Introducción

¿Qué es Python?

Python es un lenguaje de programación interpretado de alto nivel, diseñado por Guido van Rossum a inicios de los 90's. Cuenta con una segunda versión (Python 2) la cual fue liberada en el año 2000. Su versión más actual, Python 3, fue liberada en 2008. Las diferencias entre las versiones podemos notarlas en la sintáxis del lenguaje.

Características

Este lenguaje tiene varias características que lo diferencían de muchos otros lenguajes. Entre estas características se encuentran las siguientes:

- *Es interpretado*: Ejecuta instrucciones sin una previa compilación. Se hace mediante un intérprete.
- *Usa tipado dinámico*: Las variables que se utilizan pueden tomar valores de diferente tipo en cualquier momento, ya que el tipo se determina en tiempo de ejecución. No es necesario declarar el tipo de dato a utilizar como lo hace el lenguaje C.
- *Es fuertemente tipado*: La variable que tenga un valor de un tipo concreto no puede usarse como si fuera de un tipo distinto (ejemplo: cadenas con números) a menos que se haga una conversión.
- *Es multiparadigma*: Permite adoptar varios paradigmas de programación como: programación estructurada, programación orientada a objetos (POO) y programación funcional.

Ventajas

- *Fácil de usar*: Contienen todas las expresiones que se usan en otros lenguajes pero simplificadas.
- *Expresividad*: Una línea de código en Python, puede hacer más que una línea de código en cualquier otro lenguaje. Esto implica mayor facilidad para mantener y depurar programas.
- *Legibilidad*: Python utiliza una sintaxis sencilla y elegante. Facilita la lectura de los programas.
- "*Baterías incluidas*": Al momento de instalar Python se tiene todo lo necesario para poder hacer un trabajo real. Su librería estándar, incluye módulos para manejo de email, páginas web, entre otros.
- *Multiplataforma*: El mismo código puede ejecutarse tanto en Windows, como en sistemas UNIX.
- *Open Source*: Se puede descargar e instalar cualquier versión y puede usarse para desarrollar software comercial sin necesidad de pagar.

Desventajas

- *No es el lenguaje más rápido*: Al ser un lenguaje interpretado, los programas pueden llegar a ejecutarse más lento que un programa compilado en algún otro lenguaje.
- *No posee las librerías más extensas*: A pesar de que contiene una gran cantidad de librerías desde el momento de su instalación, otros lenguajes (Java o C) contienen librerías más extensas disponibles. Sin embargo, Python puede extenderse con librerías incluso de otros lenguajes.
- *No tiene revisión de tipos*: Las variables declaradas son referencias a objetos, por lo que no es están ligados a un tipo en particular.

El Zen de Python

Un código que sigue los principios de Python de legibilidad y transparencia, se dice que es *pythonico*. De manera contraria, a un código opaco se le llama *no pythonico*. Estos principios fueron descritos por el desarrollador de Python Tim Peters en el llamado "*Zen de Python*", el cual dice lo siguiente:

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Disperso es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Aunque lo práctico gana a la pureza.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

¿En qué se utiliza? ¿Quién lo usa?

- Google hace un amplio uso de Python en su sistema de búsqueda web
- El servicio de video YouTube, es en gran medida escrito en Pyton.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, e IBM utilizan Python para las pruebas de hardware.
- La NASA ha utilizado Python para tareas de programación científica.

Instalación

La instalación de Python es bastante sencilla, sin embargo, esta puede cambiar dependiendo del sistema operativo que estemos utilizando, ya que puede ser directamente desde la página www.python.org/downloads o bien desde algún gestor de paquetes.

Instalación en Windows

Debemos entrar a la página previamente mencionada y podremos descargar el instalador de la versión más actual (Python 3.6). Con el instalador abierto, debemos buscar la opción "**Add Python 3.x to PATH**". La opción anterior nos permitirá manejar Python desde nuestra consola (cmd) en cualquier ubicación en la estructura de directorios. Posteriormente, en el instalador, bastará con darle clic en "Install Now".

Para verificar que nuestra instalación ha sido exitosa, debemos abrir una consola de Windows (cmd) y teclear lo siguiente:

```
python --version
```

Como respuesta obtendremos un mensaje que nos indica la versión instalada en nuestra computadora.

Instalación en sistemas UNIX/Linux

Es muy probable que en este sistema operativo, ya se tenga una versión instalda de Python, esto podemos comprobarlo abriendo una consola y escribir el comando:

```
python --version
```

Es posible que la versión instalada sea la versión 2, por lo que si se quiere instalar la versión 3, debemos hacer uso del gestor de paquetes ejecutando el siguiente comando (según sea el caso):

- Para Ubuntu: `sudo apt-get install python3.x.x`
- Para Fedora: `sudo yum install python3.x.x`

Sistemas OS X

Es posible que ya se tenga una versión instalada en nuestra computadora. Para verificarlo debemos ejecutar el siguiente comando en una consola:

```
python --version
```

