa a False cuando todos los valores L estén evaluados a False.

6.1.5. Comparación de Dos Cadenas

Cadena es una secuencia de caracteres y cada carácter tiene su representación numérica.
Python guarda los caracteres en la memoria codificados en el formato UTF-8.
Los primeros códigos 0 hasta 256 de UTF-8 representan caracteres idénticos a los caracteres ASCII.
La diferencia es que cada carácter en ASCII usa solamente un byte.
Así, el sistema ASCII es limitado a representar solamente 256 caracteres.
El UTF-8 no tiene tamaño de caracteres fijo, puede usar
un byte para cierto carácter y hasta cuatro bytes para un otro carácter.

Con la función "ord" podemos obtener el número ordinal del carácter:

```
>>> ord('A')
65
>>> ord('Z')
90 >>>
ord('a')
97
```

Con la función "chr" podemos obtener el carácter que corresponde a un código dado:

```
>>> chr(91)
'[' >>> chr(92)
'\\'
>>> chr(96)
'`'
```

Para comparar dos caracteres, o dos cadenas, se puede usar cualquiera de los seis
operadores de comparación: $<$, $<=$, $==$, $!=$, $>=$, $>$.

Cuando comparamos dos caracteres, se comparan sus representaciones numéricas.
El menor carácter es aquel cuya representación numérica es menor.

Cuando se comparan dos cadenas, si una de ellas es la cadena vacía: "", y la otro no,
la cadena vacía es el menor.

Cuando se comparan dos cadenas no vacías, primero se comparan sus caracteres iniciales.
Si estos tienen números ordinales distintos, entonces el menor carácter determina la menor cadena.
Si estos tienen números ordinales iguales, se compara el siguiente carácter de cada cadena.
Cuando se llegue a caracteres distintos, el menor carácter determina la cadena menor.
Si en este proceso, una cadena termina antes, esta es la cadena menor.
Si las cadenas tienen los mismos caracteres y son igual de largos, son iguales.

```
>>> "" < " "
True
>>> "ab" < "abc"
True
```

```
>>> "abc" < "abd"
True
>>> "abcde" == "abcde"
True
>>> "Beta" < "alpha"
True
```

Tenga presente que todas letras mayúsculas son menores que cualquier letra minúscula.

6.1.6. Comparación de Dos Números

Los seis de los operadores de comparación: <, <=, ==, !=, >=, >,
se pueden usar para comparar objetos enteros sin problemas.

Al comparar los números flotantes, lo único que cabe enfatizar es que
no se deben de usar los operadores ==, y !=.

La razón es que en la máquina los números flotante se representan con al cantidad finita de bits.

Solamente algunas fracciones es posible traducir exactamente a flotantes. Tales fracciones son:
un medio, un cuarto, un octavo, un dieciseisavo, y así hasta cierta profundidad, y
también las sumas y restas de tales fracciones.

Las demás fracciones pueden necesitar infinita cantidad de bits, para representarse exactamente.
Algunas fracciones como 1 / 3 hasta en el sistema decimal requieren infinita cantidad de dígitos.

Por lo tanto, no siempre existe la representación exacta para los números decimales
en la representación flotante, ya que flotantes tienen limitada cantidad de dígitos binarios.

Un ejemplo de "resultado sorpresa" es:

```
>>> 0.3 == (0.1 * 3)
False
>>> 0.1 * 3
0.30000000000000004
```

Comparar dos objetos tipo flotante directamente con operadores ==, o "!="
puede llevar a resultados inesperados y errores difíciles de depurar.

La solución es identificar si la diferencia entre dos flotantes es suficientemente pequeña,
en términos relativos, o absolutos, o en la cantidad de representaciones cercanas,
para los propósitos del programa en cuestión.
Si la diferencia es suficientemente pequeña, tomamos estos dos números como iguales.

Una solución, no muy buena, de este tipo de problema es usando la función "round":

```
round(x, D)
```

la cual redondea la variable x de tipo flotante a D dígitos decimales.
La idea es que números muy cercanos se redondean al mismo numero,
cuando la cantidad de dígitos decimales D es suficientemente pequeña.

Pero, hay que tener cuidado con el uso de "round".

Lo siguiente funciona:

```
>>> round(0.3, 6) == round((0.1 * 3), 6)
True
```

pero lo siguiente no:

```
>>> round(0.3, 6) == round(0.1, 6) * 3
False
```

Siempre hay que redondear la expresión completa, no solamente una parte.

Vamos a hablar más de solución de este problema en la sección 8.6.1 en la página 179.

Si necesitan un análisis más detallado de este problema y sus soluciones, lean el siguiente texto:
http://www.cygnus-software.com/papers/comparingfloats/comparingfloats.htm

6.2. Ejecución Condicionada (Ramificación)

Un bloque de declaraciones es una secuencia ordenada de declaraciones una tras otra,
normalmente escritas de manera que cada declaración empieza en una nueva línea de código.
De aquí en adelante vamos a usar la palabra "bloque" aunque se trate de una sola declaración.

Hasta ahora, todos nuestros ejemplos consistían de los bloques de declaraciones.

A veces deseamos, o tenemos que, condicionar un bloque.
Quiere decir, el bloque se ejecuta sólo si se cumple cierta condición.

Por ejemplo, Para usar un número como divisor, tenemos que usar la condición
que este número no sea cero. En las matemáticas no se usa la división entre cero.
El resultado de tal operación no existe, así que, la división con cero no tiene significado.

En Python existe la declaración "if-elif-else" para condicionar la ejecución de las declaraciones.
El "elif" es corto de "else, if".

6.2.1. La Declaración Condicional "if-elif-else"

La declaración condicional "if-elif-else" tiene tres partes "if", "elif" y "else", donde:

1. las partes "elif" y "else" son opcionales y
2. solamente la parte "elif" se puede repetir, y se puede repetir las veces que deseamos.

Esto nos lleva a cuatro posibles formas: if, if-else, if-elif, y if-elif-else.

A continuación se presentarán ejemplos de cada forma.

La Declaración Condicional "if"

Ahora cuidado, porque desde aquí vamos a empezar a usar los espacios (tabuladores),
o sea la sangría, como parte de la sintaxis del código Python.

La declaración "if" tiene el siguiente formato:

```
if <condición>:
    <declaración_condicionada>
```

En el primer renglón, la "condición" es una expresión lógica, se evalúa a tipo lógico.
Solamente puede ser False (falso), o True (cierto).

El segundo renglón tiene dos partes:
La primera parte son 4 espacios (1 tabulador) para desplazar la declaración hacia la derecha.
La segunda es la "declaración_condicionada".

El espacio inicial, o sea, la sangría de la segunda línea, forma parte de sintaxis de lenguaje Python.

La "declaración_condicionada" se ejecutará solamente si la "condición" de primer renglón es True.
Si la "condición" es False, el intérprete brincará la "declaración_condicionada" como si no existiera.

De hecho, en lugar de la "declaración_condicionada"
podríamos tener todo un bloque de declaraciones,
donde todas las declaraciones del bloque empiezan sangría de mismo tamaño.

Vamos a aprender esto con el ejemplo ya mencionado: No hay que dividir entre cero.
Un ejemplo:

```
print("Este programa calcula el recíproco de un entero.")
número = int(input("¿Para cuál número desea el recíproco? "))

recíproco = None
if número != 0:
    recíproco = 1.0 / número
print("El recíproco de", número, "es:", recíproco)
print("Gracias por usar este programa.")
```

La única línea condicionada es la que tiene sangría. Esta línea calcula el recíproco.

La penúltima línea que imprime el recíproco no está condicionada ya que no tiene sangría.
La última línea tampoco está condicionada.

La declaración, "recíproco = None", muestra el uso practico de tipo genérico: None.
Sin esta definición a None, o sea, si se remueve esta línea de código,
la definición de la variable "recíproco" es la declaración condicionada "recíproco = 1.0 / número".
En caso de que el usuario ingrese número 0, esta definición no sucedería y
la penúltima línea reportaría una excepción al tratar de imprimir la variable "recíproco" que no existe.

¡Ejecute este programa y haga pruebas ingresando varios números, entre cuales el cero!

En la siguiente modificación, la penúltima línea también es condicionada y
no es necesario definir la variable "recíproco" a None, ya que;
las dos únicas líneas que se refieren a la variable "recíproco" están condicionadas juntas.

```
print("Este programa calcula el recíproco de un entero.")
número = int(input("¿Para cuál número desea el reciproco? "))

if número != 0:
    recíproco = 1.0 / número
    print("El recíproco de", número, "es:", recíproco)
print("Gracias por usar este programa.")
```

En este caso tenemos un bloque de declaraciones condicionado.
El bloque condicionado consiste de dos declaraciones:
el cálculo del recíproco y su impresión a la pantalla.

La última línea sigue sin ser condicionada ya que no tiene la sangría.

El problema con este código es que cuando el usuario ingresa cero,
no se presenta ningún resultado en la pantalla y esto puede ser confuso para el usuario.
El remedio para esta situación es la siguiente forma.

La Declaración Condicional "if-else"

La forma "if-else" tiene el siguiente formato:

```
if <condición>:
    <bloque_ejecutado_en_caso_True>
else:
    <bloque_ejecutado_en_caso_False>
```

En la forma "if-else" siempre se ejecuta uno de los dos bloques de declaraciones.
Si la condición es True se ejecuta el bloque del "if": "bloque_ejecutado_en_caso_True".
Si la condición es False se ejecuta el bloque del "else": "bloque_ejecutado_en_caso_False".

El siguiente ejemplo muestra el uso de la forma "if-else", para mejorar el programa anterior.

```
print("Este programa calcula el recíproco de un entero.")
número = int(input("¿Para cuál número desea el reciproco? "))

if número != 0:
    print("El recíproco de", número, "es:", 1.0 / número)
else:
    print("El recíproco de 0 no existe.")
print("Gracias por usar este programa.")
```

Aparte de evitar la división con cero, la declaración "if-else" se usa cada vez cuando deseamos elegir entre dos opciones como por ejemplo: par-impar, aceptable-inaceptable, y otros.

Las Declaraciones Condicionales "if-elif" y "if-elif-else"

A veces tenemos más de dos opciones entre las cuales hay que elegir.

Digamos que tenemos opciones A, B, y C. Entonces la estructura que identifica la opción y ejecuta el bloque de declaraciones correspondientes, se puede escribir de tres maneras:

```
# "else" ultima opción   # "else" no usado        # "else" por defecto
# "if-elif-else"         # "if-elif"              # "if-elif-else"

if <opción> == <A>:      if <opción> == <A>:      if <opción> == <A>:
    <bloque_A>               <bloque_A>               <bloque_A>
elif <opción> == <B>:    elif <opción> == <B>:    elif <opción> == <B>:
    <bloque_B>               <bloque_B>               <bloque_B>
```

```
else:                    elif <opción> == <C>:    elif <opción> == <C>:
    <bloque_C>               <bloque_C>               <bloque_C>
                                                  else:
                                                      <bloque_por_defecto>
```

El "if" se usa solamente para averiguar la primera opción, digamos A: "if <opción> == <A>:",
después se usa "elif" para identificar la opción, o las opciones que siguen hasta la penúltima.
La última opción, "C", es donde los tres casos difieren.

La primera columna: "else" ultima opción,
muestra que se puede usar "else" para ejecutar el bloque correspondiente a la última opción.
Esto funciona solamente si estamos seguros que la variable "opción"
tiene uno de valores: A, B, o C.

La segunda columna: "else" no usado,
usa "elif <opción> == <C>:", para identificar la última opción y ejecutar su bloque; no se usa "else".
Esta versión funciona cuando "opción" puede tener otros valores aparte de A, B, o C,
pero no queremos ejecutar ninguna declaración en estos otros casos.

La tercera columna: "else" por defecto,
para cada posible valor de "opción" usa un "if" o "elif" que lo identifica y
el "else" se usa para el caso por defecto, cuando la "opción" no es ninguno de A, B, o C.
Normalmente el "bloque_por_defecto" se usa para reportar el error, o sea levantar una excepción,
si la "opción" no tiene ninguno de los valores esperados.

La Prevención de Error Común

La declaraciones condicionales "if" y "elif", por ejemplo:

```
if <condición>:
   <bloque_ejecutado_en_caso_True>
```

no aceptan declaraciones dentro de la "condición".
Solamente aceptan una expresión que evaluada resulta en un objeto lógico.

Esto asegura que el intérprete reportará el error, o sea levantará una excepción, al encontrar

```
if x = A:                # "x = A" es una declaración de asignación
```

en lugar de:

```
if x == A:               # "x == A" es una expresión lógica
```

Los que tienen experiencia como programadores en algunos otros lenguajes,
saben que este error es relativamente común y a veces difícil de encontrar.

6.2.2. Las Cuatro Formas Básicas de Condición

Existen cuatro formas básicas para una condición después del "if", o "elif".
Las formas no-básicas se logran usando los operadores lógicos: and y or,
para combinar las formas básicas en una expresión lógica compleja.

Para los siguientes ejemplos usaremos las variables "x" y "y".

La Primera Forma Básica: Sin Operadores de Comparación o Inclusión

La primera forma básica de condición es usando simplemente un objeto,
sin operadores de comparación ni inclusión.

Por ejemplo:

```
if x:
    <bloque>
```

En esta forma de condición, implícitamente se aplica la conversión de objeto "x" a objeto lógico,
aplicando la función de conversión "bool" implícitamente.
La declaración anterior es equivalente a:

```
if bool(x):
    <bloque>
```

Esta forma de condición no es muy clara y hay que evitarla, o
acompañarla con un comentario el cual va a aclarar la intención de su uso:

```
# Si la cadena "x" no está vacía
if x:
    <bloque>
```

Esta forma de condición también abarca el uso de una función en lugar de la variable:

```
# Si el resultado de la función no es cero
if <función>(x):
    <bloque>
```

Durante la depuración de un programa, cuando buscamos eliminar los errores,
a veces deseamos que temporalmente la condición siempre sea cierta o siempre falsa.
En tal caso usamos una de dos

```
# if x:                          # if x:
if True:                         if False:
    <bloque>                         <bloque>
```

La condición original queda convertida en comentario usando el símbolo #, y
en su lugar se introduce "if True:", o "if False:" dependiendo de lo que queremos lograr.

La segunda forma básica: "is", o "is not".

La segunda forma básica de condición es usando el operador "is", o "is not".

Esta forma se usa para cuatro propósitos;
para comparar: identificadores de domicilio en la memoria, None, False o True.

```
if x is y:        if x is None:       if x is False:      if x is True:
    <bloque>          <bloque>            <bloque>            <bloque>

if x is not y:    if x is not None:   if x is not False:  if x is not True:
    <bloque>          <bloque>            <bloque>            <bloque>
```

Recuerde que:
el "x is y" es equivalente a "id(x) == id(y)", y
el "x is not y" es equivalenta a "id(x) != id(y)".

Los "x == None", "x == False", "x == True" y "x != None", "x != False", "x != True",
no deben de ser usados; el intérprete ya no garantiza los resultados de estos.

La tercera forma básica: "in", o "not in".

La tercera forma básica de condición es usando el operador de inclusión "in", o "not in".

Esta forma se usa cuando se quiere averiguar la inclusión de un elemento o sub-colección,
dentro de una colección.

```
if x in <colección>:
    <bloque>
```

```
if y not in <colección>:
    <bloque>
```

Por ahora la única colección que conocemos es la cadena.
Pronto conoceremos más colecciones: lista, tupla, diccionario y conjunto, y
esta forma de la condición va a ser usada más seguido.

La cuarta forma básica: <, <=, ==, !=, >=, >

La cuarta forma básica de condición es usando operadores de comparación: <, <=, ==, !=, >=, >.

Estos operadores aplican solamente a los objetos comparables y
comparan los valores, o contenidos de objetos.

Estos operadores no se deben usar para comparar con el tipo None, o tipos lógicos: False o True.

A los tipos numéricos estos operadores aplican en sentido matemático con restricción que
los operadores == y != no se deben usar para comparar los números de tipo flotante.

Estos operadores aplican a las cadenas a través de los números ordinales de los caracteres.

Con cada nuevo tipo de datos en Python que aprendemos,
también aprenderemos como se comparan.

Más adelante crearemos nuestros propios tipos de datos y
aprenderemos como hacerlos comparables.

6.2.3. La Forma "if-else" de Una Línea

Cuando se desea asignar uno de dos valores, a una variable, dependiendo de una condición,
se puede usar la declaración "if-else" de una línea:

```
<variable> = <valor_T> if <condición> else <valor_F>
```

A veces se dice que esta declaración es un "operador ternario",
dado que "if" y "else" separan tres operandos: "valor_T", "condición" y "valor_F".

Esta declaración condicional "if-else" de una línea es equivalente a
la declaración condicional "if-else" que ya conocemos:

```
if <condición>:
    <variable> = <valor_T>
else:
    <variable> = <valor_F>
```

Ejecute:

```
print("Este programa reporta el signo de su número entero.")

x = float(input("Ingrese un número entero: "))

if x == 0:
    print("El cero no es ni positivo, ni negativo.")
else:
    signo = "positivo" if x >= 0 else "negativo"
    print(signo)
```

6.2.4. Declaraciones Condicionales Anidadas

A veces, en una declaración condicional, el bloque condicionado puede ser, o puede contener, otra declaración condicional.

Por ejemplo, deseamos identificar si una variable x de tipo flotante es negativa, cero, o positiva. Distinguir entre las tres opciones se puede lograr de dos maneras:
declaración condicional anidada, o declaración condicional de múltiple opción:

```
# declaraciones anidadas
if x < 0.0:
    print("Negativo")
else:
    if x > 0.0:
        print("Positivo")
    else:
        print("Cero")
```

```
# múltiple opción
if x < 0.0:
    print("Negativo")
elif x > 0.0:
    print("Positivo")
else:
    print("Cero")
```

A la izquierda tenemos código que usa declaraciones condicionales anidadas:
en el externo "if-else", el bloque de "else" es otro "if-else" interno, anidado.

A la derecha se logra lo mismo, usando "if-elif-else" para verificar varias condiciones.
El uso de la múltiple opción es preferible sobre las declaraciones condicionales anidadas, en casos sencillos cuando existe múltiple opción equivalente.

Otro ejemplo. Para que un alumno cumpla con el curso, debe de asistir al menos 80 % de clases y debe de tener promedio mayor o igual a 70 %.
El programa reporta si el alumno aprobó el curso y si no, el razón de reprobar.

```
asistencia = float(input("Ingrese asistencia (0% a 100%): ")) / 100.0
promedio = float(input("Ingrese su promedio (0 a 100): ")) / 100.0

if asistencia >= 0.8:
```

```
        if promedio >= 0.7:
            print("Aprobado.")
        else:
            print("Reprobado por promedio bajo.")
    else:
       print("Reprobado por baja asistencia.")
```

El bloque de "if" en la declaración condicional externa, es un "if-else" anidado, interno.

Esta lógica también se puede lograr con el operador lógico "and", en lugar de condiciones anidadas:

```
asistencia = float(input("Ingrese asistencia (0% a 100%): ")) / 100.0
promedio = float(input("Ingrese su promedio (0 a 100): ")) / 100.0

if asistencia >= 0.8 and promedio >= 0.7:
    print("Aprobado.")
else:
    print("Reprobado.")
```

Este caso usa operador lógico "and" para evitar el uso de declaraciones condicionales anidadas, pero pierde la capacidad de identificar la razón de reprobar.

Podemos identificar los cuatro casos con "if-elif-else":

```
asistencia = float(input("Ingrese asistencia (0% a 100%): ")) / 100.0
promedio = float(input("Ingrese su promedio (0 a 100): ")) / 100.0

if asistencia >= 0.8 and promedio >= 0.7:
    print("Aprobado.")
elif asistencia < 0.8 and promedio >= 0.7:
    print("Reprobado por baja asistencia.")
elif asistencia >= 0.8 and promedio < 0.7:
    print("Reprobado por promedio bajo.")
else:
    print("Reprobado por baja asistencia y promedio bajo.")
```

Como pueden ver, tomar decisiones a base de varias condiciones
se puede lograr de varias maneras. depende de:
qué se quiere lograr, conocimiento de lenguaje Python y creatividad en diseño de código.

6.2.5. Ejercicios

Escriba un programa que ...

1) pide un entero al usuario y reporta si es divisible entre 3, o no.

2) pide dos números al usuario y reporta el mayor, o
reporta que son iguales.

3) pide un entero de 1 a 5 y lo escribe en correspondiente símbolo romano: I, II, III, IV, V.

4) pide precio de un libro en la librería A y en la librería B.
reporta el porcentaje de diferencia en precios con referencia al precio más barato.

5) pide el usuario su ingreso mensual comprobable, y la cantidad que debe al banco.
Para obtener un préstamo nuevo se necesita mínimo $10,000 pesos de ingreso mensual y
no deber al banco más de $30,000 pesos de prestamos anteriores.
Reporta si el usuario aprueba las condiciones para el crédito y
si no los aprueba reporta cuál es la razón: bajo ingreso mensual, alta deuda al banco, o ambos.

6.3. Ciclos (Bucles)

A veces tenemos una secuencia de datos sobre cuales hay que ejecutar una misma declaración, o un mismo bloque de declaraciones.
Para simplificar, vamos a hablar de bloque, aunque puede que este sea una sola declaración.

Para lograr la repetición de un bloque, sobre varios datos, se usan las
declaraciones "while-else", o "for-else". Los vamos a llamar: ciclo "while" y ciclo "for".

Las formas más completas de las declaraciones "while-else" y "for-else" son:

```
<bloque_preparatorio>
while <condición>:
    <bloque_de_ciclo>
else:
    <bloque_else>
```

```
<bloque_preparatorio>
for <condición>:
    <bloque_de_ciclo>
else:
    <bloque_else>
```

En ambos casos, la parte "else" es opcional:

```
<bloque_preparatorio>
while <condición>:
    <bloque_de_ciclo>
```

```
<bloque_preparatorio>
for <condición>:
    <bloque_de_ciclo>
```

Los bloques preparatorios pueden ser vacíos, o sea no existentes:

```
while <condición>:
    <bloque_de_ciclo>
```

```
for <condición>:
    <bloque_de_ciclo>
```

Los bloques de ciclo siempre deben de existir, aunque podemos dejarlos temporalmente vacíos, durante el proceso de desarrollo de código.

Para dejar bloques de ciclo temporalmente vacíos se usa la declaración "pass" (pasar).
En las ultimas versiones de Python se introdujo el operador "..."
que hace lo mismo que hace "pass", o sea, no hace nada,
más que causar que el procesador no haga nada durante un par de ciclos.
Se produce una pause de duración de unos micro segundos.

```
while <condición>:
    pass
while <condición>:
    ...
```

```
for <condición>:
    pass
for <condición>:
    ...
```

Los bloques de ciclos así son sintácticamente correctos y tal código se va a ejecutar,
mientras estamos desarrollando y depurando otras partes del código.
Después debemos regresar y escribir los bloques de estos ciclos,
eliminando todos los "pass" y los "...".

Lo que vamos a ver en los ejemplos que siguen es lo siguiente:

1) El ciclo "while" es controlado por una condición que puede ser:
dependiente de un contador, o dependiente de un centinela.

2) El ciclo "for" es controlado por una condición que puede ser:
dependiente de un iterador, o dependiente de una colección de datos (ítems, elementos).

Una colección tiene todos sus ítems en la memoria listos para ser leídos.
Un iterador genera cada ítem en el momento en que se le pida.

6.3.1. El Ciclo "while" Condicionado por un Contador

Un contador es una variable que lleva cuenta sobre la cantidad de repeticiones.
Para los contadores, tradicionalmente se usan los nombres: i, j, k.

Digamos que vamos a calcular la suma de cuatro números proporcionados por el usuario.
Podemos usar el ciclo "while", ya que cada número será procesado de la misma manera.

```
MAX = 4
print("Este programa calcula la suma de", MAX, "números.")

suma = 0.0
i = 0
while i < MAX:
    i = i + 1
    número = float(input("Proporcione el dato " + str(i) + ": "))
    suma = suma + número
print("La suma es:", suma)
```

Primero establecemos una variable MAX que guarda la cantidad de repeticiones que deseamos.
A veces este dato se obtiene del usuario, usando la función "input".

En el bloque preparatorio establecemos que: la suma empieza desde 0.0;
ya que todavía no hemos sumado ningún número,
y el contador empieza desde 0; ya que todavía no hemos hecho ninguna repetición.

El ciclo "while" implica que vamos a hacer varias repeticiones del bloque de ciclo y
la condición que determina el fin de las repeticiones es "i $<$ MAX". Esta condición compara
el contador de repeticiones ejecutadas contra la cantidad máxima deseada de las repeticiones.

Si la condición es cierta, se ejecuta el bloque del ciclo y
se regresa a evaluar la condición de "while" otra vez.

Bloque de ciclo se repetirá cuatro veces para valores: 0, 1, 2 y 3 del contador "i".
Cuando el contador "i" llegue a valor 4, la condición no se cumple y
las repeticiones del bloque de ciclo se han terminado.

Después del ciclo "while" se ejecuta la función "print" la cual imprime la suma deseada.

En este ejemplo, dentro del bloque de ciclo en cuestión hacemos tres cosas:
1) aumentamos el contador de repeticiones
2) pedimos el dato al usuario
3) agregamos el dato a la suma.

Algunos programadores prefieren aumentar el contador de repeticiones "i = i + 1"
al final del bloque del ciclo. En este caso el bloque de ciclo cambiaría a:

```
while i < MAX:
    número = float(input("Ingrese el dato " + str(i + 1) + ": "))
    suma = suma + número
    i = i + 1
```

Con pequeño ajuste "str(i + 1)" en el argumento de "input"
el programa se ejecutaría con el mismo efecto.

Dado que "input" recibe solamente una cadena, tenemos que usar la función de conversión "str"
para convertir el entero "i + 1" a cadena y concatenarla a otras dos cadenas.

6.3.2. Operadores de Asignaciones Especiales

El uso de ciclos en los cuales se ajustan los contadores y acumuladores, como la suma,
es tan frecuente que existen operadores de asignación especiales para esto.

Por ejemplo

```
i = i + 1
suma = suma + número
```

se escribe

```
i += 1
suma += número
```

para lograr el mismo efecto: aumentar contador por 1 y la suma por el valor de número.

Existen tales operadores de asignación para casi cada operador aritmético,

```
+=, -=, *=, /=, //=,%=, **=,
```

y operador de bits.

```
&=, |=, ^=, <<=, >>=
```

Los vamos a usar ya que son optimizados para ser un poco más rápidos que asignación simple =.

Algunos lenguajes tienen el operador ++ equivalente a "+= 1" y el operador – equivalente a "-= 1".
El lenguaje Python no tiene los operadores ++ y –; los "+= 1" y "-= 1" están optimizados.

6.3.3. El Ciclo "while" Condicionado por un Centinela

Un centinela es un valor especial que indica el fin de los datos, y no es un dato en sí.
Por ejemplo, si vamos a sumar una cantidad de datos no negativos,
podemos usar el valor -1 como centinela para identificar que ya no hay más datos.

El siguiente ejemplo es un programa que recibe una cantidad desconocida de números positivos,
y calcula su media. La media es la suma de los datos dividida entre la cantidad de los datos.

```
print("Este programa suma números no-negativos.")
print("Proporcione el valor -1 cuando no hay más datos.")

suma = 0.0
i = 0
dato = float(input("Proporcione un número, o el -1: "))
while dato >= 0.0:
    suma += dato
    i += 1
    dato = float(input("Proporcione un número, o el -1: "))

media = None if i == 0 else (suma / i)
print("La media de los datos es: ", media, ".", sep="")
```

En este caso tenemos el primer dato recibido en el bloque preparativo, antes del ciclo.

Dentro del ciclo agregamos el dato ya obtenido en la suma, lo contamos y
pedimos el siguiente dato. El ciclo se repite mientras el dato es un número no-negativo.

Después del ciclo, se calcula la media y se imprime el resultado.
El "if-else" de una línea evita la división con cero al calcular la media si no hubo datos.

Este ciclo "while", aunque existe un contador, es controlado por el centinela,
ya que el contador no aparece en la condición "dato >= 0.0" de la declaración "while".

6.3.4. El Ciclo "for" Condicionado por un Iterador

Un iterador es un objeto que nos proporciona una secuencia de datos de manera que,
los datos son generados de manera que en cada llamada al iterador se regresa el siguiente dato,
o la señal "StopIteration" si no hay más datos.
De esta manera se ahora espacio de la memoria.

En los capítulos más avanzados aprenderemos sobre iteradores en detalle y
aprenderemos a diseñar nuestros propios iteradores.
Ahora aprenderemos a usar un iterador llamado "range".

El Iterador "range"

El iterador "range" genera una secuencia de objetos tipo entero.
Existen tres maneras de usar el "range", con uno, dos o tres argumentos:

```
range(FIN)              # FIN - el enero no incluido
range(INI, FIN)         # INI - el primer entero
range(INI, FIN, PASO)   # PASO - el paso de avance
```

Esto se aclara en los siguientes ejemplos.

El caso "range(FIN)" genera valores enteros desde cero hasta inclusive el predecesor del FIN.

```
range(5)                # Genera 5 objetos: 0, 1, 2, 3 y 4.
```

El caso "range(INI, FIN)" genera enteros desde INI hasta inclusive el predecesor del FIN.

```
range(11, 16)            # Genera objetos: 11, 12, 13, 14 y 15.
```

El caso "range(INI, FIN, PASO)" genera valores que avanzan en la cantidad que dicta el PASO. y termina antes del FIN.

```
range(2, 20, 4)          # Genera objetos: 2, 6, 10, 14, 18.
range(9, -10, -3)        # Genera objetos: 9, 6, 3, 0, -3, -6, -9.
```

El "range" es el iterador que frecuentemente se usa para controlar el ciclo "for".

