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).

Conversion a otras bases numéricas

En Python también es posible cambiar de una base numérica a otra gracias a métodos que el propio lenguaje nos da. Si no se está familiarizado con otras bases numéricas, sugerimos visitar esta página para poder entender el proceso de conversión. Las bases numéricas más conocidas (aparte de la decimal) son la binaria, hexadecimal y octal. Si nosotros queremos convertir de decimal hacia otra base numérica podemos utilizar los métodos `bin()` para el caso de números binarios, `hex()` para números hexadecimales y `oct()` para números octales.

```
>>> y = 11
#Base binaria
>>> binario = bin(y)
>>> print(binario)
0b1011
#Base octal
>>> octal = oct(y)
>>> print(octal)
0o13
#Base hexadecimal
>>> hexa = hex(y)
>>> print(hexa)
0xb
```

Si lo que se desea es convertir de una base cualquiera a un número de base decimal, podría hacerse de la siguiente forma:

```
>>> x = 10
# Conversión de binario a decimal
>>> print(int(str(x),2))
2
>>> y = "a"
#Conversión de hexadecimal a decimal
>>> print(int(str(y),16))
10
```

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.

Conjuntos - Mutable

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

Es una colección desordenada de objetos/datos Un conjunto no almacena elementos repetidos Un conjunto no puede tener objetos mutables, sin embargo un conjunto es mutable

```
>>> x = {(1,2,3)} # Lo acepta, ya que una tupla es un objeto inmutable
>>> x = {[1,2,3]}   # Marca error, ya que una lista es un objeto mutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Creación de un conjunto

Para crear un conjunto hay de 2 formas, la primera es haciendo uso de la función `set()` la cual recibe como parámetro una lista o una tupla.

```
>>> x = set([1,2,3])
>>> x = set((1,2,3))
>>> print(x)
{1, 2, 3}
```

La segunda es declarándolo con llaves `{ }`.

```
>>> conjunto = {1, 5, 3, 10, 7}
>>> print(conjunto)
{1, 10, 3, 5, 7}
```

Operaciones entre conjuntos

- add() -> Agrega un elemento al conjunto, si ya existe no lo agrega

```
>>> x = {1,4,6,9}
>>> x.add(5)
>>> print(x)
{1, 4, 5, 6, 9}
>>> x.add(9)
>>> print(x)
{1, 4, 5, 6, 9}
```

- remove() -> Elimina un elemento especificado

```
>>> x = {1,4,5,6,9}
>>> x.remove(4)
>>> print(x)
{1, 6, 9}
```

- pop() -> Devuelve el primer elemento

```
>>> x = {1,4,5,6,9}
>>> eliminado = x.pop()
>>> print(eliminado)
1
>>> eliminado = x.pop()
>>> print(eliminado)
4
```

- in -> Pertenencia, devuelve un booleano si un elemento está dentro de un conjunto

```
>>> x = {1,4,5,6,9}
>>> print("¿2 se encuentra en el conjunto?", 2 in x)
¿2 se encuentra en el conjunto? False
```

- union() -> Retorna la unión entre 2 conjuntos como un nuevo conjunto

> Nota: También podemos hacer la unión entre conjuntos con el operador `|`

```
>>> x = {1,4,5,7,9}
>>> y = set([3,4,5,6])
>>> z = x | y     # con operador
>>> print("Union: ", z)
Union:  {1, 3, 4, 5, 6, 7, 9}
>>> z = x.union(y)  # con método
>>> print("Union: ", z)
Union:  {1, 3, 4, 5, 6, 7, 9}
```

- intersection() -> Retorna la intersección de 2 conjuntos como un nuevo conjunto (datos iguales entre los 2 conjuntos)

> Nota: También podemos hacer la intersección entre conjuntos con el operador `&`

```
>>> x = {1,4,5,7,9}
>>> y = set([3,4,5,6])
>>> z = x & y           # con operador
>>> print("Interseccion: ",z)
Interseccion:  {4, 5}
>>> z = x.intersection(y)   # con método
>>> print("Interseccion: ",z)
Interseccion:  {4, 5}
```

- difference() -> Retorna la diferencia entre dos o más conjuntos como un nuevo conjunto (todos los elementos que están en este conjunto pero no en otros)

> Nota: También podemos hacer la diferencia entre conjuntos con el operador `-`