En este sistema operativo podemos instalar Python de la misma manera que en Windows, descargando el instalador desde la página, sin embargo también podemos hacerlo utilizando un gestor de paquetes como *brew*, haciendo uso del comando:

```
brew install python3
```

El intérprete de comandos

Al abrir el intérprete de Python, entraremos al modo interactivo, el cual nos permitirá ejecutar diversos comandos de Python. Para iniciarlo, en una terminal, debemos escribir `python` en la línea de comandos. Si existieran varias versiones instaladas, se le puede especificar la deseada, por ejemplo, escribiendo: `python3` . Para salir del modo interactivo, se debe usar la combinación de teclas CTRL+Z

El prompt de comandos lo identificaremos por >>>. Este prompt indica que se puede escribir un comando para ser ejecutado, por ejemplo:

```
>>> print("Hola mundo")
```

Existen algunas herramienta útiles para la exploración de Python. Una de ellas es la función `help()` , la cual tiene dos modos. Si escribimos solo `help()` entraremos al sistema de ayuda. En ese sistema el prompt cambia a `help>` dentro del cual podemos ingresar el nombre de algún módulo como, por ejemplo, *math* y explorar la documentación.

Si se le ingresa un parámetro a la función `help()` se obtiene documentación inmediata del argumento.

```
>>> x = 10

>>> help(x)
```

Otra función útil es `dir()` , que nos ayudará a listar los objetos en un espacio particular. Si la utilizamos sin parámetros, listrá las variables globales, pero también puede listar los objetos de un módulo o incluso de un tipo:

```
>>> dir()

>>> dir(int)
```

Tenemos una función más, la cual es `type()` . Esta devolverá el tipo de dato del objeto que recibe como parámetro:

```
>>> type(5)
```

Propiedades del lenguaje y estándares

Con la finalidad de que nuestro código sea más legible y no se obtengan errores al momento de la ejecución, se han establecido diversas convenciones para la escritura de código, las cuales pueden encontrarse en la siguiente liga:

- Guía de estilo para código Python

El documento completo se denomica PEP 8 (Python Enhancement Proposal)

Tipos de datos

Mutabilidad

En el lenguaje Python, se hablará mucho del concepto de mutabilidad e inmutabilidad. Estos conceptos se refieren a la capacidad de los objetos de cambiar su valor una vez asignados a un identificador. Se dice que una clase es **`inmutable`** si cada objeto de esa clase tiene un valor fijo después de su instanciación, es decir, no podrá ser cambiado. En caso contrario, el concepto de **`mutabilidad`** nos indica que podremos cambiar el valor del objeto después de ser instanciado, o bien, asignado a una variable.

Numéricos y sus operadores - Inmutable

Constante None

None es una constante especial que se encuentra en el lenguaje Python, cuyo valor es nulo. Sin embargo, debido a muchas condiciones debemos mencionar:

- None no es lo mismo que False.
- None no es 0.
- None no es una cadena vacía.
- Si se compara a None con otra cosa que no tenga un valor None, siempre se obtendrá False.
- None es el único valor nulo.

Este tiene su propio tipo de dato (NoneType). A las variables podemos asignarles None y al ser un valor nulo, todas las variables que contengan a None, son iguales entre sí.

Numéricos

Dentro del lenguaje Python, encontraremos que podemos utilizar tres tipos de datos numéricos:

- Enteros
- Flotantes
- Complejos

Con ellos podemos realizar operaciones con los operadores aritméticos, justo como lo haríamos en clase de matemáticas. Se cuenta con el operador para suma (+), para resta (-), multiplicación (*), división (/). Además contamos con operadores como módulo (%) que nos permite obtener el residuo de una división y también la exponenciación (**) para poder elevar un número a una potencia.

Ejemplo de uso de los operadores:

```
>>> x = 2+5*6/2
17.0
>>> y = 5%2
1
>>> x = 2**3
8
```

Podemos tener operaciones también con los números complejos:

```
>>> a = 2+5j
>>> b = 6+2j
>>> a+b
(8 + 7j)
>>> a-b
(-4 + 3j)
>>> a/b
(0.549999999 + 0-6499999999j)
```

Para el caso del tipo de dato flotante, habrá ocasiones en donde nosotros queramos disminuir el número de decimales. Es decir, imaginemos que tenemos la siguiente operación:

```
>>> 10/3
3.33333333333333
```

¿Qué pasa si yo únicamente quiero dos decimales? La solución es sencilla, sólo basta con que al momento de querer imprimir el valor, sigamos el siguiente formato:

```
>>> x= 10/3
>>> print("%.2f"%x)
3.33
```

En el caso anterior imprimimos una cadena con formato, en donde con el % indicamos que vamos a imprimir un tipo de dato, el .2 hace referencia al número de decimales que queremos que imprima (en este caso, son 2) y la letra *f* es para indicar que es un tipo de dato flotante. Este tipo de notación con el operador % se estudiará más a fondo en el apartado de cadenas.