El Ciclo "for" con Iterador "range"

Digamos que deseamos imprimir tabla de cuadrados y cubos de números 1 a 10.
Esto se puede lograr usando el ciclo "for" guiado por el iterador "range".

```
# Imprimir el encabezado
print("  x   x^2   x^3 \n" + "=" * 15)

# Imprimir los renglones de la tabla uno en cada repetición
for i in range(1, 11):
    print(format(i, '3d'), format(i * i, '4d'), format(i ** 3, '5d'))

# Imprimir una linea al fin de la tabla
print("-" * 15)
```

El primer "print" imprime el encabezado de la tabla.

El "range" genera enteros 1 a 10, el 11 ya no.
El bloque del ciclo imprime tres enteros en cada repetición,
el número, su cuadrado y su cubo.

El ultimo "print" imprime una línea para señalar el fin de la tabla.

6.3.5. El Ciclo "for" Condicionado por una Colección

A diferencia de un iterador que genera cada ítem (elemento) en el momento en que se le pida,
o manda la señal "StopIteration",
una colección ya tiene todos sus ítems (elementos) en la memoria.

La única colección de datos que conocemos hasta ahora es la cadena,
el cual es una colección de caracteres, o sub-cadenas.

Según vamos aprendiendo más colecciones, practicaremos el uso del ciclo "for" con ellas.

El siguiente es el ejemplo de uso de ciclo "for" para procesar caracteres de una cadena.
En este ejemplo, en cada repetición simplemente se imprime un carácter seguido de un espacio.

```
cadena = "abecedario"

for c in cadena:
    print(c, end=" ")
print()
```

También se pueden usar las funciones "reversed" y "sorted" para procesar los ítems,
o sea los caracteres al revés, u ordenados por su número ordinal "ord" en TUF_8.

```
cadena = "abecedario"

for c in reversed(cadena):
    print(c, end=" ")
print()

for c in sorted(cadena):
    print(c, end=" ")
print()
```

La función "reversed" se puede usar en todas las colecciones de datos de Python,
La función "sorted" se puede usar en todas las colecciones que contienen objetos comparables.

Ya sabemos procesar colección tal como la ordenamos, a la reversa y ordenada por el Python.
Después aprenderemos el cuarto caso, cuando deseamos ordenar los elementos al azar.

6.3.6. Ciclos Anidados

Los ciclos "while" y "for" se pueden usar anidados uno dentro del otro en cualquier orden:
"while" en "for", "for" en "for", "for" en "while" y "while" en "while", hasta varios niveles.

El siguiente ejemplo es un reloj descompuesto.
Muestra la hora desde 00:00:00 hasta 23:59:59 de manera no sincronizada con tiempo real.
Después aprenderemos a diseñar un reloj real en una interfaz gráfica de usuario.

```
for hours in range(24):
    for minutes in range(60):
        for seconds in range(60):
            print(format(hours, '2d'), ':',format( minutes, '2d'), ':',
                    format(seconds, '2d'), sep="")
            for i in range(100):
                pass
```

El ejemplo presenta uso de ciclos "for" anidados.
También presenta una declaración "print" a través de dos líneas de código.

La última declaración "for", la más interna:

```
for i in range(100):
    pass
```

es una mala manera de controlar la pausa, modificando el número 100.
Aunque la declaración "pass", dice "no hagas nada" al intérprete,
este "no hagas nada" de hecho causa que el procesador durante algunos ciclos no haga nada, y así pierda tiempo. Perder tiempo así no es buena idea.
Al estudiar el modulo "time" aprenderemos controlar tiempo de una pausa de manera mejor.

Otro ejemplo de ciclos "for" anidados escribe un rectángulo hecho de símbolos gato: #.

```
RENGLONES = 6
COLUMNAS = 6

for r in range(RENGLONES):
    for c in range(COLUMNAS):
        print("#", end="")
    print()
```

También podemos hacer que la cantidad de ciclos en el "for" interno, anidado,
dependa del contador de ciclos "r" del "for" externo.
Para imprimir una cantidad mayor de gatos # en cada repetición.
En este caso, como el ciclo interno es muy sencillo, esto se puede lograr de dos maneras:

```
RENGLONES = 6

for r in range(RENGLONES):
    for c in range(r + 1):
        print("#", end="")
    print()
```

```
RENGLONES = 6

for r in range(RENGLONES):
    print("#" * (r + 1))
```

Ejecute estos programas.

Use el ejemplo con un ciclo "for" y modifique el código para lograr los siguientes resultados:

```
EJERCICIO 1
#
 #
  #
   #
    #
     #
```

```
EJERCICIO 2
##
# #
#  #
#   #
#    #
#     #
```

```
EJERCICIO 3
######
#####
####
###
##
#
```

6.3.7. Las Declaraciones "continue" y "break"

Aparte de usar los contadores y las centinelas para guiar un ciclo "while" y
de usar los iteradores y las colecciones para guiar un ciclo "for",
existen dos declaraciones: "continue" y "break" para los siguientes propósitos:

"continue" interrumpe la repetición actual y transfiere ejecución a la condición del ciclo.

"break" interrumpe la repetición actual y no permite más repeticiones de este ciclo,
o sea, transfiere la ejecución a la declaración que sigue después del ciclo y su "else".

Como siempre, lo mejor es aprender de los ejemplos.

Vamos a calcular la raíz cuadrada de una serie de números positivos recibidos del usuario.
No calculamos raíces de números negativos ya que no queremos entrar a los números complejos.

```
print("\nEste programa calcula la raíz cuadrada de números positivos.")
print("Oprima [Enter] sin dato para terminar el programa.\n")

while True:
```

```
        dato = input("Ingrese un número positivo, o sólo [Enter]: ")

        if dato == "":
            break

        n = float(dato)
        if n < 0.0:
            continue

        print("\tLa raíz cuadrada de", n, "es", format(n ** 0.5, '.9f'), '\n')

    print("\nGracias por usar el programa.\n")
```

En este programa la primera novedad es el uso de "while True:".
La única manera de que este ciclo termine sus repeticiones es que se ejecute
la declaración "break" que está dentro del ciclo.

La segunda novedad es el uso de la declaración "break" y
la tercera novedad es el uso de la declaración "continue".

Analice este programa y ejecútelo proporcionando números positivos y negativos.
Para completar el programa oprima la tecla Enter, o sea Return, sin proporcionar ningún dato.

No siempre se tienen que usar ambos, y "continue" y "break". Se usan de manera independiente.

El siguiente ejemplo muestra el uso de "break".

Analicen y ejecuten el siguiente programa.

```
# Este programa escribe los caracteres de la primera palabra.

cadena = "Hola usuario."

for c in cadena:
    if c == " ":
        break
    print(c, end="")
print()
```

6.3.8. El "while-else" y "for-else"

Las formas más completas de las declaraciones "while-else" y "for-else" son:

```
<bloque_preparatorio>
while <condición>:
    <bloque_de_ciclo>
else:
    <bloque_else>
```

```
<bloque_preparatorio>
for <condición>:
    <bloque_de_ciclo>
else:
    <bloque_else>
```

El "bloque_else", el que sigue "while", o igual el que sigue "for",
se ejecuta cuando las repeticiones de ciclo han terminado, pero
solamente bajo la condición que las repeticiones no fueron interrumpidas por un "break".

Si "while" o "for" terminan las repeticiones por causa de la declaración "break",
el "bloque_else" no se va a ejecutar. Se continua con declaraciones que siguen después.

Por ejemplo, deseamos factorizar números enteros de 2 a 10 y a la vez identificar los primos.

```
for n in range(2, 11):
    for i in range(2, n):
        # Si el "i" es un factor de "n"
        if n % i == 0:
            # El otro factor es
            j = n // i
            print(n, "=", i, "*", j)
            # para que el "else" no se ejecute, provocamos un "break"
            break
    else:
        print(n, "es número primo.")
```

El "for" externo no tiene su "else", mientras el "for" interno lo tiene.
El "break" condicionado por el "if n % i == 0:" controla si el "else" se ejecuta, o no.

Si se encontró algún factor de n, se ejecuta el bloque de "if" dentro de cual es un "break".
Dado que la repetición de "for" es interrumpida por el "break", el "else" no se ejecuta.

Si no se encontró ningún factor de n, no se entra al bloque del "if", no se ejecuta el "beak" y
el "else" se ejecuta y reporta que n es un número primo.

6.3.9. Ejercicios

Escriba un programa que ...

1) haga lo siguiente. Editorial vende libros a las librerías con 40 % de descuento.
El costo de envío es $10 para el primer libro más $1 por cada libro adicional.
Escriba un programa que pide la cantidad de libros (2 a 5),
pide el precio de cada libro ($100.00 a $999.00) y
reporta el precio total (libros y envío) de compra.

2) escriba una tabla de números 1 a 20 en la primera columna.
La segunda columna son los cuadrados de estos números.
La tercera columna son los cubos, pero solamente para números de 1 a 10.
Para los números de 11 a 20 no se escriben los cubos.

6.4. Librería Estándar

6.4.1. El Modulo "random", Números Seudo-Aleatorios

Los números seudo-aleatorios simulan generación de números al azar.
Los números son de un intervalo o una colección proporcionada, y
siguen la función elegida de distribución, o densidad, de probabilidad.

Los números no son totalmente al azar, sino seudo-aleatorios porque
existe algún algoritmo que los genera.

Generación de números al azar es importante para diversas aplicaciones,
entre cuales los juegos y las simulaciones.

La documentación sobre el modulo "random" se puede encontrar en el sitio:
https://docs.python.org/3/library/random.html

El Contenido de Modulo "Random"

Para trabajar con los números seudo-aleatorios, llamados "números al azar".
primero tenemos que importar le modulo "random":

```
import random
```

Algunas de las funciones de la librería random son.

```
# FUNCIÓN                         # LO QUE DEVUELVE

random.randint(MIN, MAX)          # Uno de: MIN, MIN+1, MIN+2,  ...,  MAX.
random.randrange(FIN)             # Uno de: 0, 1, 2, 3, 4, ..., FIN-1.
random.randrange(INI, FIN)        # Uno de: INI, INI+1,  ...,   FIN-1.
random.randrange(INI, FIN, STEP)  # Uno de: INI, INI+PASO, ..., FIN-?.

random.random()                   # Un flotante de intervalo [0, 1).
random.uniform(I, F)              # Un flotante de intervalo [I, F).

random.choice(colección)          # Uno de los elementos de la colección.
random.shuffle(colección)         # La colección reordenada al azar.
```

Para iniciar la secuencia de números aleatorios el intérprete de Python usa
la información de generadores de números aleatorios de sistema operativo, o de tiempo actual.
De esta manera cada vez que se inicie el programa se genera una secuencia distinta.

A veces, para los propósitos de depurar un programa,
deseamos que se repita la misma secuencia cada vez que lo iniciemos.

La función "seed" nos permite establecer un valor inicial para los números seudo-aleatorios,
de manera que cada vez que iniciemos el programa se va a repetir la misma secuencia.

La función "seed" acepta cualquier tipo de dato incorporado.

Usando el Modulo "random"

Vamos a ver el primer ejemplo que genera números enteros seudo-aleatorios.

No es necesario importar todo el modulo "random".
Se importan solamente las funciones que necesitamos.

```
from random import seed, randint

NÚMEROS = 50
MIN = 8
MAX = 12
```

```
seed(7)

for i in range(NÚMEROS):
    if i% 10 == 0:
        print()
    print(format(randint(MIN, MAX), '4d'), end="")

print('\n')
```

Este programa imprime una secuencia de números enteros seudo-aleatorios,
en el intervalo proporcionado a la función "randint".
Los números MIN y MAX están ambos incluidos entre los generados.

Ya que se está invocando la función "seed",
la misma secuencia de números resultará cada vez que se ejecute el script.
Esto es bueno en proceso de depuración de código para reproducir el mismo error.
Una vez que programa es depurado, se convierte la declaración de llamada a "seed" a comentario:

```
# seed(7)
```

Después de esto, cada vez que se ejecute el script,
se genera una secuencia distinta de los enteros.

La función "randrange" tiene los mismos parámetros como la función "range" que ya conocemos.
La diferencia es que "randrange" regresa los enteros de la secuencia "range" al azar.

```
from random import randrange

NÚMEROS = 50
INI = 3
FIN = 13
PASO = 3
print("\nLos números entre los cuales se generarán los seudo-aleatorios:")
for i in range(INI, FIN, PASO):
    print(format(i, '4d'), end="")
print()

print("\nNúmeros seudo-aleatorios generados a base de la lista anterior:")
for i in range(NÚMEROS):
    if i% 10 == 0 and i != 0:
        print()
    print(format(randrange(INI, FIN, PASO), '4d'), end="")
print('\n')
```

Si deseamos generar números tipo flotante de distribución uniforme,
tenemos funciones: "random" y "uniform".

También vamos a mostrar otra manera de importar.

```
import random as rnd
```

```
NÚMEROS = 20

print("\nNúmeros seudo-aleatorios del intervalo [0, 1):")
for i in range(NÚMEROS):
    if i% 4 == 0:
        print()
    print(format(rnd.random(), '18.12f'), end="")
print()

INI = 10.0
ANCHO = 5.0
print("\nNúmeros seudo-aleatorios del intervalo [10, 15):")
for i in range(NÚMEROS):
    if i% 4 == 0:
        print()
    n = ANCHO * rnd.random() + INI        # n = rnd.uniform(INI, INI + ANCHO)
    print(format(n, '18.12f'), end="")
print('\n')
```

La declaración

```
n = ANCHO * rnd.random() + INI
```

es equivalente a

```
n = rnd.uniform(INI, INI + ANCHO)
```

Existen más funciones para distintas distribuciones estadísticas.
Para detalles hay que estudiar el modulo "random" en la documentación oficial.

6.4.2. Ejercicios

6.5. Recomendaciones y Ejemplos

6.5.1. Recomendaciones

6.5.2. Ejemplos

Programas con Menú

El siguiente código muestra varias cosas nuevas:
El uso de cadena de múltiples líneas, delimitado por triples comillas.
El uso de tres declaraciones condicionales anidadas "if-else", "if", y "if-elif-else".

Analice este programa y ejecute lo.

```
print('=' * 60, end="")
print(“””
    Este programa ejecuta una operación de su elección.
        0) SALIR
```

```
        1) Suma
        2) Multiplicación
"""")
print('=' * 60)

opción = int(input("¿Cuál número de menú elige (0 a 2)? "))
if 0 <= opción <= 2:
    if opción != 0:
        a = float(input("\n¿Cuál es el primer número?  a = "))
        b = float(input("¿Cuál es el segundo número? b = "))
        print()
        if opción == 1:
            print(a, "+", b, "=", a + b)
        elif opción == 2:
            print(a, "*", b, "=", a * b)
        else:
            print("ERROR. El programa no debería llegar aquí.")
else:
    print("ERROR: La opción", opción, "no está en el rango 0 a 2.")
print('=' * 60)
print()
```

Tabla de Temperaturas

El siguiente programa convierte escala de temperatura
de grados fahrenheit a grados celsius y a la escala absoluta de Kelvin.

```
TABLE_WIDTH = 30

print("\nEscalas de temperatura:")
print("fahrenheit celsius Kelvin")
print("=" * TABLE_WIDTH)

for fahrenheit in range(-50, 151, 10):
    celsius = 5.0 * (fahrenheit - 32) / 9.0
    kelvin = celsius + 273.15

    print(format(fahrenheit, '9.2f'), format(celsius, '9.2f'),
          format(kelvin, '9.2f'))

print('-' * TABLE_WIDTH, '\n')
```

Validación de la Entrada con "while"

Al procesar cierta cantidad de datos recibidos del usuario a través del teclado,
aparte de usar un contador, o un centinela, para determinar el fin de los datos,
en cada ciclo podemos preguntar el usuario si tiene más datos.
Eso lo haremos en el siguiente ejemplo.

El ejemplo también muestra el uso de ciclo "while" anidado, para validar la respuesta.
La validación es un poco exagerada, sólo para mostrar lo que se puede hacer con el "while".

```
print("Este programa calcula la media de los datos.")

suma = 0.0
i = 0
más = 's'
while (más == "S" or más == "s"):
    dato = float(input("Proporcione el dato: "))
    suma += dato
    i += 1

    # Aceptar la entrada solamente si se recibe uno de: "S", "s", "N", "n"
    valido = False
    while not valido:
        más = input("¿Tiene más datos? (s/n): ")
        if (más == "S" or más == "s" or más == "N" or más == "n"):
            valido = True

media = None if i == 0 else (suma / i)
print("La media de los datos es:", media)
```

Después de aprender sobre las excepciones ("exceptions"),
vamos a poder escribir todavía mejores validaciones de los datos de entrada.

Por ahora sólo vamos a notar que este caso es peor que usar contador o centinela,
ya que el usuario tiene que responder a dos "input" por cada dato que proporcione.

Simulación de "do-while"

Algunos lenguajes tienen una declaración conocida como el ciclo "do-while".
Es similar a "while", solamente que la condición de "while" está después del bloque:

```
do:
    <bloque>
while <condición>
```

En la forma "do-while" el bloque de ciclo es ejecutado al menos una vez
antes de que la condición es evaluada.

En Python no existe "do-while" pero se puede lograr de dos maneras:

```
<bloque>
while <condición>:
    <bloque>
```

```
while True:
    <bloque>
    if not <condición>:
        break
```

El siguiente ejemplo, que ya vimos anteriormente, muestra el caso de la izquierda.

```
print("Este programa suma números no-negativos.")
print("Proporcione el valor -1 cuando no hay más datos.")

suma = 0.0
i = 0
dato = float(input("Proporcione un número, o el -1: "))
while dato >= 0.0:
    suma += dato
    i += 1
    dato = float(input("Proporcione un número, o el -1: "))

media = None if i == 0 else (suma / i)
print("La media de los datos es: ", media, ".", sep="")
```

Pedimos el primer dato antes de entrara al ciclo "while",
así que esto es ejecutado al menos una vez, como si fuera "do-while".

Calculando la Varianza

```
print("Este programa calcúla media y vanianza de población.")
n = int(input("¿Cuantos datos hay en la muestra? "))

if n > 0:
    suma_de_datos = suma_de_cuadrados = 0.0
    for i in range(n):
        mensaje = "Ingrese el dato " + str(i + 1) + " de " + str(n) + ": "
        dato = float(input(mensaje))
        suma_de_datos += dato
        suma_de_cuadrados += dato * dato

    media_de_datos = suma_de_datos / n
    media_de_cuadrados = suma_de_cuadrados / n
    varianza = media_de_cuadrados - media_de_datos ** 2.0
    print("La media de la población es: ", format(media_de_datos, '.4g'))
    print("La varianza de la población es:", format(varianza, '.4g'))

print("Gracias.")
```

Usando "choice"

La función "choice" del modulo "random" elige ítem al azar de una colección de ítems.

```
import random as rnd

ÏTEMS = 40
TEXTO = "cadena"

print("\nNúmeros seudo-aleatorios del intervalo [0, 1):")
```

```
for i in range(ÏTEMS):
    if i% 8 == 0:
        print()
    print(format(rnd.choice(TEXTO), '>4s'), end="")
print('\n')
```

6.6. Tareas

1) El primer día de enero es Domingo. Pida el usuario el día de enero, 1 a 31, y
escriba el acrónimo de este día: Lun, Mar, Mie, Jue, Vie, Sab, Dom.

2) Pide al usuario tres enteros y los reporta en orden: mayor, mediano, menor.
si algunos o todos son iguales los imprime en orden A >= B >= C.

3) Modifique el ejemplo "programas con menú" agregando opciones "Restar" y "Dividir".
La opción "Dividir" debe de evitar la división entre cero.

4) El primer día del año 2016 es Viernes. El año es bisiesto, el Febrero tiene 29 días.
Pida el usuario el mes 1 a 12 y el día del mes, sin aceptar dato fuera de rango real.
Escriba el acrónimo de este día: Lun, Mar, Mie, Jue, Vie, Sab, Dom.

5) Pide al usuario los números de día, mes y año de su nacimiento.
Si la suma de los dígitos es mayor que 9, suma los dígitos del resultado.
Continua sumando dígitos de los resultados hasta obtener resultado de un dígito.
Se imprime el dígito final resultante.

6) Pide un número en sistema hexadecimal y lo convierte en sistema decimal

7) Genera un número entero al azar de 1 a 1000.
Pide al usuario adivinar el número y reporta: bajo, exacto o alto.
Si numero de usuario es bajo o alto, la da la oportunidad de adivinar de nuevo.
El juego continua hasta que el usuario adivine el número en cual caso gana, o
si el usuario falló por doceava vez, en cual caso perdió.

8) Escriba la lista de primeros 25 números primos.

9) Escriba el programa que juega "piedra, papel y tijeras" contra el usuario.
Programa genera al azar uno de tres valores 1, 2, o 3 que corresponden a piedra, papel o tijeras.
Después ofrece menú al usuario para elegir 1) piedra, 2) papel o 3) tijeras.
Al final el programa compara los dos objetos elegidos:
papel cubre la piedra, tijeras cortan el papel, piedra rompe las tijeras y reporta quién ganó:
"gana la máquina", empatados, o "gana el usuario".

7. Colecciones Incorporadas

Colecciones "incorporadas" son aquellas que son parte del intérprete y
no necesitan ser importadas de las librerías. Son disponibles automáticamente.

Mucho del poder de las colecciones viene cuando se usan en la programación estructurada;
cuando son procesadas bajo ciertas condiciones y dentro de los ciclos de repeticiones.

Aquí aprenderemos la primera parte de trabajo con las colecciones: que características tienen.
En el capitulo sobre la programación estructurada 6 en la página 65 escribiremos
programas que usan las colecciones de manera práctica.

7.1. Lista "list"

7.1.1. Definición de Tipo Lista

Recuerde los tipos de objetos explicados en la sección 4.4 en la página 49.

Definición de la Lista

Lista ("list") es un **tipo de dato** en Python
Lista es una **colección ordenada** de objetos, **de posiblemente diferentes tipos**.
Lista es un **objeto mutable**; se le pueden reordenar y modificar sus ítems (elementos).
Lista es un **tipo de dato dinámico**; se le pueden agregar, o eliminar los ítems.

Una lista se puede definir usando los corchetes, [ ],
dentro de los cuales se proporcionan los ítems, separados por comas.

Siguen algunos ejemplos de listas:

```
ficha = ["Juan Perezoso", 32, 25349.50]
```

La "ficha" es una lista de tres objetos: cadena "Juan Perezoso", entero 32 y flotante 25349.50.

```
lista_1 = [0, 1, 2, 3]
```

La "lista_1" es una lista de cuatro ítems: enteros de cero a tres.

```
a = []
```

La "a" es una lista vacía. Por el momento no contiene ningún ítem.

Una lista puede tener como ítems los objetos de cualquier tipo, incluso otras listas:

```
lista = [ficha, lista_1, a, ["yo", "I", "Eu", "Je", "Io", "Ja"]]
```

Las listas que contienen listas u otras colecciones como: tuplas, diccionarios y conjuntos,
las estudiaremos en la sección 7.1.8 en la página 110.
Por ahora nos dedicaremos a listas que tienen ítems de tipo inmutable:
None, lógicos, numéricos y cadenas.

Lista es Un Objeto

Aunque es una colección de objetos, una lista es un "tipo de dato", o sea objeto en sí,
ya que se puede manejar, o sea referenciar y procesar como un todo.
Por ejemplo, podemos imprimir una lista como un objeto, sin tener que hacerlo ítem por ítem.

```
>>> ficha = ["Juan Perezoso", 32, 25349.50]
>>> print(ficha)
['Juan Perezoso', 32, 25349.5]
```

Se dice que lista es una "colección ordenada", porque al cambiar aunque sea un ítem de su lugar,
para el intérprete de Python, esto ya no es la misma lista:

```
>>> [0, 1, 2] == [1, 0, 2]
False
```

En continuación estudiaremos lo que se pueda hacer con las listas como objetos mutables y dinámicos.

Conversión Entre Lista y Cadena

La función "list" (lista) convierte otro tipo de colección de objetos a una lista.
El siguiente ejemplo convierte la cadena, el cuál es una colección de caracteres, a una lista:

```
>>> cadena = "Soy una cadena."
>>> list(cadena)
['S', 'o', 'y', ' ', 'u', 'n', 'a', ' ', 'c', 'a', 'd', 'e', 'n', 'a', '.']
```

La conversión de las cadenas a las listas y vice versa es muy común y existen métodos para esto.

Cada objeto tipo cadena tiene el método "split" (separar) para convertirse a una lista.
La ventaja es que podemos especificar el sub-cadena, o carácter, que sirve como el separador.
Por ejemplo, para separar cada palabra usamos el espacio como el separador:

```
>>> cadena = "Soy una cadena."
>>> cadena.split(" ")
['Soy', 'una', 'cadena.']
```

La sub-cadena que sirvió como el separador quedó eliminado, no aparece en la lista.

Para convertir una lista de cadenas a una sola cadena se usa el método "join" (juntar).
Se especifica la sub-cadena que se va a insertar entre los ítems de la lista.

```
>>> lista = ["un", "dos", "tres", "cuatro"]
>>> cadena = " -> ".join(lista)
>>> cadena
'un -> dos -> tres -> cuatro'
```

La Función "map"

La función "map" aplica alguna otra función a cada elemento de la lista:

```
map(<función>, <lista>)
```

Por ejemplo si tenemos una lista de valores numéricos, podemos convertirlos a cadenas:

```
>>> lista_n = [1, 2, 3, 4]
>>> lista_c = list(map(str, lista_n))
>>> lista_c
['1', '2', '3', '4']
```

Podemos aprovechar esto para convertir las listas de números a una cadena:

```
>>> lista_n = [1, 2, 3, 4]
>>> cadena = ", ".join(map(str, lista_n))
>>> cadena
'1, 2, 3, 4'
```

7.1.2. Creación de Una Lista Nueva

Lista Nueva Con Ítems Especificados

Podemos crear una lista usando los corchetes y dentro de ellos,
especificar una lista de los ítems explícitamente:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La lista se asigna a alguna variable para poder referirse a ella más fácil.

De hecho, así es como empezamos el tema de las listas, creando algunos ejemplos:

```
ficha = ["Juan Perezoso", 32, 25349.50]
lista_1 = [0, 1, 2, 3]
a = []
lista = [ficha, lista_1, a, ["yo", "I", "Eu", "Je", "Io", "Ja"]]
```

Lista Nueva Vacía

En programas seguido empezamos con una lista vacía y
la llenamos con objetos durante la ejecución del programa.
Una lista vacía se puede crear de las siguientes maneras:

```
lista = list()
lista = []
```

Estas dos asignaciones son equivalentes; el nombre "lista" referencia una lista sin ítems.

Lista con Longitud Predeterminada

Al estar agregando los elementos a la lista, la longitud de la lista crece.
La longitud de una lista es el número de ítems en la lista. La lista vacía tiene longitud cero.

Cuando se crea una lista, el intérprete automáticamente deja espacio extra en la memoria
para agregar algunos ítems adicionales si el programa se lo pide.

Si el espacio apartado para la lista está lleno y agregamos más ítems, entonces
el interprete tiene que crear una nueva lista, apartando más memoria.
La lista existente se tiene que copiar a este nuevo lugar más grande.
Esto consume un tiempo extra.

Para evitar copiar la lista una y otra vez con el aumento de su longitud,
si sabemos cuantos ítems vamos a manejar, de una vez creamos la lista de tal longitud.
Por ejemplo, si sabemos que necesitaremos lista de 10 ítems podemos crearla así:

```
LONGITUD = 10
lista = [None] * LONGITUD
lista = [0] * LONGITUD
lista = [1] * LONGITUD
FACTOR = 5
lista = [0, 1] * FACTOR
```

Multiplicar una lista por un entero, igual como en caso de las cadenas,
concatena la lista a sí misma dada cantidad de veces.

Lista Nueva Usando el Iterable "range"

Vamos a estudiar los iterables más adelante en detalles.
Por ahora solamente necesitamos entender que los iterables son
objetos que crean secuencias de ítems.

El iterable "range" se usa para crear una lista de siguiente manera:

```
<lista_nueva> = list(range(<INI>, <FIN>, <PASO>))
```

Los valores INI, FIN y PASO tienen el siguiente significado:

INI el primer valor incluido.

FIN el valor que no está incluido.

PASO el paso de avance.

```
>>> lista = list(range(2, 6))
>>> lista
[2, 3, 4, 5]
```

Algunos ejemplos adicionales:

```
>>> list(range(1, 3))
[1, 2]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
```

Ejemplos que usan el PASO:

```
>>> list(range(0, 9, 3))
[0, 3, 6]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(9, 0, -3))
[9, 6, 3]
>>> list(range(9, -1, -3))
[9, 6, 3, 0]
```

Entren al modo interactivo de Python, reproduzcan estos ejemplos e intenten algunos nuevos.

Lista Nueva de Listas Existentes

Se puede formar una lista nueva sumando listas existentes.

```
>>> lista_1 = [0, 1]
>>> lista_2 = ['a', 'b', 'c']
>>> lista_3 = [0.0, 1.0]
>>> lista = lista_1 + lista_2 + lista_3
>>> lista
[0, 1, 'a', 'b', 'c', 0.0, 1.0]
```

Extrayendo sublistas

Se puede formar una lista nueva, extrayendo sub-listas de una lista existente.
Para esto se usa la forma:

```
<lista_nueva> = <lista_existente>[<INI>:<FIN>]
<lista_nueva> = <lista_existente>[<INI>:>FIN>:<PASO>]
```

Se pueden usar los índices positivos, o negativos para trabajar con los lugares en la lista.
Igual como en caso de las cadenas de caracteres:
El primer ítem en la lista es siempre referenciado por el índice 0 y los demás: 1, 2, 3, ...
El último ítem en la lista es siempre referenciado por el índice -1 y los anteriores: -2, -3, ...

Los valores INI, FIN y PASO tienen el siguiente significado:

INI el primer lugar incluido.

FIN el lugar que no está incluido.

PASO el paso de avance.

Es mejor aprender de ejemplos:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista = L_09[2:6]
>>> lista
[2, 3, 4, 5]
```

En la expresión [2:6] el 2 es el primer lugar incluido y el 6 es el lugar no incluido.

Otros ejemplos:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L_09[1:3]
[1, 2]
>>> L_09[5:]
[5, 6, 7, 8, 9]
>>> L_09[:5]
[0, 1, 2, 3, 4]
```

Si el FIN no está especificado, quiere decir: hasta el final de la lista.
Si el INI no está especificado, quiere decir: desde el inicio de la lista.

Con el PASO podemos brincar algunos ítems regularmente:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L_09[0:9:3]
[0, 3, 6]
>>> L_09[::3]
[0, 3, 6, 9]
```

Usando el PASO negativo podemos invertir el orden de leer la lista:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> L_09[9:0:-3]
[9, 6, 3]
>>> L_09[::-3]
[9, 6, 3, 0]
```

Analicen este ejemplo.

Listas Invertidas y Ordenadas

Para invertir una lista se puede usar el PASO con valor -1,

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista_invertida = L_09[::-1]
>>> lista_invertida
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

también puede usar el iterador "reversed" (invertida).
El iterador lee la lista y regresa los ítem uno por uno, desde el final hacia el inicio de la lista.
El iterador "reversed" no crea una lista, tenemos usar el convertidor "list" para crear la lista.

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> lista_invertida = list(reversed(L_09))
>>> lista_invertida
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Si deseamos que la nueva lista sea la forma ordenada de una lista existente,
usamos la función "sorted" (ordenada).

```
>>> lista_desordenada = [0, 9, 1, 8, 2, 7, 3, 6, 4, 5]
>>> lista_ordenada = sorted(lista_desordenada)
>>> lista_ordenada
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

La función "sorted" produce una nueva lista con los ítems ordenados desde menor hacia mayor.
Podemos invertir la lista usando el parámetro "reverse" y asignándole el valor True.

```
>>> lista_desordenada = [0, 9, 1, 8, 2, 7, 3, 6, 4, 5]
>>> lista_descendiente = sorted(lista_desordenada, reverse=True)
>>> lista_descendiente
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

7.1.3. Copiar Una Lista

Copiar una lista no es muy intuitivo cuando empezamos a hacerlo en Python.
Por lo tanto le dedicaremos a esta técnica toda esta subsección. Pongan atención.

Si tenemos una lista "a" y deseamos copiarla para formar una nueva lista "b",
la cuál tiene los mismos ítems que la lista "a", NO podemos hacer: "b = a".

Veamos que pasa en realidad:

```
>>> a = [0, 1, 2, 3]
>>> b = a
```

Después de asignar "b = a" las listas son iguales:

```
>>> b == a
True
```

PERO, la lista "b" no es una copia de la lista "a".
La variable "b", después de "b = a" simplemente referencia la misma lista "a".
Podemos verificarlo comparando los identificadores de los domicilios referenciados en la memoria:

```
>>> id(b) == id(a)     # Equivalente a: b is a
True
```

Se trata de solamente una lista, referenciada por dos nombres distintos.

Entonces, ¿cómo hacer una copia de la lista? Existen varias maneras.

Las siguientes dos maneras de crear la copia de una lista
son aplicables cuando la lista no contiene ningún ítem que sea una colección mutable.
Las colecciones mutables incorporadas son: lista, tupla, diccionario, y conjunto.
Para tales casos vea la sección 7.1.8 en la página 110.

Copiar Lista Usando la Operación Sublista

La más sencilla es usando la técnica de sublista que ya aprendimos.
podemos extraer una sublista la cuál de hecho es la lista completa:

```
>>> a = [0, 1, 2, 3]
>>> b = a[:]
>>> b == a
True
>>> id(b) == id(a)
False
```

Esta vez el intérprete sí hizo una copia. El "b" es igual al "a" pero identificadores son distintos, o sea, las listas "a" y "b" son idénticas, y son dos listas distintas, como deseábamos.

Copiar Lista Usando la Función "list"

Para crear una lista nueva idéntica a otra existente, se puede usar la función que convierte otros tipos de colecciones e iteradores en una lista.
Esta función también convierte una lista a tipo lista, creando una lista nueva.

```
<lista_nueva> = list(<lista_existente>)
```

Por ejemplo

```
>>> a = [0, 1, 2, 3]
>>> b = list(a)
>>> b == a
True
>>> b is a
False
```

7.1.4. Funciones Incorporadas Aplicables a Listas

Entre las funciones integradas de Python existen varias que son relacionadas con las colecciones. Vamos a ver su uso en el ejemplo de listas de objetos inmutables.

Todas estas funciones se aplican en forma:

```
<función>(<lista>)
```

Ya vimos las funciones "reversed" y "sorted" en esta sección. Seguimos con más funciones.

La Longitud de Una Lista

La longitud de una lista se puede determinar usando la función "len" ("length", longitud).
La lista vacía tiene la longitud cero, la lista de un ítem tiene longitud uno y así sucesivamente.

```
>>> len([])
0
>>> len([0])
1
>>> len([0, 1])
2
```

```
>>> lista = [0, 1, 2, 3, 4, 5]
>>> len(lista)
6
```

Si nos interesa específicamente si la lista es vacía o no, podemos usar la conversión de tipo lista a tipo lógico. Una lista vacía se convierte en False y una lista no vacía se convierte en True.

```
>>> a = []
>>> bool(a)
False
>>> b = [0]
>>> bool(b)
True
```

Las Características de la Lista

Si todos los elementos de la lista son de tipo numérico: entero y/o flotante,
podemos usar las funciones "min" y "max" para identificar el ítem mínimo, o máximo.
Calculando la diferencia, podemos obtener el rango de los datos.

```
>>> a = [-1.0, 0, 2.5, -2]
>>> min(a)
-2
>>> max(a)
2.5
>>>
>>> rango = max(a) - min(a)
>>> rango
4.5
```

También podemos usar "sum" para calcular la suma de elementos.
Si dividimos la suma con la cantidad de números obtenemos el promedio (media).

```
>>> a = [-1.0, 0, 2.5, -2]
>>> sum(a)
-0.5
>>> media = sum(a) / len(a)
>>> media
-0.125
```

La lista con todos ítems cero es frecuentemente usada. Juega el papel de cero entre listas.
Se puede detectar si alguno de los ítems es cero "any" , o si todos los ítems son cero "all".

```
>>> a = [-1.0, 0, 2.5, -2]
>>> any(a)
True
>>> all(a)
False
```

7.1.5. Lista es Objeto Dinámico

Objetos dinámicos son aquellos cuyos elementos se pueden
agregar o eliminar durante la ejecución de programa.
Para el intérprete lo más importante de un objeto dinámico es que tal objeto puede
cambiar la cantidad de memoria que va a ocupar,
mientras los objetos estáticos necesitan la constante cantidad de memoria durante su "vida".

Simplemente dicho, lista es un objeto dinámico porque podemos agregar y eliminar sus ítems.

Vamos a conocer varios métodos de objeto tipo "lista". Los métodos se aplican en forma:

```
<lista>.<método>([argumentos])
```

Los símbolos < y > encierran las partes que se deben de sustituir.
Los símbolos [y] encierran las partes opcionales.
El punto y los paréntesis son obligatorios tal cual.
Quedará claro después de ver algunos ejemplos.

Agregando Ítems a Una Lista

Para agregar un nuevo ítem a la lista se usa el método "append".

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.append(3.0)
>>> a
[-1.0, 0, 2.5, -2, 3.0]
```

Si deseamos que el nuevo ítem esté en algún lugar particular y no en le fin de la lista,
usamos el método "insert" (insertar), al cual se le especifica el lugar y el ítem.

Para insertar un nuevo ítem de manera que sea el primero de la lista, se usa le índice cero.

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.insert(0, 3.0)
>>> a
[3.0, -1.0, 0, 2.5, -2]
```

Si deseamos agregar más de un ítem, o sea deseamos agregar una lista,
se usa el método "extend" (extender):

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.extend([3.0, 4.0, 5.0])
>>> a
[-1.0, 0, 2.5, -2, 3.0, 4.0, 5.0]
```

Eliminando Ítems de Una Lista

Existen dos maneras de eliminar un ítem de la lista, dependiendo como vamos a identificar este ítem.

El método "remove" (remover) elimina el primer ítem encontrado que tenga un valor dado:

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.remove(0)
>>> a [-1.0, 2.5, -2]
```

El método "pop" (extraer) elimina el ítem a base de su posición en la lista y
devuelve el valor que estaba en la posición especificada.
Si al "pop" no le especificamos la posición, elimina el último ítem; el de la posición -1.

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.pop()
-2
>>> a.pop(1)
0
>>> a
[-1.0, 2.5]
```

Si necesitamos borrar todos los ítems de la lista, esto se logra con el método "clear" (despeja).
Al aplicar este método la lista se queda vacía.

```
>>> a = [-1.0, 0, 2.5, -2]
>>> a.clear()
>>> a
[]
```

Aparte de las funciones y los métodos usados para manejar las listas,
también se puede usar la declaración "del" ("delete", borrar) de Python,
para eliminar algunas partes de la lista, o para eliminar el objeto lista.

Para eliminar solamente un elemento, digamos el último de la lista, podemos hacer lo siguiente

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L_09[-1]
>>> L_09
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

A diferencia de "pop", la declaración "del" simplemente elimina el ítem, no nos devuelve su valor.
El "pop" lo usamos para procesar un ítem y eliminarlo de la lista porque ya está procesado.
El "del" lo usamos para simplemente descartar un ítem.

Con la declaración "del" también podemos eliminar una sublista:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L_09[2:6]
>>> L_09
[0, 1, 6, 7, 8, 9]
```

En la expresión "del l_09[2:6]", el 2 es el primer lugar eliminado y el 6 es el primero que queda.

Al aplicar la declaración "del" a la lista, sin especificar índice o sublista, se elimina el objeto lista:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L_09
>>> L_09
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'l_09' is not defined
```

La excepción es levantada porque se pide el valor de "l_09", el objeto que ya no existe, y por lo tanto el nombre "l_09" ya no está definido.

A diferencia del método "clear" el cuál elimina todos los ítems de la lista, la declaración "del <lista>" elimina el mismo objeto lista de la memoria. Los ítems quedan eliminados también en caso de que ninguna otra lista los esté referenciando.

Si deseamos usar "del" para lograr el efecto de "clear", podemos hacerlo así:

```
>>> L_09 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> del L_09[:]
>>> L_09
[]
```

El objeto lista sobrevivió, pero quedó vacío.

Lista es Objeto Mutable

Lista es un objeto mutable, lo que quiere decir que lo podemos modificar "en su lugar".

La modificación "en su lugar" es aquella que no está creando un objeto nuevo, modifica el objeto existente en el mismo lugar de la memoria donde ya se encuentra.

Reordenando los Ítems

Por ejemplo ya vimos la función "sorted" en esta sección. Esta función crea una lista ordenada nueva. Si deseamos ordenar una lista, pero sin crear una lista nueva, usamos el método "sort" (ordenar).

```
>>> lista = [0, 9, 1, 8, 2, 7, 3, 6, 4, 5]
>>> domicilio = id(lista)
>>> lista.sort()
>>> lista
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> id(lista) == domicilio
True
```

Como podemos ver, la lista ordenada se encuentra en el mismo domicilio de la lista original; fue ordenada "en su lugar".

Si deseamos que la lista esté ordenada desde ítem mayor hacia el menor, podemos invertir la lista "en su lugar" asignando True al parámetro "reverse" (invierte).

```
>>> lista = [0, 9, 1, 8, 2, 7, 3, 6, 4, 5]
>>> lista.sort(reverse=True)
>>> lista
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Si deseamos invertir la lista, sin ordenarla podemos usar el método "reverse".

```
>>> lista = [0, 9, 1, 8, 2, 7, 3, 6, 4, 5]
>>> lista.reverse()
>>> lista
[5, 4, 6, 3, 7, 2, 8, 1, 9, 0]
```

Modificando los Ítems y las Sublistas

Lista, como un objeto mutable permite reordenamiento de sus ítems, y
también permite cambiar el valor de algún ítem especifico, o alguna sublista especifica.

Para modificar un ítem usamos usando el índice de su lugar dentro de los corchetes [].

```
>>> L_08 = [0, 2, 4, 6, 8]
>>> L_08[3] = 13
>>> L_08
[0, 2, 4, 13, 8]
```

El ítem en lugar número 3 fue modificado a valor 13.

También podemos modificar una sublista.
La sublista nueva puede tener longitud distinta de la modificada.

```
>>> L_08 = [0, 2, 4, 6, 8]
>>> L_08[1:4] = [1, 2, 3, 4, 5, 6, 7]
>>> L_08
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

La expresión "L_08[1:4]" se refiere a la sublista de longitud 3, que consiste de lugares 1, 2 y 3.
Los valores en estos lugares son: 2, 4 y 6. Esta sublista se elimina de la lista "L_08" y en su lugar se inserta la lista [1, 2, 3, 4, 5, 6, 7], la cuál tiene longitud 7.

7.1.6. Procesamiento de las Listas

Búsqueda de los Ítems

Bueno, todo este tiempo, trabajando con las listas, hemos trabajado con los ítems.
Solamente nos falta ver algunos métodos específicos para el trabajo con los ítems.

Inclusión

Seguido necesitamos saber si algún dato esta en la lista como su ítem, o no.
Para averiguar esto usamos **operadores de inclusión** "**in**" (en) y "**not in**" (no en).

Las formas en que se usan estos operadores es:

```
<objeto> in <colección>
<objeto> not in <colección>
```

Por ejemplo:

```
>>> lista = [2, 3, 5, 7, 11, 13, 17, 19]
>>> 4 in lista
False
>>> 5 in lista
True
>>> 6 not in lista
True
>>> 7 not in lista
False
```

El resultado de estos operadores es uno de dos valores lógicos: False o True.

Lugar (índice)

A veces queremos saber más. Queremos saber ¿en que lugar está el objeto de interes?
Deseamos conocer su índice. Para esto se usa el método "index" (índice).

Por ejemplo:

```
>>> lista = [2, 3, 5, 7, 11, 13, 17, 19]
>>> lista.index(5)
2
>>> lista.index(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 6 is not in list
```

Primero le preguntamos a la lista ¿en cuál lugar está el objeto entero 5?
La respuesta fue: en el lugar número 2.

Después preguntamos la lista: ¿en cuál lugar está el objeto entero 6?
La respuesta fue: "ValueError" (error de valor), el 6 no está en la lista.

Después aprenderemos a manejar las excepciones. Por ahora, antes de pedir el índice de un objeto, usaremos el operador "in", o el operador "not in", para verificar si el objeto está en la lista.

El método "index" devuelve el índice del primer lugar en el que encuentre el objeto buscado.
No nos dice si el objeto aparece múltiples veces, ni en cuales lugares. Por ejemplo:

```
>>> lista = ["a", "b", "c", "a", "b", "c", "a", "b", "c"]
>>> lista.index("b")
1
```

Solamente sabemos que hay un carácter "b" en el lugar número 1. No sabemos si los hay más.
La primera ayuda con este problema es que el método "index" tiene dos formas más:

```
<lista>.index(<objeto>, <INI>)
<lista>.index(<objeto>, <INI>, <FIN>)
```

En el siguiente ejemplo preguntamos la lista si tiene el carácter "b" en lugar 3, o después.
No nos interesan los lugares 0, 1, y 2.

```
>>> lista = ["a", "b", "c", "a", "b", "c", "a", "b", "c"]
>>> lista.index("b", 3)
4
```

La lista responde: "ignorando lugares antes del tercero, el carácter "b" aparece en lugar 4".

Esta vez le preguntaremos la lista si tiene el carácter "b" en lugares 5, o 6.

```
>>> lista = ["a", "b", "c", "a", "b", "c", "a", "b", "c"]
>>> lista.index("b", 5, 7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'b' is not in list
```

La lista responde que no existe carácter "b" en la sublista que consiste de lugares 5 y 6.

Conteo

Dentro de la búsqueda de los objetos en las listas, también podemos averiguar
cuantas veces aparece un objeto en la lista. Para esto usamos el método "count" (conteo).

```
>>> lista = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> lista.count(2)
3
```

La lista responde que el entero 2 aparece tres veces entre sus ítems.

Para tratar la búsqueda de ítems como debe hacerse, necesitamos ramificaciones y ciclos.
Los vamos a aprender en los siguientes capítulos.

7.1.7. Lista y Ciclos

ESCRIBIR !!!

7.1.8. Lista de Varias Dimensiones

7.2. Tupla ("tuple")

7.2.1. Definición de Tipo Tupla

Recuerde los tipos de objetos explicados en la sección 4.4 en la página 49.

Tupla ("tuple") es un **tipo de dato** en Python
Tupla es una **colección ordenada** de objetos, **de posiblemente diferentes tipos**.
Tupla es un **objeto inmutable**; no se le pueden reordenar ni modificar sus ítems (elementos).
Tupla es un **tipo de dato no-dinámico**; no no se le pueden agregar, o eliminar los ítems.

Tupla es prácticamente lo mismo que lista, excepto que tiene sintaxis un poco distinta y
no se le puede modificar "en su lugar", ni a sus ítems tampoco.
Cualquier intento de modificación crea una tupla nueva.

Tupla, ente las listas, es como una constante entre las variables.

Una lista se le puede modificar de cuatro maneras:
reordenando los ítems
agregando los ítems
borrando los ítems
modificando el valor de uno, o varios ítems.

Si necesitamos una lista en la cual no se va a hacer ninguna de estas modificaciones,
preferimos usar una tupla en su lugar.

Las tuplas son la opción preferida porque el intérprete es más rápido con las tuplas que con las listas.

La Sintaxis de la Tupla

Una tupla es una secuencia de objetos, de mismos o distintos tipos, separados por las comas:

```
tupla_1 = "Juan Perezoso", 32, 25349.50
tupla_2 = 0, 1, 2, 3
tupla_3 = “a”,
tupla_4 = 8.20,
```

Tupla de un ítem, de longitud 1, tiene que tener una coma, de otra manera sería un escalar.
Sin una coma, los últimas dos líneas serían: una cadena de una letra y un flotante:

```
cadena = “a”
flotante = 8.20
```

El Python escribe cada tupla siempre dentro de los paréntesis para facilitar su identificación:

```
>>> tupla = 0, 1, 2, 3
>>> tupla
(0, 1, 2, 3)
```

No son los paréntesis lo que define una tupla; es la coma:

```
>>> a = (1)
>>> type(a)
<class 'int'>
>>>
>>> a = 1,                          >>> a = (1,)v
>>> type(a)                         >>> type(a)
<class 'tuple'>                     <class 'tuple'>
```

En este último ejemplo, el interprete dice: el “(1)” es un entero, y el “1,” es una tupla.
Y si usamos el “(1,)” es una tupla. No porque tiene los paréntesis, sino porque tiene la coma.

En algunas situaciones los paréntesis que delimitan la tupla son necesarios.
Por ejemplo, si usamos varias tuplas, o si tenemos una tupla dentro de alguna secuencia:

```
lista_1 = [0, 1, 2]          # Tres enteros
lista_2 = [0, (1, 2)]        # Un entero y una tupla
lista_3 = [(0,) (1, 2)]      # Dos tuplas
```

En la asignación, es costumbre de escribir
la tupla a la izquierda de = sin los paréntesis y
la tupla a la derecha de = con los paréntesis:

```
x, y = (1, 2)
```

Tupla puede contener otras tuplas y así recursivamente:

```
tupla = ((1, 2), 3, (“a”, “b”, “c”, (False, True)))
```

Esta tupla contiene una tupla (1, 2), un entero 3, y otra tupla (“a”, “b”, “c”, (False, True)).
La otra tupla contiene tres cadenas y la tupla (False, True).

Las tuplas pueden contener inclusive los objetos mutables, como listas:

```
tupla = ((1, 2), 3, [“a”, “b”, “c”])
```

Esta tupla contiene una tupla (1, 2), un entero 3, y una lista [“a”, “b”, “c”].

Conversión entre una tupla y una lista

Una Lista se puede convertir en una tupla usando la función de conversión “tuple”.
Y se puede usar la función de conversión “list” para convertir una tupla en una lista.

```
>>> lista = ["Juan Perezoso", 32, 25349.50]
>>> tupla = tuple(lista)
>>> tupla
('Juan Perezoso', 32, 25349.5)
>>>
>>> lista = list(tupla)
>>> lista
['Juan Perezoso', 32, 25349.5]
```

7.2.2. Los Usos de la Tupla

Tupla es una colección estática inmutable. Se usa como una lista constante.
Tupla no es dinámica; no se le pueden agregar ni eliminar ítems.
Tupla no es mutable; no se le pueden reordenar ni modificar sus ítems.

Un uso de la tupla es para que sirva como llave en un diccionario.
Tal tupla no puede contener objetos mutables, como una lista, por ejemplo.
Este uso lo vamos a estudiar en este capitulo al estudiar los diccionarios.

Otro uso de la tupla es para pasar argumentos a las funciones y
para recibir múltiples objetos regresados de las funciones.
Esto lo vamos a aprender al estudiar las funciones.

El siguiente uso de la tupla es para intercambiar los valores de dos, o más, variables:

```
x, y = (y, x)
a, b, c = (c, a, b)
```

En muchos otros lenguajes, la única manera de intercambio es usando una variable temporal:

```
tmp = x
x = y
y = tmp
```

El uso adicional de la tupla es para agrupar una secuencia de asignaciones en una sola línea.
Se recomienda hacerlo sólo cuando los objetos asignados son de alguna manera relacionados:

```
nombre = "Juan Perezoso"
edad = 32
ahorro = 25349.50
```

se puede escribir como:

```
nombre, edad, ahorro = ("Juan Perezoso", 32, 25349.50)
```

Y como se ha dicho, siendo las operaciones con tuplas más rápidas que las operaciones con listas, usamos tuplas en lugar de las listas, cada ves que la lista va a permanecer constante.

7.2.3. Tuplas Mapeables y No-Mapeables

El Valor de Tupla Puede Cambiar

Una tupla no contiene los valores de sus ítems, más bien es una secuencia de referencias a los ítems.

Por ejemplo,

```
>>> t = 1, 2
>>> e = 3
>>> l = ["a", "b", "c"]
>>> tupla = t, e, l
```

es lo mismo que

```
>>> tupla = ((1, 2), 3, ["a", "b", "c"])
```

solamente que en el ultimo caso el intérprete asigna referencias a los ítems automáticamente.

Cuando se dice que una tupla es inmutable,
esto quiere decir que no va a reordenar, ni modificar las referencias a los ítems que contiene.

En cuanto a sus ítems inmutables, esto implica que los valores de estos ítems tampoco cambiarán, ya que al cambiar su valor, también cambiaría su domicilio en la memoria y
cambiarían las referencias hacia ellos en la tupla.

Pero, es posible que un ítem mutable cambie su valor sin que la tupla sepa de esto.
Por ejemplo,

```
>>> lista = ["a", "b", "c"]
>>> tupla = (len(lista), lista)
>>> tupla
(3, ['a', 'b', 'c'])
>>> lista.append("d")
>>> tupla
(3, ['a', 'b', 'c', 'd'])
```

La tupla no puede prevenir los cambios en su ítem mutable, en este caso de la lista, ya que la tupla no sabe nada de la lista, excepto que tiene la referencia a ella.

Por esto dividimos las tuplas en **mapeables** "hashable", las que no contienen ítems mutables, y las tuplas **no-mapeables** "not hashable", las que contienen ítems mutables que puedan cambiar.

La razón de permitir que las tuplas siendo inmutables puedan contener ítems mutables y dinámicos, es que uno de los usos de las tuplas es para comunicar secuencia de argumentos a las funciones. A veces algunos de los argumentos necesitan ser mutables para que la función cumpla su propósito.

El uso principal de la tupla, hacer posible que una secuencia sea la llave en un diccionario, se limita solamente a las tuplas mapeables.

La Inmutabilidad de la Tupla

La tupla, siendo una lista estática e inmutable, no tiene métodos que la modifican: "append", "insert", "extend", "remove", "pop", "clear", "sort", "reverse".

Tampoco se le puede reasignar valor a un ítem, o una sub-tupla, usando índices. Los [i] y los [i:j] e [i:j:k] no funcionan de lado izquierdo de la asignación, para modificar la tupla:

```
tupla = (0, 1, 2, 3, 4, 5)
# No funciona:
tupla[5] = 1
tupla[1:4] = (11, 12, 13)
```

Pero, sí funcionan de lado derecho de la asignación, para copiar las partes de una tupla a otras variables:

```
# Sí, funciona:
tupla = (0, 1, 2, 3, 4, 5)
x = tupla[5]
sub_tupla = tupla[0:-1:2]
```

No se pueden eliminar los ítems de la tupla usando la declaración "del":

```
tupla = (0, 1, 2, 3, 4, 5)
# No funciona:
del tupla[5]
del tupla[1:4]
```

Pero, sí se puede eliminar la tupla misma:

```
tupla = (0, 1, 2, 3, 4, 5)
# Sí, funciona:
del tupla
```

7.2.4. Operaciones, Funciones y Métodos de Tuplas

Acabamos de ver cuales operadores, funciones y métodos no funcionan sobre las Tuplas. Ahora vamos a ver los que sí funcionan.

Los operadores, funciones y métodos de las listas que no las modifican, funcionan también sobre las tuplas.

Conversión de Tupla a Cadena

```
>>> tupla = ('Hola,', '¿Cómo', 'está', 'usted?')
>>> oración = " ".join(tupla)
>>> oración
'Hola, ¿Cómo está usted?'
```

Operadores de Multiplicación * y Suma +

La operación de multiplicación de una tupla con entero, concatena la tupla sobre si misma, dado número de veces.

```
>>> tupla = (0, 1, 2) * 3
>>> tupla
(0, 1, 2, 0, 1, 2, 0, 1, 2)
```

En la operación de asignación, la expresión a la derecha resulta en una tupla y la variable "tupla" se asigna a esta tupla.

La operación de la suma concatena las tuplas.

```
>>> tupla_1 = ("Hola,", "¿Cómo", "está")
>>> tupla_2 = tupla_1 + ("usted?",)
>>> tupla_2
('Hola,', '¿Cómo', 'está', 'usted?')
```

Las Funciones "min", "max", "sum", "len", "any", "all"

Para las tuplas que contienen datos numéricos podemos obtener el rango de datos.

```
>>> tupla = -1, 0, 1, 2
>>> min(tupla)
-1
>>>
max(tupla)
2
>>> rango = max(tupla) - min(tupla)
>>> rango
3
```

Para las tuplas que contienen datos numéricos podemos obtener la media de los datos.

```
>>> tupla = (-1, 0, 1, 2)
>>> sum(tupla)
2
>>> len(tupla)
4
>>> media = sum(tupla) / len(tupla)
>>> media
0.5
```

Seguido deseamos saber si algunos o todos ítems de una tupla son ceros.

```
>>> tupla = -1, 0, 1, 2
>>> any(tupla)
True
>>> all(tupla)
False
```

Vamos a seguir usando las tuplas durante el resto de libro y
aprenderemos sus usos de muchos ejemplos

7.3. Conjunto ("set")

7.3.1. Definición de Tipo Conjunto

Recuerde los tipos de objetos explicados en la sección 4.4 en la página 49.

La colección "conjunto" ("set") es tipo de dato que se usa para modelar los conjuntos matemáticos.

Esta colección la usamos cuando vamos a necesitar la relación de subconjunto y/o algunas de las operaciones de conjuntos: unión, intersección, diferencia, diferencia simétrica, entre otras.

El conjunto es un tipo de dato dinámico, mutable y no-mapeable.

El conjunto "set" es una colección de objetos **mapeables**, de **mismo o diferentes tipos**, donde: **no existe orden** de los objetos y **no existe repetición** de los objetos.
Solamente nos interesa cuáles objetos (elementos, ítems) están en el conjunto y cuáles no.

7.3.2. Creación de Un Conjunto

El conjunto **vació** se puede crear solamente de una manera:
usando la función de conversión al conjunto, "set", sin argumentos:

```
<nombre_del_conjunto> = set()
```

Por ejemplo:

```
>>> conjunto = set()
>>> conjunto
set()
```

Un conjunto **no vacío** se puede crear usando su representación literal:
secuencia de objetos entre llaves: **{ }.**

```
>>> z = {"a", "b", "c"}
>>> z
{'a', 'b', 'c'}
```

No se puede crear el conjunto vacío usando llaves, ya que se crearía un diccionario.
La expresión literal de los diccionarios también usa las llaves, solamente que
en lugar de contener secuencia de ítems, contiene secuencia de pares: "<llave>: <valor>".
Lo veremos en la sección 7.4 en la página 122.

Conversión de Colección a Conjunto

Como ya mencionamos, la función "set" (conjunto) convierte un iterable, o una colección a conjunto, si esta colección es mapeable. De otra manera se levanta la excepción "TypeError".

Este ejemplo está usando el iterable "range".