A pesar de que esto da un buen resultado, habrá ocasiones en que el intérprete nos redondee el resultado. Si queremos trabajar sin que se modifique nuestro valor, podemos utilizar un módulo llamado `decimal`, sin embargo, el estudio de ese módulo quedará para después (cuando se revise el tema de módulos y paquetes).

Booleanos

Cuando nos referimos a un tipo de dato booleano, hablamos de un valor que sólo tiene dos valores posibles: *verdadero* o *falso*. En el lenguaje Python, podemos encontrar estos dos valores con las palabras `True` (Verdadero) y `False` (Falso). Este tipo de dato se utiliza mucho en los sentencias condicionales, las cuales se estudiarán más adelante en el curso, y así como podemos realizar operaciones con números enteros, también podemos realizar operaciones con valores booleanos. Este tipo de operadores se estudiarán en el apartado de Control de flujo.

Podemos asignar un valor booleano a una variable de la siguiente manera:

```
>>> x = True
>>> print(x)
True
>>> y = False
>>> print(y)
False
```

Función `bool()` para saber qué es True y qué es False

```
>>> bool(0)
False
```

```
>>> bool(25)
True
>>> bool([])
False
>>> bool([1,2])
True
```

Esto es porque Python toma como `True` cualquier cosa que tenga un valor, ya sea un número, cadena o tupla. Por otro lado toma como `False` cuando algún dato no tenga valor, como por ejemplo el número 0, una cadena vacía [], una tupla vacía, etc.

Operadores lógicos

Cuando manejamos booleanos, podemos utilizar operadores que nos ayudarán a evaluar expresiones y determinar si algo es falso o verdadero. Los operadores son `and`, `or` y `not`.

- and: 'y' lógico. Este operador da como resultado True si y sólo si sus dos operandos son True.

```
>>> print("True and True: ", True and True)
True and True:  True
>>> print("True and False: ", True and False)
True and False:  False```
>>> print("False and True: ", False and True)
False and True:  False```
>>> print("False and False: ", False and False)
False and False:  False
```

- or: 'o' lógico. Este operador da como resultado True si algún operando es True.

```
>>> print("True or True: ", True or True)
True or True:  True
>>> print("True or False: ", True or False)
True or False:  True
>>> print("False or True: ", False or True)
False or True:  True
>>> print("False or False: ", False or False)
False or False:  False
```

- not: negación. Este operador da como resultado True si y sólo si su argumento es False.

```
>>> print("not True: ", not True)
not True:  False
>>> print("not False: ", not False)
not False:  True
```

Expresiones compuestas (Jerarquía de operadores)

Python evalúa primero los not, luego los and y por último los or

Si no se está acostumbrado a evaluar expresiones lógicas compuestas, se recomienda utilizar paréntesis para asegurar el orden de las operaciones.

- El operador not se evalua antes que el operador and

```
>>> print(not True and False)
False
```

- Manejo de paréntesis para asegurar jerarquía

```
>>> print((not True) and False)
False
```

- Primero se evalua el paréntesis

```
>>> print(not (True and False))
True
```

- El operador not se evalúa antes que el operador or

```
>>> print(not False or True)
True
```

- El operador and se evalúa antes que el operador or

```
>>> print(False and True or True)
True
```

Cadenas - Inmutable

> Nota: Para delimitar una cadena siempre será entre comillas dobles `"cadena"` o con comillas simples `'cadena'` . No hay diferencia entre si usas uno u otro.

A una variable también podemos asignarle una cadena de texto, muchas veces conocidas como *strings*. Para poder esta asignación existen varias formas, ya que podemos hacerlo tanto con comillas dobles, como con comillas simples.

Por ejemplo:

```
cadena = "Esta es una cadena con comillas dobles"
#Las comillas dobles pueden contener comillas simples y viceversa
cadena = "Hola 'Contengo comillas simples'"
cadena = 'Hola "Contengo comillas dobles"'
```

Si deseamos darle un poco de formato a una cadena de texto, podemos darle un espacio al inicio de ella, es decir, inicia con un tabulador. Además, es posible que queramos dar un salto de línea en una parte específico de la cadena. Esto se logra con \t y \n .

Pongamos el ejemplo:

```
>>> x = "\tIniciaré con un tabulador"
>>> print(x)
        Iniciaré con un tabulador
>>> x = "Hola contengo un \nSalto de línea"
>>> print(x)
Hola contengo un
Salto de línea
```

Cuando queramos evitar saltos de línea poniendo `\n` y tabuladores con `\t` , podemos declarar una cadena usando triple doble comilla (""") o triple comilla simple (''').

Por ejemplo:

```
>>> cadena = """Cadena que acepta
... multiples lineas
```

```
... y   tabuladores"""
>>> print(cadena)
Cadena que acepta
multiples lineas
y       tabuladores
```

Concatenar cadenas

Al igual que con los tipos de datos numéricos, podemos utilizar operadores como `+` y `*`.