```
>>> v = set(range(1, 10, 2))
>>> v
{1, 9, 3, 5, 7}
```

En este ejemplo se puede ver que Python representa los elementos del conjunto en cualquier orden que quiera, ya que en un conjunto no hay orden.

El siguiente ejemplo está usando las colecciones:

```
>>> w = set("abc")
>>> w
{'a', 'b', 'c'}
>>> x = set(["a", "b", "c"])
>>> x
{'a', 'b', 'c'}
>>> y = set(("a", "b", "c"))
>>> y
{'a', 'b', 'c'}
```

Observen que Python representa un conjunto como una secuencia de ítems entre llaves { }. Esta es la **representación literal** de un conjunto.

La función de conversión a conjunto acepta solamente un argumento, así que la tupla tiene que estar escrita dentro de los paréntesis. Lo siguiente **no funciona**:

```
>>> w = set("a", "b", "c")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set expected at most 1 arguments, got 3
```

7.3.3. Conjunto Es Dinámico

Siendo un conjunto dinámico, podemos agregar y eliminar sus ítems.

Para agregar un ítem a conjunto dado, tenemos el método:

```
add
```

Por ejemplo:

```
>>> a = {"a", "b"}
>>> a.add("c")
>>> a
{'a', 'b', 'c'}
```

Para modificar un conjunto en relación con un otro, agregando o quitando elementos, tenemos cuatro métodos:

```
# operaciones           # métodos                    # operadores

unión                   update                       |=
intersección            intersection_update          &=
diferencia              difference_update            -=
diferencia simétrica    symmetric_difference_udate   ^=
```

Para eliminar ítems, podemos usar los métodos "pop", "remove", "discard" y "clear".
También se puede usar la declaración "del" para eliminar ítem, o el conjunto entero.

Los métodos tienen ventaja que pueden aceptar cualquier secuencia, no solamente otro conjunto.

```
>>> x = {"e", "f", "g", "h"}
>>> x.update(["a", "e", "i"])    # Actualizado usando un lista
>>> x
{'e', 'g', 'a', 'i', 'h', 'f'}
```

En caso de los operadores, ambos operandos tienen que ser conjuntos:

```
>>> x = {"e", "f", "g", "h"}
>>> x |= {"a", "e", "i"}         # Actualizado usando un conjunto
>>> x
{'e', 'g', 'h', 'a', 'f', 'i'}
```

A veces deseamos procesar ítem por ítem y al procesar cada ítem eliminarlo del conjunto.
Para esto se usa el método "pop" (extraer).

```
>>> x = {"e", "f", "g", "h"}
>>> ítem = x.pop()
>>> ítem
'e'
>>> x
{'f', 'h', 'g'}
```

El orden en el cual "pop" extrae ítems del conjunto no está garantizado.
Si no hay elementos en el conjunto, el método "pop" levanta la excepción "KeyError".

```
>>> z = set()
>>> ítem = z.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

Si deseamos eliminar un ítem específico del conjunto, usamos el método "remove" (remover).
El argumento de "remover" puede ser un subconjunto o solamente un ítem.

```
>>> x = {"e", "f", "g", "h"}
>>> x.remove("f")
>>> x
{'e', 'h', 'g'}
```

El método "remove" levanta la excepción "KeyError" si el ítem no está en el conjunto.
Si no deseamos que esto suceda se usa "discard" (descartar) en lugar de "remove".

```
>>> x = {"e", "f", "g", "h"}
>>> x.discard("d")
>>> x
{'e', 'f', 'h', 'g'}
```

El "discard" no levanta ninguna excepción si el elemento no está en el conjunto.

El método "clear" (despeja) elimina todos los ítems del conjunto, y lo deja vacío:

```
>>> b = set((1, 2, 3))
>>> b.clear()
>>> b
set()
```

Conjunto es Mutable

El conjunto es tipo de dato mutable.
Normalmente a las colecciones mutables se les puede cambiar el orden de los ítem.
Pero, en un conjunto el orden no importa. Solamente nos interesa la pertenencia al conjunto.
Por lo tanto, la función "reversed" no es aplicable a los conjuntos.
La función "sorted" aplica, pero solamente con el propósito de imprimir los ítems en orden.
El "sorted" crea una lista de referencias a los ítems en el conjunto y la ordena.

```
>>> z = {"a", "g", "c", "e"}
>>> z
{'e', 'a', 'g', 'c'}
>>> sorted(z)
['a', 'c', 'e', 'g']
```

Dado que en el conjunto el orden no importa, no se les puede acceder a los ítems por un índice [i].
No existe: el ítem primero, el ítem segundo, ..., el ítem último.

7.3.4. Funciones Incorporadas Aplicables a Conjunto

Para saber la cantidad de elementos en el conjunto se usa la función "len", ("length", longitud).

```
>>> conjunto = {1, 2, 3, 4, 5, 6}
>>> len(conjunto)
6
```

Si los ítems en el conjunto son de tipo numérico podemos usar funciones:
"sum", "max", "min", "any", "all".

Podemos por ejemplo calcular la media de los ítems

```
>>> c = {1, 2, 3, 4, 5, 6}
>>> media = sum(c) / len(c)
>>> media
3.5
```

Podemos calcular el rango de los ítems

```
>>> c = {1, 2, 3, 4, 5, 6}
>>> rango = max(c) - min(c)
>>> rango
5
```

Podemos verificar si algunos "any," o todos "all", ítems tienen valor cero:

```
>>> d = {-1, 0, 1}
>>> any(d)
True
>>> all(d)
False
```

7.3.5. Procesamiento de un Conjunto

Inclusión: "in" y "not in"

Con los operadores "in" y "not in" se puede verificar si un objeto es elemento de un conjunto.

```
>>> 1 in B                      >>> 1 not in B
True                            False
>>> 1 in C                      >>> 1 not in C
False                           True
```

Relaciones: Subconjunto y Super-conjunto

Podemos verificar las siguientes relaciones entre dos conjuntos:

```
                    A == B      A != B

A.issubset(B)       A <= B      A < B

A.issuperset(B)     A >= B      A > B

A.isdisjoint(B)
```

El método "isdisjoint" resulta True si A y B no tienen elementos en común; de otra manera es False. Este método no tiene operador correspondiente.

Ejemplos:

```
>>> B.issubset(D)     >>> D.issuperset(C)     >>> B.isdisjoint(E)
True                  True                    True
>>> B.issubset(C)     >>> E.issuperset(C)     >>> C.isdisjoint(E)
False                 False                   False
```

Ejecute los mismos y más ejemplos usando los operadores: ==, !=, <, <=, >=, >.

Operaciones Sobre Conjuntos

Las operaciones sobre conjuntos se pueden ejecutar como métodos o como operadores:

```
# operaciones           # métodos               # operadores

unión                   union                   |
intersección            intersection            &
diferencia              difference              -
diferencia simétrica    symmetric_difference    ^
```

Usar un método, o su operador correspondiente, no es exactamente lo mismo.
Los métodos pueden usar otro tipo de secuencias, aparte de conjunto.
Los operadores pueden ser usados repetidas veces en la misma expresión:

```
>>> A = {1, 2}; B = {2, 1}; C = {2, 3}; D = {1, 2, 3}; E = {3, 4, 5}
>>>
>>> F = E - C - B
>>> F
{4, 5}
```

Use los conjuntos A, B, C, D y E para probar la variedad de operadores aplicados a ellos.

7.3.6. Congelado "frozenset"

El conjunto "congelado" es un conjunto constante: estático, e inmutable.
En otras palabras: "congelado" es un conjunto **mapeable**.

Que sea **estático** quiere decir que siempre necesita la misma cantidad de memoria;
No crece ni disminuye su tamaño.
Que sea **inmutable** quiere decir que no podemos modificar sus ítems:
ni su orden ni su valor.

El propósito de congelado es usarlo como llave para el diccionario,
ya que solamente los objetos mapeables: inmutables y estáticos pueden ser llaves.
Esto lo estudiaremos en la sección 7.4 en la página siguiente.

Para crear un congelado se usa la conversión "frozenset" de una colección.

```
>>> a = frozenset()
>>> b = frozenset([1, 2, 3])
>>> b
frozenset({1, 2, 3})
```

Lo más usual es convertir un conjunto en congelado (congelar un conjunto),
para que puede ser usado como llave en el diccionario.

```
>>> conjunto = {"a", "b", "c", "d"}
>>> c = frozenset(conjunto)
>>> c
frozenset({'c', 'b', 'a', 'd'})
```

Operaciones Sobre Congelado

El congelado se maneja igual como el objeto tipo conjunto, excepto que no es dinámico.
El congelado no cambia su contenido durante su vida.

No funciona ninguno de los métodos tipo "update" (actualizar), y
no funcionan los métodos dinámicos: "add", "pop", "remove", "discard" y "clear".

7.3.7. Conjunto y Ciclos

Para procesar los ítems de conjunto uno por uno podemos usar las declaraciones:
if-elif-else, while-else y for-else de diferentes maneras.

```
# Si elemento existe
if <element> in <conjunto>:
    ....

# Para cada ítem de conjunto
for <ítem> in <conjunto>:
    ....
```

```
# Mientras el conjunto tiene ítem
while <conjunto>:
    ....
    <ítem> = <conjunto>.pop()
# Por cada ítem de conjunto
for <i> in range(len(<conjunto>)):
    ....
```

7.4. Diccionario ("dictionary")

7.4.1. Definición de Tipo Diccionario

Recuerde los tipos de objetos explicados en la sección 4.4 en la página 49.

Diccionario es un tipo de colección **no-ordenada**, **dinámica**, y **mutable**.
Igual como un diccionario de un idioma, que contiene pares <palabra>: <significado>,
el diccionario es una colección cuyos ítems son pares:

```
<llave>: <valor>
```

El objeto "llave" tiene que ser de tipo **mapeable ("hashable")**.
El objeto "valor" puede ser de cualquier tipo conocido en Python. Frecuentemente son colecciones.
El tipo de objeto "llave" y el tipo de objeto "valor" son independientes.

Los valores son los ítems que guardamos, las llaves son las referencias a estos valores.
Diccionario **mapea** la llave al domicilio del valor en la memoria.

Parecido a como accedemos a los ítems de una lista a través de sus índices,
en el diccionario accedemos a los valores a través de sus llaves.

La característica más importante de diccionario es la rapidez de acceder a los valores que guarda.

La escritura, lectura, modificación o eliminación de un valor del diccionario toma tiempo constante.
Esto quiere decir que el tiempo es el mismo si hay tres valores o tres mil valores, ya que
el diccionario no pierde tiempo en buscar valor, calcula su domicilio a base de la llave proporcionada.

Recorrer todos los valores, o copiar diccionario, toma tiempo proporcional a la cantidad de valores.
O sea, es un tiempo aceptable. Si diccionario aumenta a doble, tiempo de recorrerlo aumenta a doble.

Hay que usar los diccionarios siempre que necesitamos acceder rápido a un conjunto de valores.

7.4.2. Creación de Un Diccionario

A veces generamos el diccionario vacío y en el programa lo vamos llenando de datos, proporcionados desde teclado, un archivo, una red, Internet, base de datos, algún otro lado, o calculamos los valores y los guardamos en el diccionario.

Crear Diccionario Vacío

Existen dos maneras de crear un diccionario vacío:

```
diccionario = dict()
diccionario = {}
```

Crear Diccionario con Ítems

Para crear un diccionario no vacío tenemos una variedad de opciones.
Todavía más opciones vamos a ver al estudiar las "comprensiones".

Podemos usar la forma literal del tipo diccionario: secuencia de pares entre llaves.
La forma literal de un diccionario es:

```
{<llave_1>: <valor_1>, <llave_2>: <valor_2>, ..., <llave_N>: <valor_N>}
```

El ejemplo de crear un diccionario usando su forma literal:

```
>>> ficha_1 = {"nombre": "Juan Perezoso", "edad": 32, "ahorro": 25349.50} >>> ficha_1
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
```

Ahora la "ficha_1" es un diccionario que tiene llaves: "nombre", "edad" y "ahorro".
Estas llaves se usan para acceder a los valores guardados, para leer, modificar, o borrarlos.
Esto lo veremos en la subsección "Procesamiento de Diccionario".

La otra manera de crear un diccionario es la función de conversión a diccionario "dict" ("dictionary").

Usando el convertidor "dict" se puede convertir a diccionario una secuencia de asignaciones:

```
>>> ficha_1 = dict(nombre="Juan Perezoso", edad=32, ahorro=25349.50)
>>> ficha_1
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
```

Usando el convertidor "dict" se puede convertir a diccionario otra colección,
que tenga como sus ítems pares de objetos. Por ejemplo, lista de tuplas con dos ítems:

```
>>> ficha_1 = dict([("nombre", "Juan Perezoso"), ("edad", 32),
... ("ahorro", 25349.50)])
>>> ficha_1
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
```

Para completar la parte de la creación de diccionarios,
se muestra un diccionario que tenga llaves de diferentes tipos de objetos:

```
>>> F = frozenset({"a", "b", "c"})
>>> mixto = {(3, 6): [1, 2, 3], F: 1, 999: {'a': 1, 'b': 2}}
>>> mixto
{(3, 6): [1, 2, 3], 999: {'a': 1, 'b': 2}, frozenset({'c', 'b', 'a'}): 1}
```

Pueden ver que Python reordena los ítems del diccionario como le guste.

Crear Diccionario con Llaves

A veces sucede que tenemos un diccionario y deseamos crear otro nuevo
que tenga las mismas llaves del primer diccionario, pero que esté sin valores
los cuales vamos a agregar después.

En este caso podemos usar la función "fromkeys" (desde_llaves), para formar tal diccionario.
Por ejemplo

```
>>> ficha_1
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
>>> ficha_2 = {}
>>> ficha_2.fromkeys(ficha_1)
{'nombre': None, 'ahorro': None, 'edad': None}
```

El nuevo diccionario "ficha_2" tiene las mismas llaves del "ficha_1", pero con todos valores None.
En la parte "Procesamiento del Diccionario" aprenderemos llenar los valores nuevos.

En lugar de un diccionario existente, podemos usar cualquier otra colección con ítems mapeables,
para que sirvan como llaves del nuevo diccionario.
También es posible especificar los valores iniciales distintos a None:

```
>>> dicc_2 = dict()
>>> dicc_2.fromkeys(["a", "b", "c"], 0)
{'c': 0, 'a': 0, 'b': 0}
```

Estamos creando llaves del diccionario nuevo de una lista y con valores iniciales asignados a cero.

7.4.3. Diccionario es Dinámico

Agregando Ítems

Se puede agregar ítem usando el operador [] y
se pueden agregar varios ítems usando el método "update" (actualizar).

Siendo un objeto dinámico podemos agregar un ítem usando operador []:

```
<diccionario>[<llave>] = <valor>
```

Por ejemplo, empezamos con diccionario vacío y agregamos elementos:

```
>>> d = {}
>>> d["nombre"] = "Juan Perezoso"
>>> d["edad"] = 32
>>> d["ahorro"] = 25349.50
```

```
>>> d
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
```

También podemos agregar otro diccionario, para juntarlos en uno sólo.
Para esto se usa el método "update" (actualizar):

```
>>> d_2 = dict()
>>> d_2.update(d)
>>> d_2
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
```

Para borrar un ítem de diccionario, o borrar el diccionario completo,
se puede usar la declaración "del" ("delete", borrar):

```
>>> d_2
{'nombre': 'Juan Perezoso', 'ahorro': 30000.0, 'edad': 32}
>>> del d_2["edad"]
>>> d_2
{'nombre': 'Juan Perezoso', 'ahorro': 30000.0}
```

Para eliminar todos los ítems del diccionario se usa el método "clear" (despejar).

```
>>> d_2 {'nombre': 'Juan Perezoso', 'ahorro': 30000.0}
>>> d_2.clear()
>>> d_2
{}
```

Con esto diccionario todavía existe, pero ahora está vacío.
Si deseamos eliminar el mismo objeto diccionario, usamos "del":

```
>>> d_2
{}
>>> del d_2
>>> d_2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'd_2' is not defined
```

Ahora el diccionario d_2 ya no existe.

También existen las funciones "pop" (extraer) y "popitem" las cuales eliminan el ítem de diccionario, después de devolver su valor. Esto se explica en la subsección "Procesamiento de diccionario".

Diccionario es Mutable

Siendo diccionario mutable se le pueden reordenar y modificar los ítems.
Dado que diccionario es una colección no ordenara, el cambio de orden de los ítems no aplica.
Para modificar un ítem, simplemente se sobrescribe el valor usando su llave:

```
>>> d_2
{'nombre': 'Juan Perezoso', 'ahorro': 25349.5, 'edad': 32}
>>> d_2["ahorro"] = 30000.00
>>> d_2
{'nombre': 'Juan Perezoso', 'ahorro': 30000.0, 'edad': 32}
```

7.4.4. Funciones Aplicables

Al objeto tipo diccionario se le pueden aplicar las funciones: “len”, “sum”, “min”, “max”.
La función “len” reporta la cantidad de ítems en el diccionario.
Otras funciones mencionadas se refieren a las llaves del diccionario.

```
>>> dicc = {1: "a", 2: "b", 3: "c", 4: "d"}
>>> len(dicc)
4
>>> sum(dicc)
10
>>> min(dicc)
1
>>> max(dicc)
4
```

7.4.5. Procesamiento de Diccionario

Para procesar diccionario tenemos una variedad de métodos para obtener valor especifico,
obtener la lista de llaves, obtener la lista de valores, y la lista de ítems, o sea pares “llave: valor”.
Estos métodos funcionan bien en combinación con: if-elif-else, while-else, y for-else.

Si Conocemos las Llaves

Una manera de obtener el valor, usando su llave, es el operador []:

```
>>> dicc = {1: "a", 2: "b", 3: "c", 4: "d"}
>>> x = dicc[2]
>>> x
'b'
```

En caso de que la llave en los corchetes no esté en el diccionario se levanta “KeyError”.

También tenemos las funciones “get” (obtener) y “pop” (extraer).
Lo común es que estas dos funciones devuelven el valor, sin levantar excepción si la llave no existe.
A estas funciones se les puede proporcionar un valor por defecto,
el cual se devuelve en lugar del valor, cuando la llave no está en el diccionario.
La diferencia es que “pop” elimina ítem del diccionario y el “get” no lo elimina.

```
>>> dicc
{1: "a", 2: "b", 3: "c", 4: "d"}
>>> dicc.get(3)
'c'
>>> dicc.get(5, "x")
'x'
>>> dicc.get(5)
>>> dicc
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> dicc.pop(3)
```

```
'c'
>>> dicc.pop(5, "NE")
'NE'
>>> dicc
{1: 'a', 2: 'b', 4: 'd'}
```

Tenemos a nuestra disposición los operadores de inclusión "in" y "not in"
para verificar si cierta llave está dentro del diccionario:

```
>>> dicc
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> 1 in dicc
True
>>> 0 not in dicc
True
```

Si No Conocemos las Llaves

Si necesitamos procesar un diccionario y no conocemos su contenido,
tenemos varios métodos a nuestra disposición.

Podemos usar el método "popitem" el cual
regresa un ítem (llave, valor) elegido del diccionario al azar y lo elimina del diccionario:

```
>>> dicc
{1: "a", 2: "b", 3: "c", 4: "d"}
>>> dicc.popitem()
(1, 'a')
>>> dicc.popitem()
(3, 'c')
>>> dicc
{2: 'b', 4: 'd'}
```

Si continuamos usando el "popitem", en un momento el diccionario queda vacío.
Si usamos el "popitem" cuando diccionario es vacío se levanta "KeyError".
El "popitem" es práctico en combinación con un ciclo que controla la cantidad de repeticiones.

Existen tres vistas "views" para obtener: llaves, valores o ítems del diccionario:

```
dicc.keys()       # la vista de llaves
dicc.values()     # la vista de valores
dicc.items()      # la vista de ítems: (llave, valor)
```

Estas vistas son copias superficiales "shallow copies" de los datos en el diccionario.
Este término lo aclararemos en la sección 7.5 en la página siguiente.
Estas visas se pueden convertir a listas usando la función de conversión "list", o "sorted":

```
>>> dicc
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
>>> list(dicc.keys())
[1, 2, 3, 4]
```

```
>>> sorted(dicc.values())
['a', 'b', 'c', 'd']
>>> sorted(dicc.items(), key=None, reverse=True)
[(4, 'd'), (3, 'c'), (2, 'b'), (1, 'a')]
```

7.4.6. Diccionario y los Ciclos

7.5. Copias

7.5.1. Copia Superficial "Shallow" y Copia Profunda "Deep"

Copiando una lista que contiene otra colección: lista, tupla, diccionario, o conjunto.

7.6. Comprensiones y Servicios

7.6.1. Comprensión de Listas

pairs = [(v, k) for (k, v) in d.iteritems()].

7.6.2. Comprensión de Diccionarios

7.6.3. Servicios

zip: pares = zip(d.values(), d.keys())

7.7. Librería Estándar

7.7.1. El Modulo "os", el Sistema Operativo

7.7.2. El Modulo "collections", Los Contenedores

7.8. Recomendaciones y Ejemplos

TABLAS NO ESTÁN COMPLETADAS !!!!!!!!

Operación	Cadena	Tupla	Lista	Congelado	Conjunto	Diccionario
Conversión	str()	tuple()	list()	frozenset()	set()	dict()
Mutable			Si		Si	Si
bool()	Si	Si	Si		Si	Si
==, !=,	Si	Si	Si		Si	Si
is, is not	Si	Si	Si		Si	Si
in, not in	Si	Si	Si		Si	Si
<, <=, >=, >	Si	Si	Si		Si	
* N	Si	Si	Si			
+	Si	Si	Si			
len()	Si	Si	Si		Si	Si
min()	Si	Si	Si		Si	Si
max()	Si	Si	Si		Si	Si
sum()		Si	Si		Si	Si
reversed()		Si	Si			
sorted()		Si	Si		Si	Si
.copy()			Si		Si	Si
var = nombre[key]	Si	Si	Si			Si
var = nombre[i:j:k]	Si	Si	Si			
nombre[key] = expresión			Si			Si
nombre[i:j:k] = expresión			Si			
.update(<colección>)						
.pop(<llave>)			Si		Si	Si
.remove(<ítem>)			Si		Si	
.discard(<ítem>) ???						
.clear() ???						
.count(<ítem>) ???						

Operaciones especificas

Cadena	
c.split(<s>)	Convierte la cadena en una lista de partes donde el <s> es el separador.
c.join(colección)	Une los ítems de la colección a una sola cadena, si los ítems son cadenas

Tupla	

Lista	
l.append(<ítem>)	Agrega el ítem al final de la lista.
l.insert(<índice>, <ítem>)	Inserta el ítem en lugar indicado por el índice..
l.extend(<lista>)	Concatena la "lista" a la lista "l" original.
l.sort(reverse=False)	Forma orden no-descendente (no_ascendente si True).
l.reverse()	Crea un iterable que devuelve los ítems en orden invertido.

Conjunto	
c.discard(<ítem>)	Elimina ítem si existe. No levanta la excepción "KeyError"
c.isdisjoint(<conjunto>)	Regresa False si la intersección de "c" y "<conjunto>" es vacía

7.9. Tareas

8. Unidades de Diseño y Archivos

Ya que aprendimos los conceptos básicos de la programación estructurada,
vamos a ver su último elemento: organización de código en funciones.

Después de esto vamos a presentar la clase, el concepto básico de otro paradigma:
programación orientada a objetos.

Ya sabiendo de unidades básicas de organización de código: funciones y clases,
veremos una unidad superior: modulo.
También aprenderemos como organizar código en varios módulos.

Para finalizar el capitulo, aprenderemos como usar las excepciones,
las cuales son herramienta más importante de desarrollo de código robusto,
resistente a errores no procesados y las salidas inesperadas del programa.

8.1. Funciones

Desde el inicio del curso hemos usado funciones. Ya sabemos usar funciones como:
"print", "format", "input", "int", "float", "str", "reversed", "sorted" y más.

En esta sección aprenderemos a diseñar y usar nuestras propias funciones.

Lo más importante de una función es que se define (escribe) solamente una vez y
se puede llamar para ejecución, o sea usar, muchas veces desde distintas partes de código.

Las funciones se usan porque:

- hacen posible el reuso de código; se reusan las funciones en otros programas
- descomponen el diseño de código a unidades cortas y manejables.
- hacen el código más simple para entender y modificar.
- facilitan trabajo en equipo: cada miembro del equipo trabaja en sus funciones asignadas.

Existen dos tipos de funciones según el objeto que regresan.
Cada función en Python, o regresa None, o regresa algún objeto.
Observemos dos funciones bien conocidas: "print" e "input".

El "print" es una función que imprime datos en la salida estándar: la pantalla.
Esta función ejecuta una acción útil, pero no produce ningún objeto como resultado.
Cuando no hay un objeto que regresar, automáticamente se regresa None (Ninguno).

En Python se dice "la función no regresa nada", queriendo decir que la función regresa None.

La función "input" ejecuta una acción útil: recibe datos de teclado, pero, por otro lado,
también produce y devuelve un objeto: la cadena que contiene los caracteres tecleados.

En algunos lenguajes, las funciones que no devuelven un objeto se les llama "procedimiento" y
solamente las que regresan algún objeto distinto de None, se les llama "función".
En Python, ambas son funciones: las que devuelven un objeto y las que devuelven None.

8.1.1. Definición y Ejecución de Una Función

Para escribir nuestra propia función usamos la declaración "def" ("definition"):

```
def <nombre_de_la_función>(<lista_de_los_parámetros>):   # encabezado
    <bloque_de_declaraciones>                            # cuerpo
```

La primera linea de la definición de la función se le llama el "encabezado" y
el bloque de declaraciones que sigue, se le llama el "cuerpo" de la función.

Lo más importante de entender sobre una función es lo siguiente:
El encabezado de la función se evalúa solamente una vez, al leer la definición de la función.
El cuerpo de la función se evalúa muchas veces, cada vez que se llame (ejecute) la función.

Si creen que lo entendieron, se equivocan. Así que, otra vez:

El encabezado de la función se evalúa solamente una vez, al leer la definición de la función.
El cuerpo de la función se evalúa muchas veces, cada vez que se llame (ejecute) la función.

Lo repetiremos muchas veces más a lo largo del texto.
Lo entenderán después de hacer unos cuantos errores al programar.

El encabezado consiste de cuatro partes:

1. "def" el cual es la palabra reservada y señala la definición de una nueva función.
2. "nombre_de_la_función" es una variable que referencia la función definida
   cuando esta se guarde como un objeto tipo "función" en la memoria.
   Se usa para identificar esta función particular en el resto del programa.
3. (lista_de_los_parámetros) son las variables que referenciarán argumentos
   que van a ser procesados en el cuerpo de la función.
   Una lista de parámetros vacía se identifica por tener los paréntesis vacíos: ().
4. el símbolo : significa que sigue el bloque de las declaraciones; el cuerpo de la función.

El cuerpo de la función consiste de un bloque de una o más declaraciones
que se ejecutan cada vez al llamar (invocar, usar) la función, después de ser definida.
Estas declaraciones constituyen el cuerpo de la función y
definen lo que la función hace y lo que la función regresa.

El bloque de cuerpo puede ser vació, lo que se indica usando declaraciones "pass", o "...":

```
def función_1():                  def función_1():
    pass                              ...
```

Las dos opciones son idénticas: definiciones de la "función_1" con su cuerpo vacío.
Esta situación puede aparecer durante el desarrollo de código, solamente como recordatorio,
que vamos a necesitar una "función_1", cuyo cuerpo vamos a definir después.

Para elegir nombre de una función, aplican la mismas reglas que usamos para variables:

1. el primer carácter tiene que ser una letra, o símbolo de subrayado _,
2. resto de los caracteres, aparte de letras y subrayado, también pueden ser dígitos.

Además de los requerimientos del Python, también vamos a adoptar el estándar PEP8,
el cual recomienda que el nombre de la función siempre sea un verbo
que dice en una palabra, o pocas palabras, que es lo que la función hace.

Definición de Una Función

Para despejar el misterio, de una vez escribiremos nuestra primera función.
Es una función sencilla que suma dos objetos, referenciados por "a" y "b":

```
# Definición de la función "sumar":
def sumar(a, b):
    return a + b

# El código desde el cual se invoca la función "sumar" para que se ejecute:
x = -3
y = 4
z = sumar(x, y)
print(z)
print(sumar("¡Que tal! ", "Escribimos y usamos una función."))
```

En la definición de esta función, el cuerpo consiste de una sola declaración: "return a + b".
El "return" es una declaración dentro de la cual sigue el objeto
el cual la función va a regresar ("return") a la expresión que la invocó (llamó, ejecutó).

En este ejemplo la función "sumar" la definimos una vez y la llamamos dos veces para ejecutarla.

Escriba y ejecute este ejemplo de manera interactiva, o como un script.

En continuación estudiaremos ¿Que hace el intérprete al encontrar la definición de una función?

El interprete primero procesa el encabezado de una manera y después el cuerpo de manera distinta.

La palabra "def" indica que empieza la declaración de una función.
Se procede como si fuera una asignación, donde el nombre de la función es la variable y
el objeto que se va a referenciar es el resultado de la evaluación de la lista de parámetros.
Después se añade el cuerpo de la función a este objeto de tipo "función", como su segunda parte.

La lista de los parámetros se evalúa como una secuencia de asignaciones.
En este caso la lista es "a, b". Para el intérprete, esto es idéntico a: "a=None, b=None".
La lista de los parámetros es para Python una lista de asignaciones.

Pudimos haber definido la lista asignando algunos valores a los parámetros.
Por ejemplo "a=0, b=2", o "a, b=1", son validas listas de parámetros.
En falta de valor asignado, el intérprete usa None.

Si hay un argumento asignado a un parámetro, se guarda el argumento a la memoria y
el parámetro referencia este objeto. Si no hay argumento asignado, el parámetro referencia None.
La lista de los parámetros forma la primera de dos partes del objeto "función sumar".
La segunda parte es el cuerpo de la función "sumar".