Para el caso de las cadenas de texto, el operador `+` nos servirá para poder concatenar cadenas, o dicho de una manera más sencilla, servirá para juntarlas.

```
>>> a = "Hola"
>>> b = " mundo"
>>> print(a+b)
Hola mundo
```

Para el caso del operador `*`, podremos imprimir una cadena cualquiera la cantidad de veces que le indiquemos después del operador `*`.

Por ejemplo:

```
>>> cadena = "Hola"
>>> print(cadena*2)
HolaHola
```

Conversión de cadenas a números

Debido a la gran necesidad de trabajar con números y cadenas, existen métodos para la conversión de tipos, en este caso, de cadenas a números. Si nosotros tenemos una cadena que representa un número como por ejemplo:

```
>>> x = "123"
```

Esta variable no podríamos utilizarla para operaciones numéricas, ya que es una cadena. Para poder convertirla a un número, contamos con los siguientes métodos.

- int() -> Convierte una cadena a un entero
- float() -> Convierte la cadena a un flotante

```
>>> cadena = "123"
>>> numeroEntero = int(cadena)
>>> numeroFlotante = float(cadena)
>>> print(numeroEntero,numeroFlotante)
123 123.0
```

Conversión de números a cadenas

A veces pasa que queremos usar números como cadenas, para `castear` de un numero a una cadena debe ser con la función str(), la cual recibe como parámetro un entero o cualquier otro objeto.

```
>>> numero = 4
>>> cadena = str(numero)
>>> print(cadena)
4
```

> Nota: Cuidado cuando se trate de convertir una cadena que sea un flotante a un entero, marcará error.

```
>>> cadena = "123.45"
>>> numero = int(cadena)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.45'
```

> Nota: El término `castear` o `casteo` no es más que una simple conversión de un tipo de dato a otro. Más adelante veremos como convertir una cadena o un número a otros tipos de datos como lo son `tuplas` , `listas` , etc.

Indexación de cadenas (manejarla mediante índices)

En python una cadena es una secuencia de caracteres, por lo tanto podemos acceder a sus valores mediante índices.

> Nota: Los índices siempre empezarán desde el 0 y terminarán en `número de elementos − 1` , es decir, si nuestra cadena tiene 7 caracteres, sus índices serán desde el 0 hasta el 6 (7-1).

Por ejemplo:

```
>>> x = "Python AM"
>>> print(x[0])
P
```

En el ejemplo anterior tenemos una cadena llamada "x", cuando accedemos a su índice 0 estamos diciendo que queremos el primer elemento de esa cadena, en este caso la letra "P". En caso de que no queramos un solo carácter sino varios, podemos utilizar un rango en los índices.

Por ejemplo:

```
>>> print(x[0:5])
Pytho
>>> print(x[:5])
Pytho
>>> print(x[1:5])
ytho
>>> print(x[1:])
ython AM
>>> print(x[:])
Python AM
```

> cadena[indice_inicial : indice_final - 1]

En el ejemplo anterior estamos diciendo que queremos del índice 0 al índice 5, pero cuando ponemos un índice final nunca llegará a ese valor, es decir, la cadena solo imprimirá del índice 0 al índice 4.

En el segundo caso, cuando no ponemos el valor inicial, automáticamente lo tomará como 0.

En el tercer caso, estamos poniendo el índice 1 como valor inicial y el índice 5 como valor final, por lo tanto nos va a imprimir desde el índice 1 hasta el índice 4.

En el cuarto caso, si no ponemos un índice final, automáticamente lo tomará como si fuera toda la cadena, es decir, vamos a imprimir desde el índice 1 hasta el final de la cadena.

En el último caso, cuando no ponemos ni índice inicial ni índice final, python tomará esto como si fuera desde el índice 0 hasta el final de la cadena.

Métodos asociados a las cadenas

Para el uso de cadenas contamos con diversos tipos de métodos, como los son:

- split() -> Nos servirá para poder dividir una cadena en subcadenas

> Nota: split() devuelve una lista con las subcadenas

```
>>> x = "Hola Mundo"
>>> y = x.split()
>>> print(y)
['Hola', 'Mundo']
```

Si el método `split()` no tiene ningún argumento, es decir, no tiene nada dentro de los paréntesis, el método separará la cadena cada espacio, como en el ejemplo anterior.

Cuando no queramos que los separe por espacios y queramos que separe la cadena por otra cosa, debemos pasarle una cadena como argumento al método.

Por ejemplo:

```
>>> x = "24/05/1996"
>>> y = x.split('/')
>>> print(y)
['24', '05', '1996']
```

En el ejemplo anterior, separamos nuestra cadena con `'/'` .

> Nota: La cadena separadora nunca se guardará

- join() -> Permite concatenar cadenas de una mejor manera que con el operador '+'

> Nota: join() devuelve una cadena concatenada por otra cadena. Recibe como argumento una lista o tupla

```
>>> x = '-'.join(['Junta', 'las', 'cadenas'])
>>> print(x)
Junta-las-cadenas
```

En el ejemplo anterior tenemos una lista con 3 cadenas y un separador que es el `-` , el método `join()` concatena los elementos que se encuentren en la lista (o tupla) separada por una cadena especificada, en este caso la cadena es un guión.