Después de evaluar la lista de los parámetros y establecerla como parte de objeto "función sumar",
se verifica la sintaxis del cuerpo de la función y si no tiene errores de sintaxis,
el cuerpo se agrega como la segunda parte del mismo objeto "función sumar".

Ahora estudiaremos: ¿Qué NO hace el intérprete al encontrar la definición de una función?

Lo siguiente es muy importante que se entienda bien:
Lo que NO hace el intérprete al procesar la definición de una función es:

- No ejecuta el cuerpo de la función y por lo tanto no produce ninguna acción ni resultado.

- No verifica si la función puede provocar un error semántico al ser ejecutada.

Es muy importante entender lo que no hace el intérprete al ejecutar la declaración "def",
así que, vamos a ver dos ejemplos.

Si modifican la función "sumar" en el ejemplo anterior, agregando un operador de suma extra:

```
def sumar(a, b):
    return a + b +
```

al leer el cuerpo de la función, el intérprete descubrirá este error de sintaxis y lo reportará,
ya que el interprete verifica que no haya errores de sintaxis en el cuerpo la función,
antes de aceptar el cuerpo y guardarlo en la memoria como parte de objeto "función sumar".

Por otro lado, si modificamos la función agregando un error semántico,
por ejemplo una división con cero:

```
def sumar(a, b):
    return a + b / 0
```

al leer esta declaración "def", el intérprete no va a descubrir el problema semántico.
Va a guardar el cuerpo en la memoria como un cuerpo sintácticamente correcto.
Solamente al intentar de ejecutar esta función después, se descubrirá el error.

Por esta razón, y otras más, es importante "desarrollo de software basado en pruebas".
De esto vamos a hablar más adelante en la parte avanzada de este texto.

Por ahora es importante saber:
si tenemos una función en el código y no la llamamos para ser ejecutada,
esta función no está probada y puede causar errores después, cuando empezamos a llamarla.

Ejecución de Una Función

Regresemos al ejemplo de la primera función que hemos diseñado:

```
# Definición de la función "sumar"
def sumar(a, b):
    return a + b

# El código desde el cual se invoca la función "sumar" para que se ejecute.
x = -3
y = 4
z = sumar(x, y)
print(z)
print(sumar("¡Que tal! ", "Escribimos y usamos una función."))
```

La función "sumar" se define una vez, y puede ser ejecutada las veces que necesitemos.
En este ejemplo es ejecutada dos veces: primero suma dos enteros y después dos cadenas.

Vamos a analizar la ejecución de la primera llamada de esta función:

```
z = sumar(x, y)
```

Al llegar a esta asignación, el intérprete empieza evaluando la expresión "sumar(x, y).
Identifica la función "sumar", e identifica el operador de ejecución ().
El siguiente paso es sustituir los argumentos "x" y "y" con constantes.

Dado que las variables "x" y "y" referencian enteros, los cuales son objetos inmutables,
se sustituyen las variables por valor de los objetos referenciados: "sumar(-3, 4)".

Si fueran objetos mutables, se sustituirían por su domicilio: "sumar(id(x), id(y))",
donde los "id()" serían constantes en sistema hexadecimal, de forma 0xHHHHHHHH;
los domicilios de los objetos "x" y "y".

Una vez sustituidos los valores: "sumar(-3, 4)",
el intérprete mueve la información sobre la función, incluida la lista de argumentos "-3, 4" y
el domicilio donde empieza el cuerpo de objeto "función sumar" al "stack".

El "stack" es una parte de memoria donde se guardan apilados los objetos activos de tipo "función".
Al ejecutarse una función, ya no es activa y sus datos correspondientes se remueven del "stack".

Después de mover los datos pertinentes del objeto "función sumar" al "stack",
el cuerpo del objeto "función sumar": "return a + b" se ejecuta, calculando que -3 + 4 es 1 y
la declaración "return" hace que la "función sumar" se remueve del "stack" y
el objeto "entero 1" se regrese a sustituir la expresión "sumar(-3, 4)".

La asignación

```
z = sumar(x, y)
```

al sustituir variables, se convirtió en

```
z = sumar(-3, 4)
```

y al ejecutar la función se convirtió en

```
z = 1
```

Al fin se ejecuta la declaración de asignación. Ya sabemos como:
se guarda el objeto "entero 1" a la memoria y su domicilio queda referenciado por la variable z.

Ahora, al entender el proceso de ejecución de una función es importante entender dos detalles.

El primer detalle.

Los parámetros de la función "sumar" son variables "a" y "b" que no tienen tipo de dato
ya que en Python los objetos tienen tipo de dato y no las variables.

Esto nos permite usar la función "sumar" para todos tipos de datos a los cuales es aplicable
el operador +, ya que este operador es el único que aparece en su cuerpo: "return a + b".

El segundo detalle.

Para ejecutar una función no es suficiente su nombre, en este caso "sumar".
Para ejecutar una función, hay que aplicar el operador de ejecución: () después del nombre.
El paréntesis tiene que contener los argumentos necesarios para la función.

Observen el siguiente código:

```
x = -3
y = 4
z = sumar
print(z(x, y))
```

En la asignación "z = sumar", la función "sumar" no se va a ejecutar,
ya que su nombre no esta seguido por el operador de ejecución: ().

La asignación "z = sumar" simplemente va a lograr que la variable "z" referencie
el mismo domicilio, el mismo objeto "función sumar", referenciado por la variable "sumar".
Ahora y "z" y "sumar" referencian la misma función y la función puede ser llamada
para la ejecución a través de cualquiera de las dos variables.

En la siguiente declaración "print(z(x, y))", la expresión "z(x, y)" causa la ejecución de la función.
El resultado "entero 1" no se guarda en la memoria, ya que no está asignado a ninguna variable,
solamente se imprime en la pantalla.

8.1.2. Jerarquía y Ramificación de las Funciones

Al trabajar bajo paradigma "programación estructurada", es costumbre que
casi todo el código esté dentro de funciones que se comunican llamando unas a otras y
regresando los resultados unas a otras.

Al diseñar programas a base de funciones, entre ellas emerge una estructura jerárquica,
algo como jerarquía militar. Como si existieran funciones superiores y funciones subordinadas.

En el siguiente ejemplo, los cuatro puntos representan declaraciones que pueden estar presentes,
pero de momento estas declaraciones son irrelevantes para entender la jerarquía.

Si la función "f_1" desde su cuerpo llama a ejecutar la función "f_2":

```
def f_1():
    ....
    f_2()
    ....

def f_2():
    ....
```

entonces la función "f_1" es de alguna manera superior a la función "f_2" ya que,
la función "f_2", podríamos decir, "trabaja para" la función "f_1".

En varios lenguajes de programación es obligatorio que la función superior a todas,
con la que empieza la ejecución de programa sea llamada "main" (principal).
En Python esto no es necesario; la función principal puede tener cualquier nombre.
Por lo pronto usaremos nombre "main" para la función principal,
con objetivo de facilitar la transición hacia Python a los programadores de otros lenguajes.
Entonces la función "main" será la raíz del árbol de llamadas de funciones.

Para empezar con los ejemplos concretos de programación basada en funciones,
vamos a usar ejemplo de un programa muy sencillo, tan sencillo que lo vimos al principio:

```
nombre = input("Por favor, ingrese su nombre: ")
print("Hola, ", nombre, ".", sep="")
```

Ahora vamos a escribir el mismo programa, usando sin necesidad varias funciones,
nada más para demostrar las estructuras posibles de un programa organizado en funciones.
La idea es que código sea sencillo para poder entender la relación entre las funciones.

```
def main():
    nombre = pide_nombre()
    saluda(nombre)

def pide_nombre():
    return input("Por favor, ingrese su nombre: ")

def saluda(persona):
    print("Hola, ", persona, ".", sep="")

main()
```

Este programa tiene tres funciones.
El intérprete primero lee la función "main", pero recuerden que no la ejecuta,
solamente la guarda en la memoria. Lo mismo sucede con otras dos funciones.

Al llegar a la declaración

```
main()
```

el intérprete identifica la función "main" en la memoria, e identifica el operador de ejecución: () y empieza a ejecutar el cuerpo de la función "main".
Desde el cuerpo de la función "main" se llaman a ejecutar, una por una, otras dos funciones.

Lo que nos interesa ahora son las relaciones entre estas funciones.

El "main" es la función principal, ya que es la primera que se empieza a ejecutar.
Otras dos funciones "hacen trabajo" para "main" y por lo tanto son subordinadas al "main",
mientras son independientes entre ellas; no se llaman a ejecutar una a otra.

Podríamos haber organizado estas tres funciones de manera distinta.

```
def main():
    saluda()

def saluda():
    persona = pide_nombre()
    print("Hola, ", persona, ".", sep="")

def pide_nombre():
    return input("Por favor, ingrese su nombre: ")

main()
```

En esta organización de código, "main" es superior a "saluda" y
"saluda" es superior a "pide_nombre". La jerarquía es lineal.

Si las funciones se desarrollan en el orden desde la principal, hacia la más subordinada,
esto se le llama "diseño de arriba hacia abajo" ("top-down design").

Si las funciones se desarrollan en el orden desde las más subordinadas hacia la principal,
esto se le llama "diseño de abajo hacia arriba" ("bottom-up design").

En el diseño de abajo hacia arriba, el código podría estar organizado así:

```
def pide_nombre():
    return input("Por favor, ingrese su nombre: ")

def saluda():
    persona = pide_nombre()
    print("Hola, ", persona, ".", sep="")

def main():
    saluda()

main()
```

Pero recuerde, no importa en cual orden se escriben las funciones en el código,
el "diseño de arriba hacia abajo" y "diseño de abajo hacia arriba"
hablan de orden en el que se desarrollaron.

Estas estructuras jerárquicas ramificadas de las funciones son muy comunes, pero no únicas.

A veces tenemos casos que una función subordinada tiene varias superiores que la llaman.

También hay casos donde una función se llama a sí misma indirectamente o directamente,
lo que se llama "recursión" y es muy común en ciertas estructuras de datos y sus algoritmos.

```
# Múltiples superiores
def f_1():
    ....
    f_3()

def f_2():
    ....
    f_3()

def f_3():
    ....
```

```
# Recursión indirecta
def f_1():
    ....
    f_2()

def f_2():
    ....
    f_1()
```

```
# Recursión directa
def f_1():
    ....
    f_1()
    ....
```

Los cuatro puntos representan otras declaraciones que pueden estar presentes,
pero de momento estas declaraciones son irrelevantes para entender la jerarquía.

Todas estas estructuras las vamos a encontrar en futuros ejemplos.

8.1.3. Los Parámetros y los Argumentos de Una Función

Ya vimos que una función intercambia información con el resto de código en ambas direcciones:
puede recibir datos para procesarlos y puede regresar resultados a la declaración que la llamó.

Ahora nos vamos a enfocar en cuestión ¿cómo una función recibe datos al ser llamada?.

Hay dos aspectos en este proceso:
la recepción de objetos codificada en al definición de la función, usando parámetros, y
el envío de objetos codificado en la llamada de la función, usando argumentos.

Existen funciones que no reciben datos.
En los ejemplos anteriores, la función "pide_nombre"
tiene su lista de parámetros vacía: "def pide_nombre():".
Esta función no espera ningún objeto.
Tampoco se le manda algún objeto cuando se le invoca a ser ejecutada.

Vamos a escribir una función y llamarla "aumentar".

```
def aumentar(x, aumento=1):
    return x + aumento

def main():
    a = 1
    print("a =", a)
    a = aumentar(a)
    print("a =", a)
    a = aumentar(a, 2)
    print("a =", a)
    a = aumentar(a, aumento=3)
    print("a =", a)
    a = aumentar(x=a, aumento=3)
    print("a =", a)

main()
```

Primero hay que aprender unas expresiones importantes
respecto a los parámetros y los argumentos de la función "aumentar".

Los Parámetros No-Predeterminados y los Parámetros Predeterminados

En la definición de la función "aumentar" tenemos la lista de parámetros: "x, aumento=1",
donde el "x" y el "aumento" son los parámetros.
Ya vimos anteriormente que para el intérprete esto quiere decir: "x=None, aumento=1".

Dado que el "x" de momento no tiene nada asignado, se le llama simplemente "parámetro",
o más elaborado: "parámetro no-predeterminado", o "parámetro sin asignación".

El "aumento" tiene el objeto "entero 1" asignado en la lista de parámetros,
por lo tanto se le llama "parámetro predeterminado", o más elaborado:
"parámetro con el valor predeterminado", o "parámetro con el valor asignado".

En una lista de parámetros donde se mezclan parámetros no-predeterminados y predeterminados,
primero se escriben todos los parámetros no-predeterminados y solamente después de ellos
se escriben todos los parámetros predeterminados.

Los Argumentos Posicionales y los Argumentos con Clave

En el ejemplo anterior, la penúltima, la tercera llamada a la función "aumentar" es

```
a = aumentar(a, aumento=3)
```

Al llamar una función para ser ejecutada, dentro del operador de ejecución, los paréntesis (), tenemos una lista de argumentos. En este caso los argumentos son "a" y "3".

El argumento "a" no está asignado a ningún nombre.
Tal argumento se le llama "argumento posicional" ya que su posición en la lista determina a cual parámetro en la definición de la función va a ser asignado.

Como "a" es el primero en la lista de los argumentos,
será asignado al primero en la lista de los parámetros.
O sea, el argumento "a" será asignado al parámetro "x".

El argumento "3" tiene una clave "aumento" que debe corresponder a nombre de un parámetro.
Tal argumento se le llama "argumento con clave" ("keyword argument") ya que la clave determina a cual parámetro en la definición de la función "aumentar" va a ser asignado.

Como la clave "aumento" corresponde al parámetro "aumento",
el argumento "3" será asignado al parámetro "aumento".

En una lista de argumentos donde se mezclan argumentos posicionales y argumentos con clave, primero se escriben todos los argumentos posicionales y solamente después de ellos se escriben todos los argumentos con clave.

Correspondencia de los Argumentos y los Parámetros

El siguiente ejemplo muestra que es posible escribir y funciones y otras declaraciones que no están dentro de las funciones, dentro de un modulo.

Aunque es posible, esta mezcla no se recomienda,
ya que no es fácil de entender tal código en archivos largos.
Lo usamos solamente para los ejemplos cortos.

En programación estructurada un modulo consiste de
las definiciones de las funciones,
una llamada a la función principal "main()" y
opcionalmente algunas constantes, las cuales son compartidas entre varias funciones.

El enfoque del siguiente ejemplo está en la relación entre los argumentos y los parámetros.
Observe el código y saque sus conclusiones:

```
def f(w, x, y=2, z=3):
    print(format(w, '4d'), format(x, '4d'), format(y, '4d'),
          format(z, '4d'))

def main():
    print("   w    x    y    z ")

    f(10, 11)
    f(10, 11, 12)
    f(10, 11, 12, 13)
    f(10, 11, 12, z=13)
    f(10, 11, y=12, z=13)
    f(10, 11, z=13, y=12)
```

```
        f(10, 11, z=13)
        f(10, x=11, y=12, z=13)
        f(x=11, w=10)

    main()
```

Ejecute el ejemplo como un script.

Un poco de ayuda con las conclusiones:

Es obligatorio mandar argumentos a los parámetros no-predeterminados.

Es posible, pero no es necesario,
mandar argumentos a los parámetros predeterminados,
de la manera independiente a cada parámetro predeterminado:
a todos, a unos si y a otros no, o a ninguno.

El orden de los argumentos posicionales es importante, pero
el orden de los argumentos con clave no es importante.

Argumentos posicionales se pueden usar
tanto para los parámetros no-predeterminados como para los predeterminados.
Argumentos con clave se pueden usar
tanto para los parámetros no-predeterminados como para los predeterminados.

Por último una advertencia.
En literatura sobre el lenguaje Python,
a veces escritores no ponen mucha atención en distinguir entre parámetros y argumentos.
Usan estas dos palabras como si fueran sinónimos.
Lo que tratan de decir se tiene que estar concluyendo del contexto.

8.1.4. Los objetos Inmutables Como Argumentos

Los tipos de datos en Python se dividen en inmutables y mutables.
Los inmutables no se modifican "en lugar"; en el domicilio de la memoria donde están.

Si una variable referencia a un objeto inmutable, como por ejemplo

```
x = 1
```

donde la variable "x" termina referenciando el objeto "entero 1" en cierto domicilio,
y se reasigna la variable a otro objeto

```
x = 2
```

entonces el objeto "entero 1" no se modifica en su mismo domicilio para ser "entero 2".
El nuevo objeto "entero 2" se guarda en algún otro domicilio disponible y
la variable "x" se reasigna para referenciar este otro domicilio.

El domicilio de objeto "entero 1", si no hay otras variables que lo referencian,
se recupera por parte del "colector de basura" y se convierte en domicilio disponible.

En caso de objetos mutables, que conoceremos después, el cambio se hace "en lugar",
no hay reasignación de la variable a otro domicilio.

¿Qué pasa al mandar una variable inmutable como un argumento a una función?

Por ejemplo

```
x = 1

def f(param):
    param = 2

f(x)
print(x)
```

Al ejecutar este script, se guardan en la memoria la variable "x" y la función "f", y
sigue la ejecución de la función "f" con argumento "x", o sea: f(x).

Supongamos que el objeto referenciado por "x", el "entero 1", está en el domicilio 0x10105025.
El prefijo "0x" significa que el número es representado en el sistema numérico hexadecimal.

Para empezar a ejecutar la expresión "f(x)" primero se tiene que sustituir "x".
Para esto existen dos opciones.

Se puede sustituir con el valor, con una copia, de objeto referenciado:

```
f(1)
```

o se puede sustituir con el domicilio de objeto referenciado original:

```
f(0x10105025)
```

Cuando el objeto referenciado por "x" es inmutable, la variable se sustituye con la copia del objeto.
Cuando el objeto referenciado por "x" es mutable, la variable se sustituye con el domicilio del objeto.

Dado que tipo entero es tipo inmutable,
una copia del objeto "entero 1" sustituye "x" en la expresión de la llamada y tenemos:

```
f(1)
```

Esta copia de "objeto 1" tiene domicilio distinto del original "objeto 1" referenciado por "x" y
es la copia que se le asigna al parámetro "param" en la definición de la función.

De esta manera la reasignación del "param" a objeto "entero 2" en la función
no afecta la variable "x", la cual sigue referenciando el objeto "entero 1" original.

En resumen. Si pasamos un objeto inmutable a una función, la función recibe copia de su valor.
Todos los cambios hechos a esta copia en la función, no afectan el valor de argumento original.

8.1.5. Alcances de Nombres: Globales, Locales y No-Locales

Alcances ("Scopes") de los nombres
es uno de los conceptos esenciales en los lenguajes de programación.
Puede ser implementado de manera distinta en distintos lenguajes.
Ahora estudiaremos como se manejan los alcances de los nombres en el lenguaje Python.
Por suerte, es muy sencillo de comprender.

Las funciones separan el código en unidades cortas y manejables.
Una de las ventajas de esto es que para escribir una función definimos claramente:
que datos recibe, que acciones ejecuta y cual valor regresa,
y después desarrollamos la función sin tener que pensar en todas las demás funciones.

Para facilitar este proceso, de la independencia del desarrollo de las funciones,
el cual permite que distintos miembros del equipo diseñen distintas funciones,
se utiliza el concepto de "alcance" de los nombres de las: variables, funciones, clases y módulos.

El alcance de un nombre es el nivel de visibilidad de este nombre. Los niveles son:
una función, una clase, un modulo, todo el programa.

Los Nombres Globales

Primero vamos a distinguir entre tres alcances más comunes: incorporado, global y local:
Nombres incorporados tienen alcance de todo el programa.
Nombres globales tienen alcance dentro de su modulo.
Nombres locales tienen alcance dentro de su función.

Observe el siguiente ejemplo:

```
a = 1

def aumentar(x):
    print(x + a)

print(a)
aumentar(3)
```

El nombre "a" es global, está definido a nivel de modulo.
El nombre "def" es incorporado al intérprete: es una palabra reservada, una declaración.
El nombre "aumentar" es global, está definido a nivel de modulo.
El nombre "x" es local, está definido a nivel, o sea dentro de, la definición de una función.
El nombre "print" es incorporado al intérprete; es una función y una declaración de imprimir.

Las variables globales son visibles para la lectura dentro del todo el modulo,
inclusive dentro de las funciones definidas en el mismo modulo.
Las variables locales a una función, son visibles solamente dentro de la definición de esta función.

Por ejemplo: si en lugar de la declaración "print(a)" intentamos "print(x)",
provocaríamos levantamiento de una excepción, ya que "x" no es visible fuera de su función.

Una variable global es visible al nivel del modulo y
dentro de todas las funciones definidas en el mismo modulo,
aunque la variable esté definida después de la definición de la función.

Observe el siguiente ejemplo.

```
def aumentar(x):
    print(x + b)

b = 2
aumentar(4)
```

La variable global "b" puede ser definida después de definir la función "aumentar" que la usa,
pero tiene que ser definida antes de que esta función sea llamada para la ejecución.
El orden

```
def aumentar(x):
    print(x + b)

# INCORRECTO:
aumentar(4)
b = 1
```

provocaría levantamiento de una excepción, ya que al llamar a ejecutar la función "aumentar",
el intérprete todavía no conoce el nombre "b".

Recuerden que al procesar la definición de una función, el intérprete no evalúa su cuerpo,
solamente verifica la sintaxis del cuerpo y guarda la función en la memoria como un objeto.
Por esta razón el nombre "b" mencionado dentro del cuerpo de la función "aumentar",
no causa que el interprete guarde este nombre como referencia a algo, en sus tablas de nombres.

Por otra parte, el encabezado es evaluado en el momento de leer la definición de la función y
el nombre "x" es guardado como x = None.
El parámetro "x" es reasignado a 4 cuando el "aumentar(4)" llama a ejecutar la función.

El Problema de Variables Globales

La visibilidad de los nombres globales dentro de todas las funciones del modulo,
de hecho es un problema.
Cualquier función dentro del modulo tiene la variable global a su alcance.
Si tenemos problemas con valores inesperados de una variable global,
y hay muchas funciones que usan esta variable, es complicado detectar
cual es la función que esté provocando el error de la reasignación inesperada.

Otra razón para evitar el uso de las variables globales en las funciones
es que dificultan el reuso de una función.
Si deseamos reusar una función en otro programa,
también tenemos que reusar cualquier variable global usada dentro de esta función y
posiblemente otras declaraciones relacionadas con esta variable global.

Por estas razones, en la programación estructurada tratamos de no usar variables globales.
Todos los bloques, decisiones y ciclos los programamos dentro de las funciones que se comunican.

A nivel de script tenemos solamente:
las funciones,y
una llamada a la función principal, para que empiece la ejecución del programa y
eventualmente alguna constante compartida entre varias funciones.

Por ejemplo, una constante, tres funciones y una llamada a la función principal:

```
CONST = 123

def imprime_cuadrado():
    print(CONST ** 2)
```

```
def imprime_cubo():
    print(CONST ** 3)

def main():
    imprime_cuadrado()
    imprime_cubo()

main()
```

Si una constante es usada solamente por una función,
se elimina del alcance global, y se define en el alcance local de esta función.
Esto se muestra en los siguientes dos ejemplos:

```
# SI

def imprime_encabezado():
    print("cubo")

def imprime_cubo():
    CONST = 1234
    print(CONST ** 3)

def main():
    imprime_encabezado()
    imprime_cubo()

main()
```

```
# NO

CONST = 1234

def imprime_encabezado()
    print("cubo")

def imprime_cubo():
    print(CONST ** 3)

def main():
    imprime_encabezado()
    imprime_cubo()

main()
```

Una variable a nivel global debe evitarse. Una constante a nivel global es aceptable si:
1) es usada por más de una función y
2) ninguna de las funciones que la usa, no la está modificando.

Para reforzar esta idea de no usar nombres globales en las funciones, excepto como constantes, el Python tiene declaración "global". En seguida explicamos su uso.

La Declaración "global"

Una variable global es visible dentro de las funciones para la lectura, pero
no es fácilmente accesible dentro de las funciones para la reasignación.

En el siguiente ejemplo hay dos variables globales, donde una es usada como constante y
la otra es variable la cual deseamos reasignar dentro de la función.

```
# Nombres globales:
CONST = 1
var = 2

def función(x):
```

```
    var = x
    print('Desde "función": ', CONST, var)

def main():
    función(20)
    print('Desde "main":    ', CONST, var)

main()
```

En este caso la declaración "var = x" no va a reasignar la variable global "var".
Lo que sucederá es que se va a crear una nueva variable local llamada "var"
a la cual se le va a asignar el mismo objeto referenciado por el parámetro "x".

De esta manera la variable global "var" queda escondida.
Queda fuera de la visibilidad para la lectura y reasignación,
dentro de todo el cuerpo de la función "función".

Si de verdad queremos reasignar una variable global,
el lenguaje Python no lo va a prohibir, pero si nos va a obligar a declarar la intención,
para hacernos recordar que lo que estamos haciendo no es recomendable.

Para avisar que vamos a reasignar una variable global dentro de una función,
se usa la declaración "global":

```
global <variable_global>
```

En nuestro ejemplo tenemos que declarar la intención de reasignar variable global "var",
para que el intérprete no vaya a crear una variable "var" local, escondiendo la "var" global.

La declaración se hace antes de usar la variable global "var" ya sea para la lectura, o la reasignación,
en el cuerpo de la función.

Esto lo muestra el siguiente código:

```
# Nombres globales:
CONST = 1
var = 2

def función(x):
    global var
    var = x
    print('Desde "función": ', CONST, var)

def main():
    función(20)
    print('Desde "main":    ', CONST, var)

main()
```

En este último ejemplo, las declaraciones

```
    global var
    var = x
```

están reasignando la variable global "var" al objeto referenciado por la variable "x", y
no se esta creando ninguna variable "var" local.

Ejecute los dos últimos ejemplos como scripts y observe la diferencia en los resultados.

Las Variables Locales

La localización, la limitación de alcance de las variables definidas dentro de una función,
nos permite reusar mismos nombres en distintas funciones sin provocar la "colisión de nombres".
Por ejemplo:

```
A = 5

def aumentar(x):
    print(x + A)

def disminuir(x):
    print(x - A)

aumentar(10)
disminuir(30)
```

El nombre "x" es usado en el diseño da ambas funciones: "aumentar" y "disminuir",
de manera que cada variable es independiente y referencia un objeto distinto,
sin que esto provoque problemas al intérprete, ya que cada una es local en su función.

Variables locales son definidas en la lista de parámetros, o dentro del cuerpo de la función:

```
def es_mayor_que(x):
    y = int(input("Ingrese un número entero: "))
    return x < y
```

En esta función, las variables "x" y "y" son locales a la función:
el "x" es el parámetro de la función y el "y" esta definida en el bloque del cuerpo de la función.

8.1.6. Las Funciones Anidadas

Variables No-Locales

Usualmente, las funciones se definen a nivel del modulo y sus nombres son nombres globales.

A veces, dentro de la jerarquía de las funciones, una función sencilla es subordinada
a solamente una función superior que la llama.
Entonces no tiene sentido hacer la función subordinada visible a nivel de todo el modulo.
Esta función subordinada puede ser definida dentro del cuerpo de su función superior.
Así tendremos la función superior como externa y la función subordinada como interna, o anidada.

```
def f_externa():
    ....
    def f_interna():    # Definición de la función interna
```

```
        ....
    ....
    f_interna()         # Ejecución de la función interna
    ....
```

A lo largo de este texto encontraremos muchos ejemplos prácticos
de una función definida dentro del cuerpo de otra función,
ya que esta es una situación relativamente común en Python y
ciertas técnicas de programación avanzada usan esta estructura anidada de dos funciones.

Tener tres, o más funciones anidadas, ya no tiene uso práctico en Python y no se recomienda.

Ahora presentaremos un ejemplo sencillo y artificial, o sea no-práctico, de dos funciones anidadas,
solamente para entender el concepto de la variable no-local: ni global, ni local.

En los ejemplos que siguen ya no nos interesan nombres: "def", "return", "print",
y otros nombres incorporados, ya que estos están a cargo del intérprete.
Solamente nos interesan los nombres globales, no-locales y locales,
ya que estos nombres los diseñamos nosotros para nuestros programas.

```
a = 20
b = 100

def f_externa(x):
    y = x + a

    def f_interna(z):
        return z + b

    return f_interna(y)

def main():
    print(f_externa(3))

main()
```

Desde el nivel del módulo, solamente existen nombres:
globales: variables "a", "b", y funciones "f_externa" y "main".
Otros nombres no son visibles.

Desde el nivel de la función "f_externa", los nombres visibles son:
globales: variables "a", "b", y funciones "f_externa" y "main",
locales: "x", "y", y función "f_interna".

Desde el nivel de la función "f_interna", los nombres visibles son:
globales: variables "a", "b", y funciones "f_externa" y "main",
no-locales: variables "x", "y", y función "f_interna",
locales: variable "z".

La Declaración "nonlocal"

La declaración "nonlocal" se usa en el mismo sentido en el que se usa "global",
prevenir la creación de una variable local nueva,

al desear de modificar una variable no-local dentro del nivel local.

Para entender el uso de la declaración "nonlocal",
podemos acomodar el ejemplo anterior para evitar el uso de la variable local "z",
modificando la variable no-local "y" dentro de la función "f_interna".

```
a = 20
b = 100

def f_externa(x):
    y = x + a

    def f_interna():
        nonlocal y
        y += b

    f_interna()
    return y

def main():
    print(f_externa(3))

main()
```

La declaración "nonlocal" avisa al intérprete que vamos a modificar la variable "y" no-local,
para que el interprete no vaya a crear una variable local nueva con el mismo nombre "y".

Claro que este ejemplo se puede simplificar mucho más,
solamente sirvió para el propósito de entender la declaración "nonlocal".

8.1.7. Las Funciones Recursivas

Cuando estudiamos sobre la jerarquía y ramificación de las funciones,
en la sub-sección 8.1.2 en la página 136,
mencionamos que las funciones se pueden llamar a si mismas..
Una función que se llame a si misma directamente, o indirectamente,
se le llama una "función recursiva", o simplemente una "recursión".

Funciones recursivas se usan en los algoritmos de ciertas estructuras de datos,
como son: listas ligadas, arboles, etc.
El estudio de las estructuras de datos no es el objetivo de este libro;
lo dejamos para algún otro libro.

Aquí desarrollaremos solamente algunos ejemplos de recursión,
para que se entienda el concepto y la estructura de las funciones recursivas.

Está comprobado que cualquier problema resuelto con una función recursiva,
también puede ser resuelto usando ciclo de repeticiones en lugar de la recursión.

La ventaja de la función recursiva es que hay casos donde expresa el algoritmo
de manera mucho más sencilla y entendible que el uso de ciclo.

Las desventajas de las funciones recursivas:
1) Son más lentas que iteraciones. Es más tardado llamar una función que incrementar un contador.
2) Pueden llenar el "stack", causando levantamiento de una excepción y abandono de la tarea.

Por estas razones las funciones recursivas se limitan a poca cantidad de llamadas a si mismas.

Las funciones recursivas se usan cuando una tarea sobre cierta cantidad de datos,
se puede simplificar a la misma tarea, pero con la cantidad menor de datos,
más procesamiento de los datos apartados.

Para empezar, analicemos un caso sencillo.

Queremos imprimir la secuencia de enteros positivos: n, (n-1), (n-2), ... 2, 1.
Esto ya lo sabemos hacer con un ciclo:

```
for x in range(n, 0, -1):
    print(x)
```

Para hacerlo como recursión, tenemos que pensar en esta tarea como si tuviera dos partes:
Primero hay que imprimir el número n, y después imprimir los demás: (n-1), (n-2), ... 2, 1.
De esta manera, la tarea original "imprimir una secuencia de n enteros positivos",
se redujo a la misma tarea, "imprimir una secuencia de (n-1) enteros positivos",
pero con menos datos.

Para que el conteo no continúe con ,... , 2, 1, 0, -1, -2, -3, ...,
tenemos que parar las llamadas de función a sí misma en el "caso básico".
Nuestro caso básico es imprimir el número 1.

Podemos resumir el cuerpo de una función recursiva a los siguientes pasos:

1. Si es el "caso básico",
   a) hacer la tarea sobre él y
   b) regresar,
2. Hacer la tarea sobre el dato actual,
3. Llamada recursiva para procesar el resto de los datos.
4. Regresar.

Por fin, el código:

```
def contar(n):
    if n == 1:                    # 1. Identificar el "caso básico"
        print(n, "Caso básico")   #   a) hacer la tarea sobre él
        return                    #   b) regresar

    print(n, "y abajo")           # 2. hacer la tarea sobre el dato actual
    contar(n - 1)                 # 3. llamada recursiva para otros n-1 datos
    return                        # 4. regresar

# EJEMPLOS DE USO
contar(3)
```

Este programa procesa los datos desde n hacia 1; si deseamos procesar datos desde 1 hacia n,
es suficiente hacer la tarea sobre el dato actual después del paso de la recursión,
o sea intercambiar el orden de los pasos 2 y 3.

Esto se puede ver en el siguiente ejemplo.

```
def contar(n):
    if n <= 1:                    # 1. Identificar el “caso básico”
        print(n, "Caso básico")   #   a) hacer la tarea sobre él
        return                    #   b) regresar

    contar(n - 1)                 # 2. llamada recursiva para otros n-1 datos
    print(n, "y arriba:")         # 3. hacer la tarea sobre el dato actual
    return                        # 4. regresar

# EJEMPLOS DE USO
contar(3)
```

La declaración “return” es equivalente a “return None”.
De hecho, la última declaración “return” no es necesaria,
al llegar al fin del bloque de cuerpo, la función regresaría None automáticamente.

Ya que sabemos procesar los datos de arriba hacia abajo y al revés,
también podemos procesar datos durante ambos movimientos; y abajo y arriba.
Esto lo muestra el siguiente ejemplo:

```
def contar(n):
    if n <= 1:
        print(n, "Caso básico")
        return

    print(n, "y abajo")
    contar(n - 1)
    print(n, "y arriba:")

# EJEMPLOS DE USO
contar(3)
```

Vamos a ver un caso práctico de función recursiva.
Los demás ejemplos de recursión dejaremos para los ejercicios y para el estudio de los algoritmos,
en la sección 9.2 en la página 184.

La operación “factorial”, n!, está definida para números enteros no negativos de la siguiente manera:

$$0! = 1 \quad 1! = 1 \quad 2! = 2 \cdot 1 = 2 \quad 3! = 3 \cdot 2 \cdot 1 = 6 \quad 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \quad \cdots.$$

La definición para cero es un caso especial.
Para cualquier entero positivo, n, la definición es la misma.

$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1.$$

Esta definición se puede expresar de manera recursiva, separando la tarea
para el número actual, n, y para todos los demás:

$$n! = n \cdot (n-1)!$$

Podemos usar esta separación en procesamiento de n y procesamiento de otros n-1 números,
para escribir una función recursiva:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)

# EJEMPLOS DE USO
for x in range(6):
    print(x, factorial(x))
```

Los pasos: "llamada recursiva para otros n-1 datos", "hacer la tarea sobre el dato actual" y "regresar",
están en la misma linea de código:

```
return n * factorial(n - 1)
```

El "factorial(n - 1)" es la llamada recursiva.
La multiplicación por "n" es hacer la tarea sobre el dato actual y
el "return" es regresar.

Si observamos nuestro ejemplo bien,
podemos ver que tenemos dos multiplicaciones no necesarias.
Dado que 0! = 1 y 1! = 1, no necesitamos hacer dos multiplicaciones por 1.
Podríamos utilizar el número 2 como el caso básico, y
regresar factorial de 0 y 1 como casos especiales, aparte de la función recursiva.

Aprovecharemos esta idea para recordarnos de las funciones anidadas.
La función recursiva es anidada dentro de la función "factorial".

```
def factorial(n):
    def factorial_recursivo(N):
        if N == 2:
            return 2
        return N * factorial_recursivo(N - 1)

    if n == 0 or n == 1:
        return 1
    elif n > 1:
        return factorial_recursivo(n)
    else:
        # Hay que levantar excepción en lugar de imprimir error
        print("ERROR: No calculamos el factorial de un número negativo.")

# EJEMPLOS DE USO
for x in range(6):
```

```
        print(x, factorial(x))

    print(-1, factorial(-1))
```

El uso de función anidada no es necesario.
La función "factorial_recursivo" puede moverse al nivel del modulo, o sea alcance global.

Siendo que "factorial_recursivo" no tiene ninguna otra función "superior" que la llame,
la escondemos dentro de su única función superior: "factorial".

8.1.8. Las Funciones son Objetos

Objetos

Hemos usado el término "objeto" a lo largo de este libro.
Hemos dicho que los tipos de datos son objetos y que también las partes de código,
como son las funciones, los módulos y las clases son objetos.

Vamos a aclarar esta idea de "objeto" aquí de una manera introductoria en relación a funciones.
Después aclararemos la idea un poco más al hablar de los módulos y
finalmente lograremos la aclaración completa cuando estudiemos clases y
el paradigma "programación orientada a objetos".

Vamos a observar un objeto que existe en la naturaleza, por ejemplo, la pelota de fútbol.

Objetos naturales tienen ciertas características, o atributos.
Para que una pelota de fútbol fuera usada en un partido oficial
se definen valores de sus atributos:

ATRIBUTO	VALOR
forma	esfera
material	piel
circunferencia	69 cm
masa	430 g
presión	900 g/cm^2

De esta manera el objeto "la pelota de fútbol" tiene sus atributos y cada atributo tiene su valor.
En Python, estos atributos pasivos se pueden modelar con datos.
La pelota también tiene atributos activos, los atributos de su comportamiento:
como rueda por el piso, como vuela y gira en el aire, como rebota, como se desliza y similares.
Si quisiéramos desarrollar una simulación de un partido de fútbol como un software, un juego,
estos atributos activos se pueden modelar con funciones que manipulan los datos
que describen la posición y orientación de la pelota dentro del espacio.

Para trabajar con atributos de objetos se usa el operador punto: ".".
Por ahora vamos a trabajar solamente don atributos pasivos: los datos.

Si deseamos asignar un valor a un atributo de un objeto dado se usa la asignación:

```
<objeto>.<atributo> = <expresión>
```

La expresión se evalúa y el resultado de esta evaluación se asigna como
el valor de atributo del objeto. O sea, el atributo es una variable que referencia
el resultado de la evaluación de la expresión.

Si deseamos leer el valor de un atributo de la memoria simplemente usamos el operador punto:

```
<objeto>.<atributo>
```

Vamos a ver el uso de operador punto usando una función como el objeto.

Función Como Objeto

Las funciones son objetos que consisten de la lista de parámetros en el encabezado y el cuerpo.
Siendo objetos, podemos asignarles atributos adicionales.

Esto puede ser útil cuando necesitamos que una función retenga alguna información
entre las llamadas a ejecución: que se recuerde que sucedió en la llamada anterior.

Digamos que necesitamos una función que nos proporcione una secuencia de números
que se repiten cíclicamente, un numero en cada llamada.
Por ejemplo, que cuente de cero a dos una y otra vez. Como si controla luces en un semáforo.

```
0 1 2 0 1 2 0 1 2 , ....
```

Tal función necesita memorizar cual fue el número regresado en la ejecución anterior,
para saber cual es el que sigue en la ejecución presente.

Creamos un modulo llamado “ciclo.py” con el contenido:

```
# LA FUNCIÓN
def ciclo_3(restablecer=False):
    if not restablecer:
        ciclo_3.actual = (ciclo_3.actual + 1) % ciclo_3.ESTADOS
        return ciclo_3.actual
    else:
        ciclo_3.actual = ciclo_3.ESTADOS - 1

# ATRIBUTOS ADICIONALES
ciclo_3.ESTADOS = 3
ciclo_3.actual = ciclo_3.ESTADOS - 1

# EJEMPLOS DE USO
for i in range(10):
    print(ciclo_3(), end=" ")
print()

ciclo_3(restablecer=True)
for i in range(8):
    print(ciclo_3(), end=" ")
print()
```

Al definir la función “ciclo_3”, este nombre referencia el objeto tipo función.
Después de definir la función se puede usar la forma

```
<objeto>.<atributo> = <expresión>
```

para definir un atributo nuevo de la función. En el ejemplo, esto se hace en las lineas

```
ciclo_3.ESTADOS = 3
ciclo_3.actual = ciclo_3.ESTADOS - 1
```

Ahora el objeto "ciclo_3" aparte de encabezado y el cuerpo de la función, tiene dos atributos más: una constante "ESTADOS" y una variable "actual".
La constante indica cuantos estados hay y la variable indica cual fue el último estado regresado.

El valor de la variable "actual" inicialmente se asigna al valor de "ESTADOS - 1", el valor máximo, para que en la siguiente llamada a la función, esta empiece generar números desde cero.

Mostramos ejemplos de uso dentro de un ciclo "for".
Después la restablecemos al estado inicial y vemos si en el siguiente ciclo "for" empieza desde cero.

El resultado de la prueba es:

```
$ python3 ciclo.py
0 1 2 0 1 2 0 1 2 0
0 1 2 0 1 2 0 1
$
```

Justo como debe ser.

Recuerden que el prompt "$" se va a ver distinto en su sistema.

Las funciones que memoricen su estado entre llamadas son muy frecuentes.
Para esto Python tiene un tipo de funciones especiales llamadas "funciones generadoras".
Vamos a aprender sobre las funciones generadoras en la parte avanzada del libro.
Por ahora vamos a practicar a asignar los atributos adicionales a las funciones,
ya que esto también es una técnica muy importante al nivel avanzado.

8.1.9. Verificar Tipo de Dato con "isinstance" o "type"

Regresemos a nuestra función "sumar" del inicio de esta sección sobre funciones otra vez:

```
def sumar(a, b):
    return a + b
```

Ya sabemos que los parámetros "a" y "b" no tienen tipo.
Cualquier tipo de objeto puede ser asignado como argumento a los parámetros "a" y "b".

La función "sumar" esta sumando los objetos que recibe a través de los parámetros.
Eso quiere decir que su ejecución será exitosa si "a" y "b" reciben objetos sumables.
De otro modo se levantará una excepción y se reportará con el "Traceback".

Esto nos libera de la necesidad de verificar los tipos de objetos recibidos en las funciones.
Es responsabilidad de quien está llamando la función a mandar argumentos adecuados.

En algunos casos puede ser que el diseñador de la función desea averiguar
cuál es el tipo de objeto recibido por parte de los parámetros de la función y/o
restringir la función para que acepte solamente ciertos tipos de datos.

Esto se puede lograr con las funciones: "isinstance" y "type".

Es muy raro y anti-Pythonico estar condicionando funciones así.

Aunque es muy raro que se diseñen funciones que condicionan tipos de sus parámetros, conocer las funciones "isinstance" y "type" es importante ya que pueden ayudar en el proceso de depurar el código e investigar que sucede y donde está el problema.

La Función "isinstance"

La función, "isinstance", se usa de manera:

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

y compara el objeto con el tipo y devuelve True si concuerdan, o False si no.

```
>>> isinstance(False, bool)
True
>>> isinstance(3, int)
True
>>> isinstance(3.14, float)
True
>>> isinstance("Hola", str)
True
>>> isinstance(1 + 2j, complex)
True
>>>
```

Practique usar la función "isinstance" e intente lograr que produzca la respuesta False.

Si deseamos, por ejemplo, que nuestra función "sumar" sea aplicable solamente a las cadenas, podemos hacer lo siguiente:

```
def sumar_dos_cadenas(a, b):
    if isinstance(a, str) and isinstance(b, str):
        return a + b
    else:
        # Hay que levantar excepción en lugar de imprimir error
        print("ERROR: Esta función acepta solamente las cadenas.")

print(sumar_dos_cadenas("Hola ", "¿Qué tal?"))
```

Aparte de modificar el cuerpo de la función, también cambiamos su nombre para que los programadores que la desean usar sepan lo que hace.

Usar la función "isinstance" es preferible sobre usar "type", por algunos detalles con subtipos, o sea subclases, lo que aprenderemos después. Para trabajar con los tipos de datos incorporados, podemos usar cualquiera de las dos.

La Función "type"

La función "type", se usa de manera:

```
type(<objeto>)
```

y devuelve una cadena, el cual describe el tipo de objeto.

```
>>> type(False)
<class 'bool'>
>>> type(3)
<class 'int'>
>>> type(3.14)
<class 'float'>
>>> type("Hola")
<class 'str'>
>>> type(1 + 2j)
<class 'complex'>
>>>
```

Usualmente usamos la función "type" junto con un operador de comparación.

Podemos comparar tipo de objeto con el nombre de tipo:

```
>>> type(1) == int
True
>>> type(1.23) == float
True
>>> type("Hola") == str
True
>>>
```

Podemos comparar tipo de dos objetos:

```
>>> type(1) == type(2)
True
>>> type(1.23) == type(3.21)
True
>>> type("Hola") == type("Oi")
True
>>>
```

Podemos comparar tipo de dos objetos a través de variables.

```
>>> x = True; y = False
>>> type(x) == type(y)
True
>>>
```

Si deseamos, por ejemplo, que nuestra función "sumar"
sea aplicable sólo a los tipos de datos idénticos,
podemos hacer lo siguiente:

```
def sumar_mismos_tipos(a, b):
    if type(a) == type(b):
        return a + b
```

```
        else:
            # Hay que levantar excepción en lugar de imprimir error
           print("ERROR: Esta función acepta solamente tipos idénticos.")

print(sumar_mismos_tipos(1, 2))
```

En los ejemplos estamos reportando errores usando "print". Esto no es muy Pythonico.
En la sección 8.4 en la página 174 aprenderemos a levantar las excepciones y así
usar el mecanismo de las excepciones, muy importante para el lenguaje Python.

8.1.10. Ejercicios

8.2. Módulos

Hasta ahora, cada uno de nuestros programas fue escrito en un archivo de texto, un modulo,
el cual contiene el código de lenguaje Python y tiene extensión ".py".

El objetivo de esta sección es aprender a hacer programas que consisten de varios módulos,
con el propósito de reusar módulos de un programa para el otro y reducir el tiempo de desarrollo.

Un programa Python consiste de uno o varios módulos, a veces organizados en paquete(s).

Para ejecutar el programa que consiste de varios módulos, uno de ellos es especial,
en sentido de que es el modulo que se invoca para la ejecución del programa.
Este modulo invocado para la ejecución se la llama el "script".

Técnicamente, el script no necesita tener la extensión ".py", puede tener cualquier extensión, o
puede no tener ninguna extensión. Es costumbre nombrar el script con la extensión ".py",
ya que contiene el código de Python y a veces puede ser usado como uno de los módulos.

El script importa los demás módulos de programa y hace uso de código que contienen.
Hemos visto esto en los ejemplos donde usamos los módulos de la librería estándar de Python.

El programa puede consistir solamente de un archivo de código si es muy sencillo.
También es posible que el script importa otros módulos con código adicional, y
de esta forma el programa consiste de varios módulos. Módulos pueden importar otros módulos.

De esta manera el programa puede consistir de muchos módulos, y
entre ellos es solamente un script, el que se va a llamar para que el programa inicie.

Un programa en Python inicia llamando el script para que se ejecute.
Si el script de un programa se llama, por ejemplo, "script_1.py" el programa se ejecuta usando

```
python3 script_1.py
```

Esto invoca el intérprete de Python 3, el cual abre el archivo "script_1.py", y
empieza a leer y ejecutar las declaraciones contenidas en el alcance global del "script_1.py".

Los demás módulos se ejecutan de la misma manera en el momento de ser importados.

8.2.1. Programas de Varios Módulos

Un Ejemplo Sencillo

El siguiente ejemplo es un programa que consiste de dos módulos.
Primero presentaremos el modulo importado, "el_modulo_1.py":

```
# el_modulo_1.py
print('Hola, desde el modulo importado.')
```

El modulo "el_script_1.py", es usado como el script del programa:

```
# el_script_1.py
import el_modulo_1

print('Hola, desde el script.')
```

El programa se ejecuta activando el intérprete "python3", y proporcionando el nombre del script:

```
python3 el_sctipt_1.py
```

El resultado de la ejecución es el siguiente:

```
$ python3 el_script_1.py
Hola, desde el modulo importado.
Hola, desde el script.
$
```

El prompt "$" se va a ver diferente en su sistema. Vamos a analizar lo que sucedió.

Al empezar a ejecutar el script, el intérprete primero encuentra la declaración:

```
import el_modulo_1
```

Esto hace que el intérprete suspende la lectura del script y
empieza a leer y ejecutar el modulo "el_modulo_1.py".
En el modulo, el intérprete encuentra la declaración:

```
print('Hola, desde el modulo importado.')
```

Esta se ejecuta. Con esto termina la lectura del modulo, ya que no tiene más declaraciones.
El intérprete regresa al script y continua con su lectura y ejecución. La siguiente declaración es:

```
print('Hola, desde el script.')
```

El interprete ejecuta esta declaración.
Como ya no hay más declaraciones, la lectura y ejecución del script termina y
con esto termina el programa. El interprete terminó su tarea.

Permanecer en el Modo Interactivo

Si deseamos que el interprete quede en el modo interactivo, después de ejecutar un programa,
tenemos que usar la opción "-i":

```
python3 -i el_sctipt_1.py
```

En este caso el resultado sería.

```
$ python3 -i el_script_1.py
Hola, desde el modulo importado.
Hola, desde el script.
>>>
>>> exit()
$
```

El intérprete termina el programa y queda a nuestro servicio en modo interactivo.
Podemos salir del modo interactivo ejecutando la función "exit", u oprimiendo [Ctrl][d].

Vamos a reorganizar este ejemplo.

El Ejemplo Sencillo, Organizado

En el paradigma "programación estructurada",
todo el procesamiento de datos se organiza en una jerarquía de funciones.
La ejecución empieza en el script con una llamada a la función principal,
la raíz de la jerarquía de las funciones.

Esta vez escribiremos el mismo ejemplo anterior, pero de manera correcta.
El modulo importado "el_modulo_2.py" tiene el siguiente contenido:

```
# el_modulo_2.py


def saluda():
    print('Hola, desde el modulo importado.')
```

El script importado "el_script_2.py" tiene el siguiente contenido:

```
# el_script_2.py
import el_modulo_2


def main():
    print('Hola, desde el script.')
    el_modulo_2.saluda()


main()
```

La recomendación del PEP8 es que antes y después de la definición de una función,
deben de existir dos líneas vacías. Lo respetamos en este caso, pero en los siguientes
vamos a poner solamente un espacio antes y después para no extender este texto.
En los módulos reales, seguiremos respetando la regla de dos líneas vacías.

Al ejecutar este programa se obtiene lo siguiente

```
$ python3 -i el_script_2.py
Hola, desde el script.
Hola, desde el modulo importado.
>>>
```

Al usar la opción "-i", el interprete queda activo. Toda la información del programa está en la memoria.
Podemos ejecutar el programa otra vez:

```
>>> main()
Hola, desde el script.
Hola, desde el modulo importado.
>>>
```

O podemos ejecutar algunas declaraciones,
por ejemplo la función "saluda" del modulo "el_modulo_2.py":

```
>>> el_modulo_2.saluda()
Hola, desde el modulo importado.
>>>
```

O podemos pedir ayuda sobre algunas partes de código que está en la memoria.
La llamada de auxilio:

```
>>> help(el_modulo_2)
```

Produce la respuesta:

```
Help on module el_modulo_2:

NAME
    el_modulo_2

FUNCTIONS
    saluda()

FILE
    /home/i/py/el_modulo_2.py

(END)
```

Para desplazarnos hacia abajo, si hay mucho texto, se oprime [Enter], o se usan las teclas con flechas.
Para salir de la pantalla de ayuda, se oprime la tecla [q].

También podemos pedir ayuda sobre cualquier objeto dentro de los módulos:

```
>>> help(main)
>>> help(el_modulo_2.saluda)
```

Esta manera de pedir ayuda es común durante el proceso de depurar el código y eliminar errores.

Vamos a regresar a analizar la ejecución del programa, sin entrar al modo interactivo:

```
python3 el_sctipt_2.py
```

Al llamar el intérprete, este empieza a leer y ejecutar el script.
La primera declaración es:

```
import el_modulo_2
```

Esto causa que el interprete suspenda la lectura del script y empiece a leer "el_modulo_2.py".
La declaración que encuentra en este modulo es la definición:

```
def saluda():
    print('Hola, desde el modulo importado.')
```

El intérprete evalúa solamente el encabezado, verifica la sintaxis del cuerpo y
guarda esta función como un objeto "función saluda" a la memoria.
Al leer una declaración de función, el intérprete no ejecuta el cuerpo de la función.

El modulo no tiene más declaraciones, así que el interprete regresa a leer y ejecutar el script.

La siguiente declaración es la definición de la función "main":

```
def main():
    print('Hola, desde el script.')
    el_modulo_2.saluda()
```

El intérprete lee la función y la guarda en la memoria.
La siguiente declaración es llamada a la función "main"

```
main()
```

El intérprete ejecuta la función "main", la cual contiene la declaración "print" y llama a ejecutar
la función "saluda", la que fue trasladada a la memoria desde el modulo "el_modulo_2.py".
Al ejecutar la función "saluda", también se completa la ejecución de "main" y el programa termina.

Las declaraciones de importar módulos, normalmente se escriben antes del resto de código.
Cómo ya vimos, existen varias formas de importar un modulo, o algunos nombres globales del modulo.
Estudiaremos estas formas de importar en continuación.

8.2.2. Las Formas de "import"

Declaración "import ...."

Para importar un modulo, la forma más común es:

```
import <nombre_de_modulo>
```

Por ejemplo:

```
>>> import el_modulo_2
>>> el_modulo_2.saluda()
```

a veces el nombre del modulo es muy largo, o el nombre del módulo ya está usado
como el nombre de algún otro objeto y no se puede usar como tal.
En estos casos se puede usar "as" (como),
para definir un nombre sustituto para el nombre del modulo:

```
import <nombre_de_modulo> as <ndm>
```

Por ejemplo:

```
>>> import el_modulo_2 as em2
>>> em2.saluda()

>>>> el_modulo_2.saluda()       # Levanta excepción.
```

Para el nombre sustituto se elige un nombre corto que nos recuerde de cuál modulo se trata. Los acrónimos de dos o tres letras son lo más común en la práctica.

Declaración "from .... import ...."

Si no deseamos importar todo el modulo, sino solamente ciertos nombres, o sea objetos, se usa la siguiente forma

```
from <nombre_de_modulo> import <nombre_de_objeto>
```

Por ejemplo

```
>>> from el_modulo_2 import saluda
>>> saluda()
Hola, desde el modulo importado.
>>>
```

Si el nombre del objeto importado es largo, o el nombre del objeto ya está usado como el nombre de algún otro objeto y no se puede usar como tal.
podemos aplicar "as" para definir un nombre sustituto para el nombre de objeto.

```
from <nombre_de_modulo> import <nombre_de_objeto> as <ndo>
```

Por ejemplo

```
>>> from el_modulo_2 import saluda as hola
>>> hola()
Hola, desde el modulo importado.
>>>
```

Declaración "from ... import *"

La declaración de importación de forma

```
from <module_name> import *
```

importa todos los objetos, definidos en el alcance global, o sea al nivel de modulo, EXCEPTO aquellos objetos cuyo nombre empieza por uno o varios símbolos de subrayado: _.

Vamos a escribir modulo con el nombre "estrella.py" con el contenido:

```
variable = 1
_variable = 2
```

Al importar este modulo usando estrella, "*", sucede lo siguiente:

```
>>> from estrella import *
>>> variable
1
>>> _variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name '_variable' is not defined
>>>
```

Los Pythonistas parecen divididos en opinión sobre esta forma de importar. En mi opinión, esta forma de importar debe de evitarse, ya que lleva mucho riesgo de colisión de nombres.

El uso del carácter subrayado como el primer símbolo aclararemos antes de terminar esta sección.

8.2.3. El "__pychache__" y los Archivos ".pyc"

El Código en Bytes ("byte code")

Cuando llamamos el intérprete de Python a ejecutar un "script.py", suceden varios pasos hasta que se empiece a ejecutar el programa.

El intérprete primero lee el script y los módulos importados, uno por uno, y genera archivos con la extensión ".pyc", o sea ".py compilado".
Estos archivos contienen el código Python, traducido a "byte code".

El "byte code" son instrucciones de bajo nivel, casi nivel de la máquina, pero independientes de la plataforma, o sea del sistema operativo y del hardware.

El nombre de archivo de "byte code" es formado de la manera que concatena el nombre y la versión del intérprete al nombre del modulo.
Por ejemplo, con el intérprete CPython, versión 3.4, el modulo

```
el_modulo_1.py
```

se traduce a

```
el_modulo_1.cpython-34.pyc
```

Cuando el programa termina, el archivo ".pyc" del script es automáticamente borrado.

Los archivos ".pyc" de los módulos importados, se guardan en el subdirectorio, "__pycache__", generado en el mismo directorio donde se encuentra el archivo del script.

Si el programa se ejecuta otra vez, los módulos importados no se compilan otra vez, se utilizan los archivos correspondientes ".pyc" ya compilados.

De hecho, para decidir si compilar de nuevo a un archivo ".pyc", el interprete verifica dos cosas:
Primero verifica el nombre del modulo ".pyc", por ejemplo "el_modulo_1.cpython-34.pyc", para asegurarse que este archivo corresponde al mismo intérprete y su versión, y en seguida verifica que la fecha y la hora de creación de archivo ".py" precede la del archivo ".pyc".
O sea, verifica que el archivo ".py" no fue modificado desde la ultima traducción al "byte code".

En caso de que un modulo fue modificado y tenga la hora más nueva, este archivo se compila otra vez.

Existe modulo "py_compile" para compilar módulos manualmente.
https://docs.python.org/2/library/py_compile.html

```
>>> import py_compile
>>> py_compile.compile("el_script_2.py")
'__pycache__/el_script_2.cpython-34.pyc'
>>>
```

También se puede usar el modulo "compileall", pero con cuidado.

8.2.4. Los Módulos son Objetos

Ya presentamos algunos detalles sobre el concepto de objeto en la subsección 8.1.8 en la página 153.

Ahora vamos a ver en que sentido un modulo también es un objeto de Python.
Vamos a usar un modulo para simular un objeto y sus atributos.
El objeto va a ser abstracto, matemático: un punto en el espacio de dos dimensiones.

Un punto matemático no tiene tamaño, ni material, ni color. Ningún atributo físico.
Lo único que tiene son sus coordenadas "x" y "y" en el espacio.

Vamos a hacer este punto activo, movible. Va a saber a cambiar su posición,
va a saber a regresar a su "casa": el origen de sistema de coordenadas, y
va a saber reportar donde se encuentra, imprimiendo sus coordenadas en la pantalla.

De momento nos vamos a olvidar de nuestra recomendación Pythonista
de evitar el uso de las variables globales manipuladas desde las funciones.

Nuestro modulo que simula el objeto "punto" lo llamaremos "punto_modulo.py".
Su contenido puede ser:

```
x = 0.0
y = 0.0

def mover_a(coordenada_x, coordenada_y):
    global x, y
    x = coordenada_x
    y = coordenada_y

def mover_al_origen():
    mover_a(0.0, 0.0)

def imprime_posición():
    print("(", format(x, '.3f'), ", ", format(y, '.3f'), ")", sep="")
```

Los atributos de un modulo son los nombres definidos en alcance global, o sea a nivel del modulo.
En este caso, los atributos del modulo "punto_modulo.py" son:
variables: "x", "y"
funciones: "mover_a", "mover_al_origen", "imprime_posición".

Ahora vamos a escribir el modulo "punto_modulo_script.py" para usar el modulo "punto_modulo.py".
El contenido del script "punto_modulo_script.py" es:

```
import punto_modulo as pm

# USO DE LAS VARIABLES
print("Coordenada x:", pm.x)
print("Coordenada y:", pm.y)

# USO DE LAS FUNCIONES
pm.imprime_posición()
pm.mover_a(2.0, 3.5)
pm.imprime_posición()
pm.mover_al_origen()
pm.imprime_posición()
```

Como vemos, para obtener valor de un atributo, se usa el operador punto:

```
<objeto>.<atributo>
```

Cada atributo en el ejemplo tiene como prefijo "pm.",
donde el "pm" es nombre sustituto del objeto "punto_modulo.py".

El resultado de la ejecución del script es:

```
i@i28:~/py$ python3 punto_modulo_script.py
Coordenada x: 0.0
Coordenada y: 0.0
(0.000, 0.000)
(2.000, 3.500)
(0.000, 0.000)
i@i28:~/py$
```

Este es un ejemplo no recomendable.
Solamente sirvió para indicar en que sentido los módulos son objetos en Python.
Las razones por las cuales este ejemplo no es recomendable son:
Primeramente hay que evitar el uso de variables globales en los módulos y
lo más importante es que no hay que usar módulos para simular objetos.

Para modelar los objetos vamos a usar un concepto llamado "clase".
Las clases son moldes para generar muchas instancias de objetos de mismo tipo.

Si tenemos una simulación con 100 puntos y usamos módulos para representar puntos,
tendríamos que programar 100 módulos parecidos a "punto_modulo.py".
Cualquier cambio en comportamiento de puntos, tendríamos modificar 100 veces.

En contraste, si usamos "clase" para modelar el punto, escribiríamos solamente una clase y
de manera sencilla podríamos crear tantas instancias de esta clase cuantas queremos.
Cualquier cambio necesario se haría en solamente un lugar: en la clase.
Esto lo vamos a estudiar después.

De aquí en adelante, ya que entendimos en qué sentido los módulos son objetos,
nos vamos a enfocar a la manera correcta de usar los módulos.

8.2.5. Los Módulos y las Funciones Se Diseñan Para el Reuso

Cuando estamos programando bajo el paradigma "programación estructurada",
nos basamos en una jerarquía de funciones para componer el programa.
Esta jerarquía de funciones se puede distribuir en varios módulos,
de manera que algunos módulos pueden ser reusados para otros programas.

Para facilitar el reuso de un modulo le vamos a hacer lo siguiente:
1) Escribir la cadena de documentación, "docstring", al inicio del modulo.
2) Escribir el "docstring" para cada función, entre su encabezado y su cuerpo.
3) Condicionar la ejecución de los ejemplos de uso y las pruebas al final del modulo
dependiendo si el modulo se usa como script, o no.

Los "docstrings" y el Nombre ("__name__") del Modulo

En la subsección 8.1.8 en la página 153 diseñamos el modulo llamado "ciclo.py".
Ahora lo vamos a modificar para que esté preparado para el reuso y renombrarlo a "ciclo_3.py".