Para separar una cadena mediante espacios basta con declarar una cadena que contenga un espacio.

Por ejemplo:

```
>>> x = ' '.join(['Junta', 'las', 'cadenas'])
>>> print(x)
Junta las cadenas
```

Como podemos observar, cambiamos el guión por un espacio y éste ahora es nuestro separador al momento de concatenar.

- replace() -> Devuelve una nueva cadena reemplazando una parte de la cadena por otra, como la cadena es inmutable, es decir no podemos modificarla, lo que tenemos que hacer es asignar esa nueva cadena a otra variable o a la misma variable

```
>>> c = "17/05/1996"
>>> c = c.replace('/','-')
```

```
>>> print(c)
17-05-1996
```

En el ejemplo anterior cambiamos las diagonales `/` por un guión `-` y la guardamos en la misma variable.

- find() -> Recibe una cadena a buscar y devuelve la posición del primer carácter en la primera aparición, retorna `-1` cuando no encuentra alguna coincidencia

```
>>> x = "Ferrocarril"
>>> posicion = x.find("rr")
>>> print(posicion)
2
```

Como podemos observar el método `find()` nos regresa un 2, esto quiere decir que la cadena completa `"rr"` empieza en el índice 2.

- index() -> Hace lo mismo que find(), la diferencia es que cuando no encuentra alguna coincidencia marcar un error

```
>>> x =  "Ferrocarril"
>>> y = x.index("h")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> y = x.find("h")
>>> print(y)
-1
```

Observamos que al utilizar `index()` nos marca un error, cuando utilizamos `find()` nos retorna `-1`

- startswith() -> Realiza una búsqueda al inicio de la cadena, devuelve un booleano

```
>>> x =  "Ferrocarril"
>>> print(x.startswith("Fer"))
True
```

En este caso nos retornó True ya que nuestra cadena x empieza con `Fer` .

- endswith() -> Realiza una búsqueda al final de la cadena, devuelve un booleano

```
>>> x =  "Ferrocarril"
>>> print(x.endswith("rril"))
True
```

En este caso nos retornó True ya que nuestra cadena `x` termina con `rril`

- count() -> Devuelve el número de coincidencias

```
>>> x =  "Ferrocarril"
>>> coincidencias = x.count("r")
>>> print(coincidencias)
4
```

Observamos que nos retorna 4 porque la palabra "Ferrocarril" tiene 4 `"r"`

- upper() -> Devuelve la cadena convertida a mayúsculas
- lower() -> Devuelve la cadena convertida a minúsculas
- title() -> Devuelve la cadena con formato de título, es decir, con letra capital

Imprimir cadenas con formato

Cuando queramos imprimir una cadena pero además queramos imprimir números u otros tipos de datos, lo podemos hacer usando la impresión con formato. Por otro lado nos sirve para imprimir valores de variables. Para hacer esto hay de dos formas, la primera es usando el operador `%` .

Por ejemplo:

```
>>> print("El %s cuesta %d y es la bebida tradicional %s" % ("tequila", 200, "mexicana"))
El tequila cuesta 200 y es la bebida tradicional mexicana
```

En este ejemplo, el operador % se utiliza para combinar valores con cadenas.

El operador % (el del centro) requiere cadena <-->tupla

Como podemos observar usamos `%s` para imprimir una cadena y `%d` para imprimir un entero.

Las diferentes secuencias de formato son:

- Enteros -> %d
- Flotantes -> %f
- Cadenas -> %s

La segunda forma para imprimir con formato es usando el método `format()` , este nos ayuda a imprimir algún valor en determinado lugar.

Por ejemplo:

```
>>> print("El {0} cuesta {1} y es la bebida tradicional {2}".format("tequila",200,"mexicana"))
El tequila cuesta 200 y es la bebida tradicional mexicana
```

En este ejemplo, las llaves `{}` se sustituirán por lo que se encuentra entre paréntesis en nuestra función `format()` mediante el índice que pongamos, es decir, donde este `{0}` se cambiará por `"tequila"` , que es el elemento que se encuentra en el índice 0.

Lectura de valores desde teclado

Hay veces que al momento de hacer un programa necesitamos pedirle valores al usuario, esto podemos hacerlo con la función `input()` , dentro de los paréntesis podemos ponerle un letrero al usuario para que sepa qué debe ingresar

Por ejemplo:

```
>>> x = input("Ingresa algo: ")
Ingresa algo: Python AM
>>> print(x)
Python AM
```

En este ejemplo pedimos al usuario que ingrese algo y lo guarde en la variable `x` , posteriormente imprimes x y vemos qué fue lo que ingresó el usuario.

> Nota: La función `input()` retorna una cadena, pero podemos convertirla a los diferentes tipos de datos. Por ejemplo, si vamos a trabajar con números debemos `castear` lo que nos retorne input con la función `int()`

```
>>> x = int(input("Ingresa un número: "))
Ingresa un número: 24
>>> print(x)
24
```

Ahora nuestra variable `x` ya no es una cadena, es un entero.