```
""" Este modulo está hecho para los propósitos educativos.
    Demuestra como agregar atributo a una función y
    así lograr que la función memorice su estado entre ejecuciones.
"""

def ciclo_3(restablecer=False):
    """ La función genera ciclo de valores 0 a 2,
        memorizando su estado y regresando un valor en cada llamada.
        La repetición se restablece a cero al recibir el argumento True.
    """
    if not restablecer:
        ciclo_3.actual = (ciclo_3.actual + 1) % ciclo_3.ESTADOS
        return ciclo_3.actual
    else:
        ciclo_3.actual = ciclo_3.ESTADOS - 1

ciclo_3.ESTADOS = 3
ciclo_3.actual = ciclo_3.ESTADOS - 1

if __name__ == "__main__":
    # EJEMPLOS DE USO
    ciclo_3(restablecer=True)
    for i in range(20):
        print(ciclo_3(), end=" ")
    print()

    # PRUEBAS
    prueba = True

    # Restablecer conteo desde cero
```

```
    ciclo_3(True)

    # Comparar los valores regresados
    for i in range(20):
        if ciclo_3() != i% ciclo_3.ESTADOS:
            prueba = False

    # Reportar los resultados de la prueba
    if prueba is False:
        print("\nPruebas NO cumplidas.\n")
    else:
        print("\nPruebas CUMPLIDAS.\n")
```

Antes de comentar las novedades relacionadas al modulo,
observen que una ventaja de los atributos adicionales de función, como el "ciclo_3.ESTADOS", es que los podemos usar y dentro y fuera de la función, por ejemplo en las pruebas.

Ahora, sobre el modulo. Al inicio del modulo se escribe el "docstring" del modulo, y entre el encabezado y el cuerpo de la función se escribe el "dosctring" de la función. Estos "docstring" aparecen en la página de ayuda. Al ejecutar

```
>>> import ciclo_3
>>> help(ciclo_3)
```

se obtiene la página de ayuda:

```
Help on module ciclo_3:

NAME
    ciclo_3

DESCRIPTION
    Este modulo está hecho para los propósitos educativos.
    Demuestra como agregar atributo a una función y
    así lograr que la función memorice su estado entre ejecuciones.

FUNCTIONS
    ciclo_3(restablecer=False)
        La función genera ciclo de valores 0 a 2,
        memorizando su estado y regresando un valor en cada llamada.
        La repetición se restablece a cero al recibir el argumento True.

FILE
    /home/i/py/ciclo_3.py

(END)
```

Para moverse dentro de la página de ayuda se usa la tecla [Enter], o las teclas con flechas. Para salir de la página de ayuda se oprime la tecla [q], "quit".

Ahora analicemos la declaración "if" que se encuentra en el código:

```
if __name__ == "__main__":
```

Durante la ejecución de un programa, el intérprete se refiere a cada modulo por su nombre,
EXCEPTO en el caso del modulo invocado como el script. Al modulo invocado como el script,
el intérprete se refiere como si este tuviera el nombre "__main__".
Este nombre empieza y termina con dos símbolos subrayado: __.

Cada modulo tiene un atributo que se llama "__name__" (nombre) asignado por el intérprete, y
este atributo tiene como el valor el nombre del modulo, tal como lo ve el intérprete.
Si el modulo "ciclo_3.py" es importado, para el intérprete su nombre es asignado como

```
__name__ = "ciclo_3"
```

Si el modulo "ciclo_3.py" se ejecuta como un script, para el intérprete su nombre es asignado como

```
__name__ = "__main__"
```

Durante el desarrollo de un modulo lo ejecutamos varias veces como el script.

Primero escribimos las pruebas que el código del modulo debe pasar.
Luego empezamos a escribir el contenido del modulo.
Ejecutamos el modulo como el script las veces necesarias para eliminar los erorres,
hasta que el modulo pase las pruebas. Al fin añadimos los ejemplos de uso,
para que otros programadores entiendan la intención del diseño del modulo.

Al ejecutar un modulo como el script, durante las pruebas, la condición en la declaración

```
if __name__ == "__main__":
```

evalúa a True y las pruebas que siguen esta condición se ejecutan.

Después de confirmar que el modulo funciona como esperado,
estas pruebas se dejan en el modulo para que otros programadores vean
cual es la intención de uso de las variables y funciones del modulo y como fue probado.

Al importar el modulo desde un script, la condición "__name__ == "__main__"" evalúa a False y
los ejemplos de uso y las pruebas no se ejecutan. Es como si no están ahí.

Existe otro uso de la declaración "if __name__ == "__main__":";
su uso dentro de un script.

Un modulo puede ser un script en un programa, pero puede ser un modulo importado en el otro.
Por esta razón en los módulos intencionados de ser usados de ambas maneras,
el lugar de terminar el modulo con línea

```
main()
```

terminamos el modulo con líneas

```
if __name__ == "__main__":
    main()
```

De esta manera la función "main" se va a ejecutar solamente si el modulo es invocado como el script y
no se va a ejecutar si el modulo es importado.

Vamos a crear un script que usa el modulo "ciclo_3.py". El script se llama "ciclo_3_script.py":

```
import ciclo_3 as c3

def main():
    for i in range(15):
        print(c3.ciclo_3(), end=" ")
    print()

    c3.ciclo_3(restablecer=True)
    for i in range(12):
        print(c3.ciclo_3(), end=" ")
    print()

if __name__ == "__main__":
    main()
```

En este caso estamos usando el modulo "ciclo_3.py",
justo como en los ejemplos de uso del mismo modulo lo recomiendan.

Resultados de la ejecución del script "ciclo_3_script.py" son:

```
i@i28:~/py$ python3 ciclo_3_script.py
0 1 2 0 1 2 0 1 2 0 1 2 0 1
0 1 2 0 1 2 0
i@i28:~/py$
```

Nombres Globales que Empiezan con Subrayado(s)

Vamos a desarrollar el segundo ejemplo: un modulo que calcula los factoriales y
lo vamos a usar después desde otro modulo que calcula las permutaciones y las combinaciones.

El modulo "factorial.py" tiene el siguiente contenido

```
""" Este modulo es dedicado a la calculación de los factoriales. """

def _factorial_recursivo(N):
    """ Función auxiliar para la función "factorial".
        Calcula los factoriales para números mayores que 1,
        de manera recursiva.
    """
    if N == 2:
        return 2
    return N * _factorial_recursivo(N - 1)

def factorial(n):
    """ Calculo de factoriales para enteros no negativos.
        Para los enteros negativos se reporta error en la pantalla y
        se regresa None.
        Otros tipos de datos se ignoran; se regresa None.
    """
    if type(n) == int:
```

```
        if n == 0 or n == 1:
            return 1
        elif n > 1:
            return _factorial_recursivo(n)
        else:
            # Hay que levantar excepción en lugar de imprimir error
            print("ERROR: No calculamos el factorial de un número negativo.")

if __name__ == "__main__":
    # EJEMPLOS DE USO
    print("n n! ")
    for x in range(6):
        print(x, factorial(x))

    # PRUEBAS
    prueba = True
    if factorial(0) != 1 or factorial(1) != 1:
        prueba = False
    if factorial(3) != 6 or factorial(5) != 120:
        prueba = False
    if factorial(-1) is not None or factorial(1.23) is not None:
        prueba = False
    # Reportar los resultados de las pruebas
    if prueba is False:
        print("\nPruebas NO cumplidas.\n")
    else:
        print("\nPruebas CUMPLIDAS.\n")
```

La novedad en este ejemplo es que el nombre de la función "_factorial_recursivo" empieza con el subrayado.

Esto señala a los demás programadores que esta función no fue diseñada con la intención de ser llamada desde un script, ni desde ningún otro modulo.
El subrayado como el primer símbolo señala que la función es nada más una función auxiliar la cuál está apoyando alguna otra función en el modulo, en este caso la función "factorial".

La función "factorial" es diseñada con la intención de ser usada desde otros módulos, pero la función "_factorial_recursivo", por iniciar con el símbolo de subrayado, es señalada como auxiliar que no debe ser llamada desde ningún otro modulo.

El Python no va a prevenir el uso de funciones que empiezan con el subrayado desde otros módulos, aunque va a apoyar esta idea de manera que los nombres que empiezan con subrayado no se importarán en el script cuando el modulo se importa usando la forma

```
from <nombre_del_modulo> import *
```

Otras formas de importar incluyen las funciones que inician con el subrayado también.
La idea es que a veces un modulo puede ser usado por parte de otros programadores de maneras creativas que el diseñador original del modulo no había previsto.
Así que no hay que ser demasiado restrictivos.

Si el diseñador de este modulo hubiera deseado definitivamente no permitir llamadas
a la función "_factorial_recursivo" desde otros módulos,
la podía haber programado como interna, anidada dentro de la función "factorial".
En este caso el nombre "_factorial_recursivo" no seria una variable global;
no seria un atributo del modulo y no podría ser visto desde otros módulos.
Ni siquiera necesitaría empezar con el subrayado; podría llamarse "factorial_recursivo".

8.2.6. Detalles Importantes Sobre los Módulos

Pruebas

Las pruebas ("testing") es una parte muy importante en el desarrollo de software.
Compañías de software tienen personas especializadas en esta actividad.
La teoría y la práctica de pruebas se han desarrollado mucho en las ultimas décadas.

El lenguaje Python esta diseñado para el uso extensivo de las pruebas en los proyectos.
Existen módulos y librerías de módulos que sirven como herramientas para
las técnicas modernas de pruebas: sistemáticas, bien planeadas, y organizadas.

Existen las metodologías de desarrollo de software basadas en pruebas como su parte esencial.

En esta sección sobre módulos, a penas hemos tocado el superficie del tema de las pruebas.
Esto es suficiente por ahora ya que a penas estamos aprendiendo el lenguaje.
Vamos a hablar más sobre pruebas en la parte avanzada de este curso.

Reasignación de Variable Global

Ya hemos mencionado en la subsección 8.1.5 en la página 142,
del peligro de reasignación de las variables globales dentro de los niveles locales de las funciones.
Vimos que esto es a veces aceptable, usando la declaración "global", pero debe evitarse.

Existe otro lugar donde se puede reasignar una variable global de un modulo.
Es la reasignación de variable global desde un otro modulo el cuál importa el primero.
En Python esto es técnicamente posible, pero hay que evitarlo a toda costa.
Es mejor rediseñar los módulos, que modificar variable global de un modulo desde algún otro.

Esto lleva a dos tipos de problemas:
Es muy difícil encontrar las causas de problema cuando variable global tiene valor no esperado.
Ya es difícil buscar error en el mismo modulo; tener que revisar todos los módulos es un horror.
También existe el problema de reusar un modulo si su funcionamiento depende de un otro modulo.
Esto va en contra de la idea de diseñar software en módulos con el propósito de su reuso.

Sigue el ejemplo simple de lo que no hay que hacer.

El modulo "m_1.py" tiene una variable global:

```
var = 1
```

Otro modulo, "s_1.py", importa el modulo "m_1.py" y modifica su variable global.

```
import m_1

m_1.var = 2
```

Esto hay que evitarlo uniendo estos dos módulos en uno sólo, o
rediseñar los módulos de alguna otra manera.

Importación Múltiple y Modificación Dinámica

La importación múltiple es cuando un modulo intenta importar algún otro varias veces,
de manera directa, o indirecta.

Por ejemplo, el modulo "c.py" importa los modulos "a.py" y "b.py", pero no sabe que
el modulo "b.py" también importa el modulo "a.py".
De esta maneara el modulo "c.py" importa el modulo "a.py" dos veces:
una vez directamente y otra vez indirectamente a través de importar el modulo "b.py".

Para el intérprete de Python esto no es el problema,ya que
cada vez cuando se le pide importar un modulo,
el intérprete verifica si la ultima versión del modulo ya está importada.
Si la última versión ya está importada, el Python no lo va a importar otra vez.

Cabe decir que en Python es posible modificar el código de un modulo de manera dinámica,
durante la ejecución de programa, e importar, o sea cargar este modulo de nuevo.
De esta manera podemos diseñar los programas capaces de modificarse a sí mismos.
Después del todo, un modulo es simplemente un archivo de texto
el cuál podemos modificar desde algún otro modulo.
Esto ya es un tema muy avanzado, así que lo dejamos hasta aquí.

La Función "dir"

Para ver el contenido de un modulo y averiguar cuales atributos tiene disponibles para importar,
se puede usar la función "dir".

Por ejemplo, para ver los atributos que ofrece el modulo "punto_modulo.py".
podríamos hacer lo siguiente:

```
>>> import punto_modulo
>>> dir(punto_modulo)
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__',
'imprime_posición', 'mover_a', 'mover_al_origen', 'x', 'y']
>>>
```

Se obtiene la lista de los atributos de este modulo.

Los atributos que empiezan y terminan con el doble subrayado son agregados por el intérprete.
Vamos a aprender el significado y el uso de estos atributos en la parte avanzada de este curso.

Por ahora nos vamos a enfocar a los atributos que no tienen doble subrayado.
Estos son los que vamos a usar desde un script, u otro modulo que importa a "punto_modulo.py".
Entre estos, no se distinguen las funciones de las variables. Hay que pedir ayuda.

Si pedimos ayuda sobre este modulo, obtenemos lo siguiente:

```
Help on module punto_modulo:

NAME
    punto_modulo

FUNCTIONS
    imprime_posición()

    mover_a(coordenada_x, coordenada_y)

    mover_al_origen()

DATA
    x = 0.0
    y = 0.0

FILE
    /home/i/py/punto_modulo.py

(END)
```

Este modulo no tiene las cadenas de documentación escritas.
Hay que agregar los "docstrings" del modulo y de cada función,
para que la pantalla de ayuda sea de mejor calidad.

Importación Circular

FALTA INVESTIGAR!!!

8.2.7. Ejercicios

NN) Escribe los "docstrings" del modulo "punto_modulo.py" y sus funciones.

8.3. Datos en Archivos de Texto

8.4. Excepciones

Mecanismo de excepción nos permite diseñar programas robustos a situaciones excepcionales y producir códigos sencillos y fáciles de entender.

Es importante tener en mente que el código se escribe para varios propósitos; para que:

- Cumpla con los requerimientos del usuario del programa.
- Sea aceptado y ejecutado correctamente por parte del intérprete de Python.
- Sea fácil de mantener: entender, modificar y poner a pruebas.
- Sea hecho de componentes reusables para otros proyectos.

En caso de la "programación estructurada" los componentes reusables son módulos y funciones.
En caso del paradigma "orientado a objetos" las componentes son módulos, clases y funciones.
El mecanismo de excepciones nos permite diseñar las componentes reusables que
cumplan con todos los propósitos mencionados.

Como siempre, aprenderemos usar excepciones a base de un ejemplo sencillo, un poco irreal,
para enfocar nuestra atención a las excepciones y después expandiremos las posibilidades,
hacia los casos reales y reusables.

Empezamos con un ejemplo de una función sencilla, vamos a reusar la función "reciproco".
Nuestra última versión quedó así:

```
print("Este programa calcula el recíproco de un entero.")
número = int(input("¿Para cuál número desea el reciproco? "))

if número != 0:
    print("El recíproco de", número, "es:", 1.0 / número)
else:
    print("El recíproco de 0 no existe.")
print("Gracias por usar este programa.")
```

Esto es un programa completo. Ahora, el objetivo es convertir este programa a un código
que pueda servir como un componente reusable en varios programas.

El primer paso sería poner el calculo del recíproco en una función que devuelve el resultado,
de manera que la función no se comunique con el usuario, o sea que no imprime nada en la pantalla.

Los programadores que van a usar nuestra función tal vez desean comunicar el resultado a la pantalla.
Los que no lo desean, serán felices de que la función simplemente devuelva el resultado, y
los que si lo desean, serán felices también, ya qué pueden formatear el resultado como ellos desean.

Entonces tenemos que eliminar las funciones "print" de este código.
El último "print" de agradecimiento no es necesario, simplemente lo eliminamos.
El "print" en el bloque de "if", lo reemplazamos con "return", para no dirigir resultado a la pantalla,
sino que se regrese a la expresión que llamó la función. Vamos a mantener la función en su modulo.

Estos cambios resultan en modulo "recíproco_.py":

```
def recíproco(n):
    if n != 0:
        return 1.0 / n
    else:
        print("El recíproco de 0 no existe.")

if __name__ == "__main__":
    # EJEMPLO DE USO
    for número in range(-4, 5):
        print(recíproco(número))
```

Todavía queda la cuestión ¿que vamos a hacer en el bloque "else",
cuando el recíproco de "n" no existe y no podemos devolverlo?
No debemos imprimir nada en la pantalla, pero algo tenemos que regresar;
de alguna manera tenemos que señalar que no fue posible producir el resultado.

Aparentemente, hay una solución sencilla a este problema.
Podríamos regresar el valor None como la señal que no hay resultado:

```
        else:
            return None
```

Desde nuestro punto de vista, encontramos la solución, pero
¿qué pasa desde el punto de vista del programador que va a usar nuestra función?

En lugar de simplemente llamar la función

```
y = recíproco(x)
```

este programador tiene que estar verificando si la respuesta es None, o tiene el resultado,
para saber como continuar con el procesamiento de datos.
Tiene que hacerlo justo ahí donde hace la llamada a nuestra función:

```
y = recíproco(x)
if y is None:
    ....
else:
    ....
```

Con nuestra idea de simplemente regresar el None en caso de la falta del resultado,
estamos complicando código en todos los lugares de todos los programas que usarán nuestra función.

Si este código que usa nuestra función es parte de alguna función, digamos "f", ahora es posible
que la función "f" también tiene que señalar la excepción a la expresión que la llamó.
Entonces se genera código adicional en todas las funciones que llaman "f".
Y si el código que llama "f" es parte de una función "g", entonces en la función "g", ...
Ya se imaginan como sigue el cuento.

Y esto no es todo. Nuestra función no va a ser la única que tenga un caso de excepción.
Habrá muchas funciones que a veces no pueden producir el resultado, o no pueden ejecutar la acción:
abrir un archivo, o conectarse al Internet, o conectarse a una base de datos en la red local, etc.

La consecuencia de los casos de excepción es que se tienen que hacer muchas lineas de código,
adicionales al código que simplemente cumpla con los algoritmos deseados.

Y ahora, ¿quién puede ayudarnos?
El mecanismo de excepción.

El quién lo inventó es un genio. Mi héroe.
Me salva de horas y horas de programación de código adicional que procesaría los casos de excepción.
Con el mecanismo de excepción, procesar los casos de excepciones se vuelve muy sencillo y
lo más importante: el código de los algoritmos queda limpio, fácil de entender, mantener y reusar.

8.4.1. Levantamiento y Procesamiento de Una Excepción

El mecanismo de excepción tiene dos partes:

1. levantamiento de una excepción
2. procesamiento de la excepción

La situación que vamos a analizar primero es cuando
el levantamiento está en el código del componente que va a ser reusado y
el procesamiento está en el código que va a reusar el componente.

Esto no es necesariamente siempre así. A veces el procesamiento de la excepción no existe, y a veces y el levantamiento y el procesamiento están en el mismo modulo, o hasta en la misma función.
Vamos a ver los ejemplos de todas las posibilidades.

Primero nos vamos a enfocar a lo que necesitamos hacer en nuestra función: levantar una excepción.
Después veremos como el programador que usa nuestra función puede procesar este levantamiento.

Levantamiento ("raise") de Una Excepción

Para señalar un caso de excepción, se usa la declaración "raise" (levantar):

```
raise <objeto>
```

La palabra "raise", similar al "return", es seguida por un objeto que se devuelve.

El "return" regresa el objeto a la expresión que la llamó, una función "atrás".
El "raise" regresa el objeto hasta la función que va a procesar la excepción,
lo que puede ser una o varias funciones "atrás", o hasta salir del programa y reportar "Traceback".

Usualmente el objeto lleva la información sobre que tipo de excepción sucedió y donde.
Existen cuatro opciones para formar el objeto que va a regresar la declaración "raise".

La primera opción y la más sencilla, es usar una cadena de caracteres, por ejemplo así:

```
def recíproco(n):
    if n != 0:
        return 1.0 / n
    else:
        raise "División entre cero en la función \"recíproco\"."

if __name__ == "__main__":
    # EJEMPLO DE USO
    for número in range(-4, 5):
        print(recíproco(número))
```

Esta opción se usa en raras ocasiones; a veces durante el desarrollo como opción temporal.
Normalmente se reemplaza por alguna de las siguientes opciones.

Existen objetos integrados al lenguaje Python con propósito de ser devueltos con el "raise".

Como se imaginan, no somos los primeros que se enfrentan con la excepción de dividir entre cero.
Los diseñadores de Python ya han previsto esta excepción y otras excepciones y
ya han creado una jerarquía de objetos que sirven para ser usados con el "raise".
Pueden ver la jerarquía en la sección 22.5 en la página 207.

El objeto que nos conviene en este caso es el "ZeroDivisionError" (error de división entre cero).
En consecuencia, la segunda opción que tenemos es, levantar la excepción así:

```
else:
    raise ZeroDivisionError
```

Esta opción tiene varias ventajas para el programador que va a llamar nuestra función.
Por lo pronto, por el tipo de objeto que recibió ya va a saber de que tipo de excepción se trata.
Otras ventajas veremos a rato al analizar su parte de código.

La tercera y todavía más recomendable opción es combinar las dos anteriores:

```
else:
    raise ZeroDivisionError("Función \"recíproco\".")
```

De esta manera el programador recibe otra vez las dos partes de información: que sucedió y donde.

La cuarta opción, y la más adecuada en proyectos granes, es que diseñemos nuestro propio objeto para señalar excepciones. Esto lo vamos a hacer
al empezar a estudiar el paradigma de "programación orientada a objetos".
Hasta se puede diseñar propia jerarquía de objetos tipo "...Error", subclases de "Excepction".

Por ahora nos quedamos con la tercera opción en cuanto la cuestión de excepción y
al nuestro modulo le vamos a agregar la capacidad de manejar los números flotantes también.

Sobre la función "_is_close", o sea "isclose", pueden leer en 8.6.1 en la página siguiente.

Ejecuten el código y observen la última línea del reporte "Traceback".

```
# Modulo: recíproco_.py
""" Este modulo contiene la función para calcular el recíproco. """

def recíproco(n, tol_abs=1e-12):
    """ Calcula el recíproco del primer argumento recibido.
        Levanta "ZeroDivisionError" en caso de recibir cero.
        La tolerancia absoluta para identificar cero es de 12 decimales.
        Se puede controlar la tolerancia modificando el segundo argumento.
    """
    if _is_close(n, 0.0, TOL_REL=0.0, TOL_ABS=tol_abs):
        raise ZeroDivisionError("Función \"recíproco\".")
    else:
        return 1.0 / n

def _is_close(a, b, TOL_REL=1e-15, TOL_ABS=1e-15):
    return abs(a - b) <= max(TOL_REL * max(abs(a), abs(b)), TOL_ABS)

if __name__ == "__main__":
    # EJEMPLO DE USO
    for número in range(4, -1, -1):
        print(número, ":  ", format(recíproco(número, 1e-6), '.4f'), sep="")
```

8.4.2. Procesamiento de Excepciones

La Declaración "try-except-else-finally"

¿Cómo el programador que llama nuestra función "recíproco" puede procesar esta excepción?

Supongamos que su programa tiene las siguientes funciones:

```
# Modulo: recíproco_script.py
import recíproco_ as r

def f():
    for i in range(4, -1, -1):
        print(format(i, '>3d'), end="")
        resultado = r.recíproco(i)
        print(format(resultado, '>10.6f'))

def main():
    print(" x recíproco")
    print("=" * 15)
    f()
    print("-" * 15)

if __name__ == '__main__':
    main()
```

Si decide no procesar las excepciones el resultado es un "Traceback"
al momento de intentar de calcular el recíproco de cero.
Ejecute el script "recíproco_script.py" y analice el "Traceback",
línea por línea de abajo hacia arriba.

8.5. Librería Estándar

8.5.1. El Modulo "tkinter", Interfaces Gráficas de Usuario

8.6. Recomendaciones y Ejemplos

8.6.1. Ejemplos

El Problema de la Igualdad de Dos Números flotantes

En la sección 6.1.6 en la página 71 ya explicamos que
no es confiable comparar dos números flotantes usando operadores == y !=.

Por ejemplo, normalmente esperamos que 2.2 * 3 = 6.6 y 3.3 * 2 = 6.6 sean números iguales.
Pero, al calcular las expresiones y comparar los resultados nos espera una sorpresa:

```
>>> (3.3 * 2) == (2.2 * 3)
False
>>>
>>> 3.3 * 2
6.6
>>> 2.2 * 3
6.6000000000000005
>>>
```

La solución es identificar si la diferencia entre dos flotantes es suficientemente pequeña,
en términos relativos, o absolutos, o en la cantidad de representaciones cercanas,
lo que depende de algoritmo y de lo que se quiere lograr.

En la libraría estándar del Python 3.5 ya existen funciones

```
math.isclose
cmath.isclose
```

para comparar dos flotantes hasta la tolerancia relativa, o absoluta, deseada.

Los que trabajan en Python 3.4, o anterior, pueden usar la siguiente función
para comparar dos flotantes hasta la tolerancia relativa TOL_REL, o absoluta TOL_ABS deseada:

```
def isclose(a, b, TOL_REL=1e-9, TOL_ABS=1e-6):
    return abs(a - b) <= max(TOL_REL * max(abs(a), abs(b)), TOL_ABS)

# PRUEBAS
x = 2.2 * 3
y = 3.3 * 2

# Para usar la tolerancia relativa, la ajustamos a la deseada y
# bajamos la absoluta a cero.
if isclose(x, y, TOL_REL=1e-15, TOL_ABS=0.0):   # Sustituye  if x == Y:
    print(True, "relativa")

# Para usar la tolerancia absoluta, la ajustamos a la deseada y
# bajamos la relativa a cero.
if isclose(x, y, TOL_ABS=1e-15, TOL_REL=0.0):   # Sustituye  if x == Y:
    print(True, "absoluta")
```

Hay que estudiar los “métodos numéricos” y “análisis numérico” para adecuar la función
a las necesidades de lo que se quiere lograr.

Los que usan la librería numpy tienen disponible la función

```
numpy.allclose()
```

para comparar números y arreglos de números hasta cierta precisión relativa o absoluta.

Recursión 1

Escriba una función recursiva que recibe como argumento el número n, entre uno y nueve, y
escriba una línea en la pantalla desde 1 hasta n y atrás hasta el 1.
Por ejemplo si n = 3, el programa escribe línea: 1 2 3 4 3 2 1
Haz la prueba de la función pidiendo el usuario el numero n de [1, 9].

Cantidad de Combinaciones

Vamos a desarrollar función: combinaciones, la cual calcula la cantidad de combinaciones,
o sea la cantidad de subconjuntos de tamaño n dentro de un conjunto de tamaño N.

La cantidad de subconjuntos, o sea combinaciones C es una función de N y k:

$$C = \left(\begin{array}{c} N \\ k \end{array} \right) = \frac{N!}{k! \cdot (N-k)!}$$

donde el signo de exclamación es la operación "factorial":

$$N! = N \cdot (N-1) \cdot (N-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

$$k! = k \cdot (k-1) \cdot (k-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1$$

con la definición como sigue:

$$0! = 1 \qquad 1! = 1 \qquad 2! = 2 \cdot 1 = 2 \qquad 3! = 3 \cdot 2 \cdot 1 = 6 \qquad 4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24 \cdots$$

La función para factorial de hecho ya existe en la librería "math", como "math.factorial(x)".