Listas - Mutable

> Nota: Para delimitar una lista siempre será con [ ] y sus elementos estarán separados por comas `,`

```
>>> enteros = [1,2,3,4,5]
>>> cadenas = ["Python", "AM", "es el mejor curso"]
>>> variosTipos = ["py", 1, 3.141592, True, [1,["lista", "de", "listas"], 3]]
```

Podemos declarar listas de enteros, flotantes, cadenas, listas, tuplas, etc. En resumen, puede tener como elemento cualquier tipo de dato.

En el ejemplo 1 declaramos una lista de enteros.

En el ejemplo 2 declaramos una lista de cadenas.

En el ejemplo 3 declaramos una lista de varios tipos, como podemos observar agregamos una lista como un elemento y dentro de esta otra lista. cuando hacemos esto las listas se van a manejar por jerarquías.

Saber la longitud de una lista

Para saber cuántos elementos tiene una lista usamos la función `len()`

```
>>> x = [1,2,3,4]
>>> tamano = len(x)
>>> print(tamano)
4
```

Para saber cuál es valor máximo y el valor mínimo de una lista usamos las funciones `max()` y `min()`

```
>>> x = [1,2,3,4]
>>> print(max(x))
4
>>> print(min(x))
1
```

> Nota: Las funciones `len(), max(), min()` también sirven para tuplas, diccionarios, cadenas (en este caso contará el número de caracteres que contiene y basándose en el alfabeto para max y min).

Indexación (Manejo por índices)

Igual que las cadenas, las listas son un conjunto de elementos, la diferencia es que las listas aceptan cualquier tipo de dato. Por esto mismo podemos acceder a ellas mediante índices

```
>>> variosTipos = ["py", 1, 3.141592, True, [1,["lista", "de", "listas"], 3]]
>>> print(variosTipos[0])
py
>>> print(variosTipos[4])
[1, ['lista', 'de', 'listas'], 3]
>>> print(variosTipos[4][0])
1
>>> print(variosTipos[4][1])
['lista', 'de', 'listas']
>>> print(variosTipos[4][1][2])
listas
```

En este ejemplo declaramos una lista de varios tipos, vamos a ver como se manejan estas listas mediante índices.

En el primer print, estamos accediendo al índice 0 de nuestra lista, es decir, el primer elemento de nuestra lista, como podemos ver es una cadena.

En el segundo print, estamos accediendo al índice 4, en este índice encontramos una lista, por lo tanto también podemos manejar sus índices.

En el tercer print, accedemos al índice 0 de la lista que está **dentro** de la lista en el índice 4, es decir, accedemos a un índice de una lista de listas.

En el cuarto print, accedemos a una lista, pero como vemos también es una lista, por lo tanto también podemos acceder a sus valores mediante índices.

En el quinto print, accedemos a esa última lista, si vemos primero accedemos al `índice 4` de la lista más externa, posteriormente accedemos al `índice 1` de la lista "intermedia" por así decirlo. Por último accedemos al `índice 2` de la lista más interna, este índice ya contiene un valor que es la cadena "listas".

También podemos manejar índices negativos. Cuando hagamos esto, python tomará la lista desde el último elemento dándole a ese valor el índice `-1` .

Por ejemplo, si tenemos una lista con 5 elementos, el último elemento estará en el índice 4 `(números de elementos - 1)` , a su vez, python tomará ese índice como -1. Cuando el índice sea el 3, este a su vez también será el -2, y así consecutivamente.

```
>>> x = ["uno", "dos", "tres", "cuatro", "cinco"]
>>> print(x[4])
cinco
>>> print(x[-1])
cinco
>>> print(x[3])
cuatro
>>> print(x[-2])
cuatro
```

Igual que las cadenas, podemos manejar un rango de índices.

```
>>> x = [1,2,3,4,5]
>>> print(x[:4])
[1, 2, 3, 4]
```

> Nota: lista[indice_inicial : indice_final]

Cuando manejamos rangos de índices, también podemos cambiar el incremento, esto lo podemos hacer con un tercer parámetro separado por `:`

```
>>> print(x[:4:2])
[1, 3]
```

> Nota: lista[indice_inicial : indice_final : incremento]

En el ejemplo anterior le decimos que vaya del inicio de la lista hasta el índice 3 (índice 4 - 1) e incremente de 2 en 2. Por lo tanto solo nos imprime el índice 0 y el índice 2

> Nota: Para mayor información o más explicada, ir a la parte de indexación de cadenas, es prácticamente lo mismo.

Modificando listas

Gracias a que las listas son mutables, podemos modificar sus valores, esto lo podemos hacer mediante sus índices. Tenemos que acceder a un índice y asignarle el nuevo valor.

```
>>> x = [1,2,3]
>>> x[0] = "uno"
>>> print(x)
['uno', 2, 3]
```

Métodos de listas

Como estamos trabajando con un tipo de dato mutable, todos los métodos afectarán directamente a la lista

- append() -> Agrega un elemento al final de una lista

```
>>> x = ["uno","dos","tres"]
>>> x.append("cuatro")
>>> print(x)
['uno', 'dos', 'tres', 'cuatro']
```

- insert() -> Agrega un valor en una posición determinada

> lista.insert(indice, valor)

```
>>> x = ["uno","dos","tres"]
>>> x.insert(0,"cero")
>>> print(x)
['cero', 'uno', 'dos', 'tres']
```

- remove() -> Elimina la primera coincidencia del valor dado, no retorna nada

```
>>> x = ["uno","dos","tres"]
>>> x.remove('dos')
>>> print(x)
['uno', 'tres']
```

- pop() -> Regresa el último dato de la lista y lo elimina

```
>>> x = ["uno","dos","tres"]
>>> valor = x.pop()
>>> print(valor)
tres
>>> print(x)
['uno', 'dos']
```

- reverse() -> Invierte la lista

```
>>> x = ["uno","dos","tres"]
>>> x.reverse()
>>> print(x)
['tres', 'dos', 'uno']
```

- sort() -> Ordenar listas, la lista debe ser del mismo tipo de dato

```
>>> x = ["uno","dos","tres"]
>>> x.sort()
```

```
>>> print(x)
['dos', 'tres', 'uno']
```

Cuando trabajamos con cadenas, ordena por la primera letra.

`sort()` ordena los valores de menor a mayor, para hacerlo de mayor a menor podemos hacer que el parámetro reverse sea igual a True.

```
>>> y = [2,5,1,3,6]
>>> y.sort()
>>> print(y)
[1, 2, 3, 5, 6]
>>> y.sort(reverse = True)
>>> print(y)
[6, 5, 3, 2, 1]
```

Por defecto trae el parámetro `reverse` en False.

- extend() -> Concatena dos listas

```
>>> x = [1,2,3]
>>> y = ["uno","dos","tres"]
>>> x.extend(y)
>>> print(x)
[1, 2, 3, 'uno', 'dos', 'tres']
>>> print(y)
['uno', 'dos', 'tres']
```

A la variable a la que se le aplica el método es la cadena que se va a modificar, en el ejemplo anterior, la lista `x` es la que se modifica.

- count() -> Devuelve el número de apariciones de una cadena o número

```
>>> x = [1,4,2,5,2,9,2]
>>> contador = x.count(2)
>>> print(contador)
3
```

Pertenencia y borrado mediante un índice

- del -> Borra un índice, un rango de índices o toda la lista

```
>>> x = [1,2,3,4]
>>> del x[0]
>>> print(x)
[2, 3, 4]
```

```
>>> x = [1,2,3,4]
>>> del x[1:3]
>>> print(x)
[1, 4]
```

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

- in, not in -> Pertenencia, dice si un valor está en la lista

```
>>> lista = ["hi","mano","ala","zaz", 5]
>>> print("zaz" in lista)
True
>>> print("zaz" not in lista)
False
```

Tuplas - Inmutable

> Nota: Para delimitar una tupla siempre será con ( ) y sus elementos están separados por comas `,`

Es una estructura similar a listas pero no pueden modificarse, solo crearse. Una tupla es inmutable

```
>>> tupla = (1,"cadena", [1,2])
```

Como las listas no pueden ser modificadas, solo podemos acceder a sus valores, saber cuántos elementos tiene, manejarlo por índices, etc.

```
>>> print(tupla[0])
1
>>> print(tupla[:2])
(1, 'cadena')
>>> print(len(tupla))
3
>>> print(1 in tupla)
True
```

Si las tuplas son tan similares a las listas, ¿por qué usarlas?

- Las tuplas son mas rápidas
- Los datos se protegen contra escritura
- Algunas claves se pueden usar con diccionarios

Aunque las tuplas son inmutables, python nos da la flexibilidad de cambiar de un tipo de dato a otro, para hacer esto usamos las funciones `list()` para cambiar de un tipo de dato a listas y `tuple()` para cambiar a tuplas.

```
>>> lista = list((1,2,3,4))
>>> tupla = tuple([1,2,3,4])
>>> print(lista,tupla)
[1, 2, 3, 4] (1, 2, 3, 4)
```

Para saber qué tipo de dato es, nos apoyamos con la función `type()`, la cual recibe una variable

```
>>> type(lista)
<class 'list'>
>>> type(tupla)
<class 'tuple'>
```

Del ejemplo anterior observamos que efectivamente nuestra variable lista es un objeto de la clase list y tupla es un objeto de la clase tuple. El término `objeto` lo veremos más adelante.

Empacar y Desempacar

El término empacar nos ayuda a sacar el contenido de una lista y/o tupla y guardarlo en diferentes variables. Desempacar es lo contrario, meter varios valores en una sola variable, al hacer esto creamos una tupla.

```
>>> x = [1,2,3,4,5]
>>> a, b, c, d, e = x
>>> print(b)
2
```

> Nota: Tiene que ser la misma cantidad de variables a la cantidad de datos que tenga nuestra lista o tupla. En este caso nuestra lista tiene 5 elementos, por lo tanto necesitamos 5 variables, si llegáramos a tener menos, nos marcará un error.

```
>>> a = 1, 2, 3
>>> print(a[1])
2
>>> print(type(a))
<class 'tuple'>
```

Para empacar solo basta con asignarle varios valores a una sola variable. Esto nos generará una tupla.