```
>>> import math
>>> math.factorial(4)
24
>>>
```

Vamos a practicar a desarrollar nuestra propia.vamos a practicar a desarrollar nuestra propia, con el propósito de aprender
sobre las variables no-locales.

Primero definimos las funciones independientemente.
La función factorial tiene un caso especial: 0!= 1,
todos los demás enteros siguen la misma regla: x! = x * (x - 1) *...* 2 * 1.

```
def factorial(x):
    if x == 0:
        return 1
    elif x > 0:
        y = 1
        for n in range(2, x + 1):
            y *= n
        return y
    else:
        print("ERROR: argumento recibido es negativo!")

# PRUEBAS
for k in range(5):
    print(k, factorial(k))
```

Ya que las pruebas muestran que la función actúa como esperabamos,
la incluimos dentro de la función "combinaciones":

```
def combinaciones(n, k):
    def factorial(x):
        if x == 0:
```

```
            return 1
        elif x > 0:
            y = 1
            for n in range(2, x + 1):
                y *= n
            return y
    # Regresar la cantidad de combinaciones
    return factorial(n) / (factorial(k) * factorial(n - k))

# PRUEBAS
for n in range(5):
    for k in range(n + 1):
        print(n, k, combinaciones(n, k))
```

8.7. Tareas

En todos programas de la tarea, las primeras tres líneas deben ser:

1. su nombre completo,
2. fecha para entregar tarea,
3. versión de texto, número de capitulo y número de la pregunta,

en el siguiente formato:

```
# Autor: Nombre Completo de Autor
# Fecha: 2016/02/17
# Texto: V08    Capitulo: 03    Programa: 09
```

Después dejen una línea vacía y en la quinta línea empiecen el programa.

NN) Escriba un programa que pida una letra y reporta si es vocal (aeiou) o consonante.

NN) Escriba el programa que pide nombre y apellido paterno al usuario,
separados por un espacio y los escribe en la pantalla en orden invertido:
apellido paterno y nombre separados por un espacio.

NN) Escriba un programa que pude nombre de archivo con extensión al usuario e
imprime la extensión a la pantalla.
Por ejemplo, recibe “archivo.txt” e imprime “txt”.

NN) Escriba el programa que pida tres enteros distintos al usuario y
reporta cual es el mayor y cual es el segundo mayor.

NN) Escriba un programa que pide tres enteros y
reporta si hay al menos dos iguales entre ellos.

NN) Escriba el programa que pide un entero positivo hasta 10,000 al usuario y
reporta si es número primo o no.

NN) Escriba un programa que pide un entero positivo y reporta si es número perfecto, o no.
Los numero perfectos son la suma de sus divisores propios: 6 = 1+2+3, 28 = 1+2+4+7+14.
Otros perfectos son: 496, 8128, 33550336, 8589869056, 137438691328.

NN) Escriba un programa que pide un entero y reporta cuantos dígitos 0 hay en el.

NN) Escriba un programa que pida un dígito (0 a 9) y un entero positivo (1 a 100) y
calcula cuantas veces se repite el dígito recibido en todos números positivos hasta el recibico.números enteros 0 y este número.

NN) Escriba el programa que pide un entero positivo y
reporta en la pantalla la suma de sus dígitos.

NN) Escriba el programa que pide un texto y responde si es palíndromo (simétrico) o no,
siendo texto en minúsculas. Los ejemplos de palíndromos que tiene que identificar son:
ojo, ojo rojo, luz azul, la tomo como tal, la ruta natural, yo hago yoga hoy, no deseo yo ese don,
y otros.

INTERES SIMPLE,

INTERES COMPUESTO DESPEJANDO

Funciones

NN) Escriba una función "max_2" que reciba dos objetos comparables y
regresa el más grande entre ellos.

NN) Escriba una función "max_3" que reciba tres objetos comparables y
regresa el más grande entre ellos.

9. Algoritmos

Antes de empezar a estudiar los algoritmos, vamos a completar nuestro estudio de las funciones.

9.1. Temas Avanzados sobre Funciones

9.1.1. Empaquetar y Desempaquetar Argumentos

Funciones Generadoras

Funciones "lambda"

9.2. Algoritmos y Tiempos de Ejecución

algoritmos, invariantes aserciones

seudo-código

diagramas de flujo.

fibonacci con tabla de resultados previos

9.3. El Modulo "profile", el Sistema Operativo

9.4. Recomendaciones y Ejemplos

9.5. Tareas

Parte III.

Programación Orientada a Objetos

10. Código Orientado a Objetos

10.1. Clases

Un Nuevo Paradigma

Un nuevo paradigma se refiere a
una nueva manera de pensar, una nueva manera de diseñar, analizar y programar software.

Hasta ahora aprendimos programar bajo el paradigma conocido como
"programación estructurada", o "programación procedural".

Este paradigma está basado en crear una jerarquía de funciones que logran el objetivo del programa llamándose unas a otras, pasando argumentos, ejecutando tareas asignadas y regresando resultados.

En los cuerpos de las funciones teníamos tres maneras de organizar declaraciones:
bloques; secuencias de declaraciones ejecutadas en orden de su escritura
ramificaciones "if-elif-else"; que ejecutan bloques sólo bajo ciertas condiciones y
ciclos "while-else" y "for-else"; que repiten bloques bajo ciertas condiciones.

Organizando las ramificaciones y ciclos en forma secuencial, o anidadas unas dentro de otras
en una función, la función lograba ejecutar las tareas deseadas sobre los datos.
Las funciones se escriben a nivel de los módulos, o se escribe una función anidada dentro de otra.

Conociendo estos elementos, el paradigma de programación estructurada está enfocada
en las acciones, o sea en las tareas que cada función tiene que ejecutar y comunicar.
Diseño de software bajo este paradigma es diseño de funciones y sus jerarquías.

El concepto de una "clase" es la base de otro paradigma, otra manera de pensar.
El enfoque ya no está sobre las tareas a cumplir, sino sobre las clases de objetos,
los atributos estáticos y dinámicos de los objetos y las estructuras que se forman
a base de las relaciones entre estos objetos: injerencia, agregación, y otras.
Este nuevo paradigma se le llama "orientada a objetos" (OO).
La primera fase es "diseño y análisis orientados a objetos" (DAOO).
Después sigue "programación orientada a objetos" (POO) para cumplir con el proyecto.

Adoptar esta nueva manera de pensar se logra practicando y escribiendo programas.
Aquí solamente vamos a dar una analogía como la primera idea.

Supongamos que deseamos simular tráfico en una ciudad.
Al aplicar programación estructurada, pensaríamos en las acciones:
encender motor del vehículo, arrancar, dar señalamientos, frenar, acelerar, dar vueltas, y
otras actividades que deseamos simular.
Tales actividades o funciones manejarían los datos que son diferentes tipos de vehículos.

En paradigma orientado a objetos pensaríamos de otro modo.
Pensaríamos sobre vehículos como objetos a simular.

Se formaría una clase de objetos llamada "vehículo" y dentro de esta clase
se programarían aspectos estáticos: tamaño, peso, cantidad de pasajeros y otros,
tanto como aspectos dinámicos: frenar, acelerar, dar vuelta, señalar y otros.
Cada vehículo simulado seria un "objeto", o una "instancia" de la clase "vehículo".

En lugar de que los algoritmos manejen los vehículos,
como lo teníamos en la programación estructurada,
en la programación orientada a objetos, cada vehículo sabe como comportarse.
Entonces no es necesario manejar objetos con algoritmos complejos "desde afuera".
Desde afuera se señala a un vehículo "frena" o "acelera" y
cada objeto "vehículo" ya sabe como hacerlo.

Así que programación orientada a objetos está basada en pensar sobre las clases de objetos,
cuales atributos estáticos y dinámicos necesita una clase de objetos, que señales recibe y/o
manda a otros objetos y como se relaciona con otros objetos de la misma u otras clases.

Cuando decimos "objetos", pensamos en objetos de mundo real que deseamos simular,
pero también seres vivos, como una planta, insecto, animal, ser humano, son objetos.
Hasta un evento, como un concierto, o un evento deportivo pueden ser modelados
como clases de objetos, con sus atributos estáticos y dinámicos.

Los atributos estáticos se modelan con uno o varios tipos de datos, mientras
los atributos dinámicos se modelan con funciones definidas dentro de la clase.
Tales funciones, que son definidas dentro de una clase
y modelan algún atributo dinámico de todos los objetos de dicha clase,
se les lama "métodos".

Dentro de los métodos, como dentro de las funciones, usaremos las tres estructuras conocidas:
bloques, ramificaciones y ciclos. Todo aprendido hasta ahora lo seguiremos usando.
Solamente organizaremos código a base de clases e instancias derivadas de ellas,
en lugar de organizar el código a base de funciones y procedimientos.

Ya estamos aprendiendo muchos conceptos nuevos. Es mejor aprenderlos programando ejemplos.

Todos los tipos de datos, los que ya conocemos y los que vamos a conocer,
igual como declaraciones, funciones, inclusive clases y otros, en Python son "objetos".

Vamos a empezar el estudio de clases de objetos, con objetos más "puros", o sea abstractos,
los objetos matemáticos, geométricos. Después modelaremos objetos de mundo real también.

Las primeras clases

Vamos a empezar con tres temas a lo largo del libro:
Objetos geométricos y gráficas, procesamiento de texto, y simulación.

Para empezar, vamos a diseñar el objeto: "punto en el sistema cartesiano de dos coordenadas".
Suena complicado. Simplemente vamos a diseñar la clase que representa punto en dos dimensiones.

Para empezar, tenemos que decidir el nombre de la clase y
tenemos que pensar en lo que es común a todos los puntos en dos dimensiones,
para diseñar estas características comunes como atributos de los puntos.

Para nombrar una clase se utiliza el

ENCAPSULATION, DATA HIDING

A class is code that specifies the data attributes and methods for a particular type of object.

Classes: graph, text, stochastic games

__init__

self

Instance variables and class variables.

10.1.1. Inheritance

10.1.2. Agregación (Delagación)

10.2. Desarrollo Basado en Pruebas

10.3. Un Programa en Python

10.3.1. Python interpreter's Point of View

Compiling "Source Code" Statements into "Byte Code" Statements

The "Python Virtual Machine"

The Python byte code is executed on a computer by a program called Python Virtual Machine.
The Python Virtual Machine is a part of Python interpreter, it need not be installed separately.

Python is an interpreted language.
After the source code is written, there is no compiling and linking into machine code.
The source code is sent to Python and executed through above mentioned phases.

The effect is that Python programs run slower then compiled languages, but
faster then other interpreted languages.
There are many ways to speed up Python almost to the level of compiled programs.
For speeding up Python see the chapter.

The lack of making (compiling and linking) makes software development cycle in Python
much shorter then the development cycle in compiled languages.

Python software can be made such that users can modify some modules,
adjusting the software to their needs,
without the need to make and deploy the whole software.

10.4. El Modulo "datetime",

Aquellos que están usando IDLE en Windows, van a tener que cambiar a usar "cmd.exe",
el programa "command", conocido como el Microsoft "DOS-prompt".
Para programadores en Windows, los módulos de Python que usan "tkinter",
deben de tener la extensión ".pyw" en lugar de ".py".
En otros sistemas operativos se puede usar cualquiera de las dos extensiones.

10.5. Recomendaciones y Ejemplos

10.6. Tareas

11. Iterables e Iteradores

12. Comprensiones y Utilidades

12.1. List comprehensions, dictionary comprehensions

12.2. Mapping, Filtering, Reducing

13. Procesamiento de Texto

14. Archivos Binarios e Imágenes

pickling, serialization

15. Estructuras de Datos y Algoritmos

15.1. stacks

15.2. queues

15.3. trees

15.4. graphs

Parte IV.

Programación Guiada Por Eventos

16. Patrones de Desarrollo

17. Algunas Plantillas

Parte V.

Considerar

18. Complejidad de Algoritmos

19. Ingeniería de Software

20. Procesamiento de Imágenes

21. Estudio de Casos

Parte VI.

Referencias

22. Tablas de Referencia

22.1. Palabras Reservadas ("Keywords")

Para obtener la lista de palabras reservadas en modo interactivo usa:

```
>>> import keyword
>>> keyword.kwlist
```

Al momento de escribir este texto,
la lista de las palabras reservadas contiene las siguientes 33 palabras:

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

22.2. Operadores y Precedencias

22.2.1. Operadores

La siguiente es la tabla de operadores de lenguaje Python.

```
-------------------------------------------------------------------------
=     asignación        +x    plus x            and   "y" lógico
<     menor             -x    minus x           or    "o" lógico
>     mayor             +     suma              not   "no" lógico
==    igual             -     resta             &     "y" de bits
!=    no es igual       *     multiplicación    |     "o" de bits
<=    menor o igual     /     división          ^     "o exclusivo"
>=    mayor o igual     //    cociente          ~x    "no" de bits
is,   es                %     residuo           <<    desplazamiento *
in,   está en           **    potencia (raíz)   >>    desplazamiento /

    [ ]  lista, o índice            [:], [::] sub-sting, o sub-lista
```

```
    { }  conjunto, o diccionario    {:}  diccionario
    ,    tupla                      .   objeto.atributo
    ,    separador de expresiones   ;   separador de declaraciones

    ( )  función: lista de parámetros, o llamada con argumentos
-------------------------------------------------------------------------
+ la suma de dos cadenas produce su concatenación
* una cadena multiplicada por entero N es concatenada N veces.

Variantes del operador de asignación:
+=, -=, *=, /=, //=,%=, **=, &=, |=, ^=, <<=, >>=

Los (), [] y {} se usan para agrupar operandos en las expresiones.

% ya no se usa para formateo de salida, aunque todavía funciona.
-------------------------------------------------------------------------
```

22.2.2. Precedencia

En la declaración de asignación

```
nombre = expresión
```

la expresión es evaluada primero, el objeto resultante guardado en la memoria y después el lugar inicial de la memoria RAM que ocupa el objeto es relacionado con el nombre, de manera que el objeto puede ser leído de la memoria a través del nombre.

Esto implica que primero se ejecuta el lado a la derecha de = y después el lado a la izquierda.

Con esta excepción, todos los demás operadores son ejecutados de izquierda a la derecha, tomando en cuento modificación causada por las reglas de precedencia.

```
lambda
if ... else
or
and
not
is, is not, in, not in, <, >, !=, <=, >=
==
|
^
&
<<, >>
+, -
*, /, //,%
+x, -x, ~x
**
x[index], x[index:index]
x(arguments...), x.attribute
(expressions...)
```

```
[expressions...]
{key:value, ...}
{expressions...}
```

Existe un caso especial de no respetar la precedencia usual:

```
b ** -E     se interpreta como     b ** (-E)     o sea     1 / (b ** E)
```

aunque es mejor acostumbrarse a escribir: b ** (-E), en lugar de: b ** -E.

22.3. Funciones Incorporadas

Funciones incorporadas son parte de el intérprete y no necesitan ser importadas de ningún modulo. Estas funciones tiene alcance de programa, pueden ser usadas en cualquier parte de código.

Todas estas funciones se estudian en este libro. Para obtener más información vea la página: https://docs.python.org/3/library/functions.html

```
abs()           dict()          help()          min()           setattr()
all()           dir()           hex()           next()          slice()
any()           divmod()        id()            object()        sorted()
ascii()         enumerate()     input()         oct()           staticmethod()
bin()           eval()          int()           open()          str()
bool()          exec()          isinstance()    ord()           sum()
bytearray()     filter()        issubclass()    pow()           super()
bytes()         float()         iter()          print()         tuple()
callable()      format()        len()           property()      type()
chr()           frozenset()     list()          range()         vars()
classmethod()   getattr()       locals()        repr()          zip()
compile()       globals()       map()           reversed()      __import__()
complex()       hasattr()       max()           round()
delattr()       hash()          memoryview()    set()
```

22.4. Tipos de Objetos y Comparaciones

- Números son comparados por su valor, después de ser convertidos a tipo común.
- Las cadenas son comparados por números ordinales de sus caracteres.
- Comparación de estructuras anidadas es recursiva.
- Las colecciones se comparan como si fueron convertidos a cadenas.
    - Listas y tuplas son comparadas de izquierda a la derecha, recursivamente.
    - Diccionarios se comparan, comparando sus elementos {llave: valor}.
    - Dos conjuntos son iguales si son subconjuntos uno a otro.

22.5. La Jerarquía de Excepciones

https://docs.python.org/3/library/exceptions.html

22.5.1. Las Excepciones Más Comunes

`ZeroDivisionError`	Intento de dividir entre cero.
`IOError`	No es posible abrir el archivo o dispositivo.
`NameError`	El nombre (identificador) no es conocido.
`ValueError`	No se puede convertir el objeto a tipo deseado, o el ítem buscado no está en la colección.
`TypeError`	El tipo de operando no aceptable para el operador.
`IndexError`	El índice está fuera de la secuencia.
`KeyError`	La llave buscada no existe en el diccionario.
`AttributeError`	El objeto no tiene el atributo buscado.
`NotImplementedError`	El código todavía no está implementado.
`AssertionError`	La declaración de aserción es evaluada a False.

22.5.2. Jerarquía Completa de Excepciones

https://docs.python.org/2/library/exceptions.html#exception-hierarchy

Los Errores: "...Error"

```
BaseException
    +-- SystemExit
    +-- KeyboardInterrupt
    +-- GeneratorExit
    +-- Exception
        +-- StopIteration
        +-- ArithmeticError
            +-- FloatingPointError
            +-- OverflowError
            +-- ZeroDivisionError
        +-- AssertionError
        +-- AttributeError
        +-- BufferError
        +-- EnvironmentError
            +-- IOError
            +-- OSError
            +-- WindowsError (Windows)
            +-- VMSError (VMS)
        +-- EOFError
        +-- ImportError
        +-- LookupError
            +-- IndexError
            +-- KeyError
        +-- MemoryError
        +-- NameError
            +-- UnboundLocalError
        +-- ReferenceError
        +-- RuntimeError
            +-- NotImplementedError
        +-- SyntaxError
            +-- IndentationError
                +-- TabError
        +-- SystemError
        +-- TypeError
        +-- ValueError
            +-- UnicodeError
                +-- UnicodeDecodeError
                +-- UnicodeEncodeError
                +-- UnicodeTranslateError
```

Las Advertencias: "...Warning"

```
BaseException
    +-- Exception
        +-- Warning
            +-- DeprecationWarning
            +-- PendingDeprecationWarning
            +-- RuntimeWarning
            +-- SyntaxWarning
            +-- UserWarning
            +-- FutureWarning
            +-- ImportWarning
            +-- UnicodeWarning
            +-- BytesWarning
            +-- ResourceWarning
```

23. Instalar y Ejecutar Python

Para Incluir:

Why are you using cProfile instead of timeit?

The former is for finding bottlenecks in large programs, and sacrifices some accuracy on the small scale for it. The latter is for measuring the overall performance of tiny snippets relatively precisely. timeit should be the first choice for microbenchmarks

---

#!usr/bin/env python3

La primera linea es llamada “shebang” y permite convertir este archivo de texto
a un archivo ejecutable en los sistemas tipo “Unix”, como el “Mac” y el “Linux”.
El archivo se convierte al ejecutable usando.

chmod +x <module_name.py>

En “Windows” esto no es necesario ya que el “Windows” asocia archivos a los programas
a base de la extensión “.py”. Por todos modos dejamos el “shebang” para el caso
que alguien quiere reusar este modulo y trabaja sobre un sistema tipo “Unix”.

Para ver esto, al ejecutar el script:

```
# Nombres globales:
CONST = 1
var = 2

def función(x):
    # Los nombres globales son visibles para la lectura.
    print('reporte 1:  ', CONST, var)

    # Los nombres globales no son fácilmente accesibles
    # para la reasignación dentro de la función.
    var = x

    print('reporte 2:  ', CONST, var)

función(5)
print('reporte 3:  ', CONST, var)
```

se obtendrá el “Traceback”:

```
Traceback (most recent call last):
  File "local_escondiendo_global_2.py", line 17, in <module>
```

```
    función(5)
  File "local_escondiendo_global_2.py", line 8, in función
    print('reporte 1: ', CONST, var)
UnboundLocalError: local variable 'var' referenced before assignment
```

La última linea del "Traceback" dice que
intentamos a leer (referenciar) la variable local "var",
antes de que le fuera asignado un objeto.
La penúltima linea dice en la cual línea de código estamos leyendo esta variable.

Lo que dice "Traceback" es cierto.
La asignación "var = x" esta creando una variable local en el cuerpo de función, y
la declaración "print('reporte 1: ', CONST, var)" la trata de leer antes de esta asignación.

La variable global "var" ni se menciona, ya que no es visible dentro de la función.
Está escondida por la variable local con el mismo nombre: "var".

Formato de tarea

En todos programas de la tarea, las primeras tres líneas deben ser:

1. su nombre completo,
2. fecha para entregar tarea,
3. versión de texto, número de capitulo y número de la pregunta,

en el siguiente formato:

```
# Autor: Nombre Completo de Autor
# Fecha: 2016/02/03
# Texto: V04    Capitulo: 01    Programa: 06
```

Después dejen una línea vacía y en la quinta línea empiecen el programa.

El nombre de archivo que empiece con sus iniciales y _:

```
ikb_101.py
ikb_precio.py
ikb_107_distancia.py
```

Por lo pronto aclararemos la diferencia entre:
cadena para el usuario: s ("string" ordinario para usuarios y programadores)
cadena para intérprete: r ("representation string" en forma de código para el intérprete mismo)

El interprete de Python formatea datos a texto, de dos maneras:
texto para ser entendido y posiblemente ejecutado por el interprete de Python y
texto para ser leído por parte de usuarios del programa y los programadores.

La diferencia se puede observar en el siguiente ejemplo.

```
>>> "hola"
'hola'
```

```
>>> print(_)
hola
>>>
```

Al ejecutar la expresión, "hola" al intérprete, este lo evalúa, lo identifica como una cadena y escribe resultado en formato "r" (representation string), o sea, para sí mismo,
por si necesita procesar esta cadena en las siguientes expresiones.
Esto se refleja en el hecho que la respuesta está escrita entre comillas: 'hola'.

Al ejecutar la declaración "print(_)", el simbolo _ se refiere al resultado previo: 'hola'.
Esto significa que la declaración es equivalente a la declaración "print('hola')".
La función "print" escribe para "lectores humanos": usuarios y programadores, así que,
su resultado está escrito en formato "s" (string), lo que se refleja en la ausencia de comillas: hola.

Todos tipos de datos en Python tienen dos formatos:
el ordinario para la lectura y el de representación en código.
Para algunos tipos de datos, estas representaciones son idénticas, por ejemplo enteros.

Programador puede convertir cualquier tipo de dato a un "string", o sea un "representation string", usando las funciones

```
str(dato)                          repr(dato)
```

Formateo de los Datos Incorporados

El formato es una cadena que siempre termina con uno de las siguientes símbolos:

- s, para cadenas ("strings")
- d, b, o, x, X para enteros
- e, E, f, F, g, G, % para punto flotante

El significado de los símbolos para formatear es el siguiente:

```
# enteros:                         # flotantes:
d   # decimal                      e, E   # notación científica
b   # binario                      f, F   # notación decimal
o   # octal                        g, G   # el más corto entre e or f
x   # hexadecimal                  %      # notación de porcentaje
# cadenas ("strings"):
s
```

El uso de estos símbolos se entenderá en los ejemplos concretos que siguen.

Nombres representando cantidades con unidades de medida,
siempre deben de incluir las unidades de medida.

```
SI:                                NO:
peso_N, peso_lb                    peso
masa_kg, masa_g                    masa
rapidez_m_por_s                    rapidez
aceleración_m_por_s2               aceleración
```

En constantes de conversión hay que especificar unidades con palabras completas.

```
SI:                                NO:
KILOGRAMOS_EN_LIBRA = 0.45359237   KG_EN_LB = 0.45359237
NEWTONS_EN_LIBRA = 4.44822162      N_EN_LB = 4.44822162
```

La Función "del"

Ejecutable en sistemas tipo "Unix"

Los sistemas tipo "Unix" son Linux, Mac, y algunos otros.
Para evitar teclear "python3" cada vez al ejecutar un programa y
convertir un modulo Python en programa ejecutable en sistemas tipo "Unix",
hay que hacer dos cosas.

Primero, dentro del modulo insertamos como la primera linea la que sigue:

```
#!/usr/bin/env python3
```

Si el programa está en un servidor web, la primera linea debería ser:

```
#!/usr/bin/python
```

El segundo es que en el terminal agregamos modo "ejecutable" al archivo:

```
chmod +x nombre_de_modulo.py
```

después de esto es suficiente teclear el nombre del modulo, cualquiera que este sea,
para ejecutar el modulo.

```
nombre_de_modulo.py
```

Ejecutable en Sistemas "Windows"

Desde el "DOS-prompt" en Windows, podemos simplemente ejecutar

```
nombre_de_modulo.py
```

y el programa va a funcionar ya que Windows asocia archivo por su extensión,
a la aplicación que la debe de ejecutar.

La única inconveniencia podría ser que el Python de sistema sea Python 2 y no Python 3
para el cual estamos escribiendo código.

En la sección 2.6.1 en la página 33 aprenderemos a verificar
con cuál versión de intérprete estamos trabajando.

La primera declaración en el cuerpo de "main" es la asignación

```
nombre = pide_nombre()
```

dentro de la cual es la expresión que llama a ejecutar la función "pide_nombre".
Ahora se empieza a ejecutar la función "pide_nombre".
Esta función a su vez llama la función "input" y pide el nombre al usuario.

```
return input("Por favor, ingrese su nombre: ")
```

Digamos que el usuario proporciona el nombre Juan.

Entonces, la función "input" regresa el objeto "cadena Juan".
La declaración "return" en la misma linea con el "input", regresa esta cadena "Juan"
a la función "main", donde "pide_nombre()" es sustituido por "Juan":

```
nombre = Juan
```

El objeto "cadena Juan" es guardado en la memoria y referenciado por la variable "nombre".

En seguida se ejecuta la segunda declaración del cuerpo de "main":

```
saluda(nombre)
```

El intérprete identifica la función "saluda", e identifica el operador de ejecución ().
Se sustituye la variable "nombre" con el valor "Juan" y se llama la función "saluda":

```
saluda("Juan")
```

La función "saluda" se ejecuta como si fuera la función:

```
def saluda("Juan"):
    print("Hola, ", "Juan", ".", sep="")
```

En el cuerpo de la función "saluda" se ejecuta y la función "print" la cual imprime

```
Hola, Juan.
```

Dado que la función "saluda" no tiene declaración "return",
se regresa el valor "None" a la función "main" en lugar de "saluda(nombre)".

Como el resultado de la función "saluda" en main no esta asignado a ninguna variable,
el "None" que se regresa se descarta. No se guarda en ningún lado.

La función "main" no tiene más declaraciones y así termina este programa.

Ya no vamos a analizar funciones así detalladamente, se espera que el proceso quedó entendido.

POR HACER:

En Py3.5 los conjuntos: los “in”, “remove” and “discard” pueden recibir como argumento conjunto.

En las cadenas: lugar = índice + 1

“enumerate” en programación estructurada, trabajando con secuencias.

mencionar “type” a principio

mencionar “del” al principio

mencionar “escalar” y “colección” a principio.

Agregar temas:

0) 1 lenguaje, sintaxis, semántica

1) 1 computadora: memoria, ALU, unidad de control

2) 1 Números: conjuntos de números

3) 2 Floating point numbers

4) 2 Caracteres como tipos de datos: ASCII, UTF-8

5) 2 Conjuntos

6) 2 Lógica

7) Algoritmos limitaciones NP Complete

8) Sumas, secuencias, series

Conteo básico, Permutaciones (circulares), combinaciones

A byte of Python

p28, instrucciones para ejecutar script en Win, Mac Linux a 1.2.2

Programar juegos como ejemplos.

CICLOS

programar conversiones de numeros a binario

augmented assignment operators!

POR HACER:

user controlling loop: input validation.

procesar datos: if dentro del ciclo.

acumular resultado

acumular resultado y todal, promedio, varianza como ejemplo.

ejemplo break (do...while). recibe linea a procesar de usuario y si es vacía sale.

cuenta cuantas veces se repite una letra en la cadena.

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print(count)
```

for time in range(ini, fin, step) parar cuando antes de la altura negativa.

FUNCIONES

Función en al cual "return" aparece varias veces con if y con else.

Nombres de funciones que sean verbos, funciones lógicas "es_par", "es_divisible".

Estructuras de datos, recursión ??? suma recursiva, fibonacci.