Diccionarios - Mutable

> Nota: Para delimitar un diccionario siempre será con { }, clave : valor

Es un conjunto desordenado de datos clave-valor (key-values)

Las claves de un diccionario deben ser únicas

Las claves deben de ser inmutables

```
>>> diccionario = { 'a' : 55, 5 : "esto es un string", 'a' : "segundo valor", 'a': "tercer valor"} # Si tenemos la mis
>>> print(diccionario)
{'a': 'tercer valor', 5: 'esto es un string'}
```

Podemos hacer uso de la función `dict()` para crear un diccionario

```
>>> dic = dict(python = "mejor curso", a = "dos", adios = [1,2]) # Cuando las llaves (keys) son puras cadenas, podemos
>>> print(dic)
{'python': 'mejor curso', 'adios': [1, 2], 'a': 'dos'}
```

Las claves y valores pueden ser de tipos diferentes

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> print(y[2.5])
flotante
```

Modificando un diccionario

Para modificar un valor, tenemos que acceder mediante su llave.

```
>>> diccionario = { 'a' : 55, 5 : "esto es un string"}
>>> diccionario['a'] = False # Si la llave/clave se encuentra, actualiza, sino se crea
>>> print(diccionario)
{5: 'esto es un string', 'a': False}
>>> diccionario['c'] = 'nuevo string' # Crearíamos clave/valor
```

```
>>> print(diccionario)
{'c': 'nuevo string', 5: 'esto es un string', 'a': False}
```

Pertenencia y número de elementos

Para saber si una llave pertenece a un diccionario usamos la palabra reservada `in` , no podemos saber si un valor existe. Para saber el número de clave/valor que tiene un diccionario usamos la función `len()`

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> print(len(y))
5
>>> print("blanco" in y) # Imprime False, ya que blanco no se encuentra en las llaves del diccionario
False
>>> print("lista" in y) # Imprime True, ya que lista se encuentra en las llaves del diccionario
True
```

Métodos de diccionarios

- get() -> Devuelve el valor de una llave. Devuelve None si no encuentra la llave

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> valor = y.get(2.5)
>>> print(valor)
flotante
```

En caso de no encontrarse la llave, podemos poner un valor por defecto para que no devuelva None.

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> valor = y.get('z', 34) #como segundo argumento se pone lo que retornará en caso de que la llave no se encuentre
>>> print(valor)
34
```

- keys() -> Devuelve un objeto de tipo "dict_keys" el cual contiene las claves de un diccionario. Podemos castearlo a lista para trabajar con él

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> claves = list(y.keys())
>>> print(claves)
['lista', 2, 'tupla', 2.5, 'uno']
```

- values() -> Devuelve un objeto de tipo "dict_values" el cual contiene los valores de un diccionario. Podemos castearlo a lista para trabajar con él

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> print(valores)
[[1, 2, 3], 'dos', (0, 1), 'flotante', 1]
```

- items() -> Devuelve un objeto de tipo "dict_items" el cual contiene una lista de tuplas, cada tupla contiene dos valores, uno es la clave y otro es el valor. Podemos castearlo a lista para trabajar con estos elementos

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> clave_valor = list(y.items())
>>> print(clave_valor)
[('lista', [1, 2, 3]), (2, 'dos'), ('tupla', (0, 1)), (2.5, 'flotante'), ('uno', 1)]
```

- update -> Agrega los elementos de un diccionario a otro

```
>>> y = {"uno" : 1, 2 : "dos", 2.5 : "flotante", "lista" : [1,2,3], "tupla" : (0,1)}
>>> x = {'z' : 23, 'w' : 88}
>>> y.update(x) #El que queremos que incremente . update (con qué incrementarlo)
>>> print(y)
{'w': 88, 'lista': [1, 2, 3], 'tupla': (0, 1), 'uno': 1, 2.5: 'flotante', 2: 'dos', 'z': 23}
```

- clear() -> Borra el contenido diccionario

```
>>> y.clear()
>>> print(y)
{}
```

> Nota: Borra solo el contenido, no la variable

- del -> Palabra reservada para borrar un elemento del diccionario o la variable como tal

```
>>> dic = dict(python = "mejor curso", a = "dos", adios = [1,2])
>>> print(dic)
{'adios': [1, 2], 'a': 'dos', 'python': 'mejor curso'}
>>> del dic['a']
>>> print(dic)
{'adios': [1, 2], 'python': 'mejor curso'}
>>> del dic
>>> print(dic)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'dic' is not defined
```

> Nota: Recordar que una variable es un pedazo de memoria, es decir, solo una referencia. cuando la borramos con `del` borramos esa referencia, por lo tanto la variable ya no existe