```
>>> x = {1,4,5,7,9}
>>> y = set([3,4,5,6])
>>> z = y - x  # Importa la forma en como se reste
>>> z = y.difference(x)  # con método
>>> print("Diferencia y - x: ",z)
Diferencia y - x:  {3, 6}
>>> z = x - y          # con operador
>>> z = x.difference(y) # con método
>>> print("Diferencia x - y: ",z)
Diferencia x - y:  {1, 9, 7}
```

> Nota: Ya que es una resta, importa mucho el orden, es decir, qué conjunto restes

- symmetric_difference() -> Retorna la diferencia simétrica de dos conjuntos como un nuevo conjunto (es el conjunto de elementos que están en A o B, pero no en ambos)

> Nota: También podemos hacer la diferencia simétrica entre conjuntos con el operador ^

```
>>> x = {1,4,5,7,9}
>>> y = set([3,4,5,6])
>>> z = x ^ y                    # con operador
>>> z = x.symmetric_difference(y) # con método
>>> print("Diferencia simetrica: ",z)
Diferencia simetrica:  {1, 3, 6, 7, 9}
```

Frozensets - Inmutable

No es más que un conjunto pero inmutable, es decir, que no podemos modificarlo.

```
>>> a = frozenset([1,2,3,4])
>>> a[0] = "hola"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'frozenset' object does not support item assignment
```

Como podemos ver nos marca error cuando queremos modificarlo.

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

Control de flujo

If-Else-Elif

Algunas veces en nuestros programas es necesario que tomemos algunas “decisiones”, esto en el sentido de que necesitamos decidir si ejecutar una pieza en especial de código o no, o tal vez, dadas ciertas condiciones tendríamos varias alternativas de código que ejecutar.

La sentencia **If** evalúa básicamente una operación lógica, es decir una expresión que de como resultado verdadero ó falso (true ó false), y ejecuta la pieza de código siguiente siempre y cuando el resultado sea verdadero.

Por ejemplo, realizaremos un programa que imprima la leyenda "Escriba un número positivo: " y de acuerdo a lo que reciba, si el número ingresado es mayor a cero, entonces imprima la leyenda "Ha escrito el número n" (según sea el caso). Si no, entonces imprima "¡Le he dicho que escriba un número positivo!".

```
numero = int(input("Escriba un número positivo: "))
if numero < 0:
     print("Ha escrito el número {numero}")
print("¡Le he dicho que escriba un número positivo!")
```

Ahora, probaremos el programa ingresando lo siguiente:

```
Escriba un número positivo: 2
Ha escrito el número 2

Escriba un número positivo: -10
¡Le he dicho que escriba un número positivo!
```

El utilizar **else** lo que hace es que podemos tener la capacidad de ejecutar un código alternativo en caso de que el resultado lógico de la expresión evaluada sea falso. Esta es una forma de tener varias alternativas de código a ejecutar.El uso con un ejemplo seria el siguiente.

```
a = 10
if (a != 10):
       print('La variable es diferente de 10!')
else:
      print('La variable es igual a 10!')
print('fin')
```

En este ejemplo se puede ver que la expresión sólo será verdadera si la variable es diferente de 10, dado que esto no es cierto, ejecutaremos la alternativa con “else”, el resultado de este programa sera el siguiente.

```
La variable es igual a 10!
fin
```

While

Tiene su uso en el control de flujo iterativo; su función consiste en ejecutar un bloque de instrucciones hasta que la condición deje de ser verdadera. Es importante evaluar nuestro algoritmo con el propósito de llegar a un fin en nuestro ciclo, pues en caso contrario tendremos un error lógico y nunca saldremos de él.

Este es un ejemplo de un ciclo infinito:

```
i = 10
while i > 0:
     print("Esta es la iteración número {0}.".format(i))
```

Este es un ejemplo de un ciclo finito:

```
i = 10
while i > 0:
     print("Esta es la iteración número {0}.".format(i))
     i += 1
```

For

La estructura de repetición for es útil para repetir un bloque de código en un número determinado de veces. Recordemos además que el ciclo for en Python es capaz de iterar sobre diversos objetos iterables, como lo son las listas, las tuplas, los diccionarios, las cadenas.

For each

Con un "For each" podemos acceder a los elementos de una secuencia de datos con ayuda de una variable auxiliar que nos ayudará a iterar sobre ellos.

Pongamos el ejemplo de la siguiente lista de números:

```
### Programa en Python que imprime en pantalla los elementos de una lista. ###
listaDeNumeros = [1,2,3,4,5] # esta será nuestra secuencia de datos
#si queremos acceder a los elementos de esta lista utilizaremos la estructura de repetición for
for variableAuxiliar in listaDeNumeros:
    print(variableAuxiliar)
```

la función de la variableAuxiliar para la primera iteración en este ejemplo es de tomar el valor del primer elemento de la listaDeNumeros, es decir, para la primera iteración la variableAuxiliar tomará el valor de 1, para la segunda iteración tomará el valor de 2 y así sucesivamente para todos los elementos en esta lista.

For range

Genera una tupla de números enteros que nos servirán como auxiliares para iterar sobre un conjunto de elementos. Podemos decidir qué número de iteraciones se realizarán, rangos y pasos. Recordemos que los rangos de Python no toman el último valor.

Statements in Python

- *Break*: Sólo puede utilizarse anidándolo dentro de un ciclo for o while, pero no anidado en una función o definición de clase dentro de ese bucle. Omite la cláusula “else” si tiene una. Su misión es finalizar el ciclo.
- *Continue*: Termina la iteración actual, pero continúa con el ciclo probando de nuevo con la expresión. No finaliza el ciclo, sólo la iteración en la que se encuentra.
- *Pass*: Esta expresión la usamos cuando aún no sabemos con certeza qué bloque de código escribir. No pasa nada cuando se ejecuta y pasa a la siguiente iteración, por lo que se puede decir que funciona como un relleno temporal.

Implementaciones

Do-While en Python

En python no existe *do-while* como tal, pero si alternativas que nos ayudan a implementarlo. El concepto de *do-while* consiste en que ciertas instrucciones se van a ejecutar una vez, y después cabe la posibilidad de ejecutarlas de nuevo dentro de un bucle while, pero con la certeza de que al menos una vez ya se ha ejecutado.

Switch-case

Al igual que el *do-while*, no existen palabras reservadas para un *switch-case* pero existen alternativas de implementación. Estas van desde el uso de diccionarios hasta sentencias sencillas de if-else.

Programación funcional

Estructura de una función

Para empezar

Un paradigma es una forma en la que se puede plantear, analizar y resolver un problema, en a programación tenemos bastantes paradigmas, sin embargo los principales son la programación estructurada, usado en lenguajes como C, la programación orientada a objetos, utilizada en lenguajes como Java, C#, C++, por último, tenemos a la **programación funcional**.

Python es un lenguaje **multiparadigma**, es decir, podemos trabajar en él con más de un sólo paradigma.

En este manual nos concentraremos en la programación funcional, que debido a su nombre se puede deducir que se trabajará con funciones.

Requisitos

- Conocimientos de tipos de datos en Python.
- Conocimientos en estructuras de control.
- Python3 instalado en su equipo.

¿Qué es una función?

Una función es un bloque de código ejecutable que puede ser reutilizado, puede recibir parámetros y puede regresar un valor.

¿Para qué me sirve una función?

Su principal objetivo es evitar la repetición innecesaria de código, haciendo así más eficiente nuestro proyecto.

¿Cúal es la estructura de una función en Python?

```
def nombre_de_mi_función(parametros):
    #Código a ejecutar por mi función

    return #Valor a regresar de mi función
```

- ***def*** es una palabra reservada en Python que nos permitirá crear una nueva función.
- ***nombre_de_mi_función*** será el identificador de nuestra función, es decir, como podremos mandarla a llamar en cualquier parte del código.
- ***parámetros*** son los parámetros que la función usará, puede estar vacío en caso de no necesitar ningún parametro para nuestra función.
- ***return*** es una palabra reservada de Python que le indicará al intérprete que ese valor será el que nuestra función arroje, puede no estar si nuestra función no está diseñada para regresar algún valor, sin embargo, cuando una función llega a el return, la función terminará.

Ejemplos

```
# Ejemplo de una función que recibe dos parámetros, los suma y regresa la suma de ambos

def suma(numero_a, numero_b):

    resultado = numero_a + numero_b

    return resultado
```

Al mandarla a llamar podremos obtener todo tipo de resultados dependiendo de los parámetros que envíemos.

```
>>> suma(5, 6) # Caso en el que mandemos una suma de dos números enteros
>>> 11
```

```
>>> suma(-3.00 + 4.50) # Caso en el que mandemos una suma de dos números flotantes
>>> 1.50

>>> suma('Python AM es' + ' muy cool') # Caso en el que mandemos dos parámetros del tipo string, en este caso, concate
>>> Python AM es muy cool
```

Como se puede observar, la función regresará cualquier tipo de valor operando los parámetros que se le envíen, siempre y cuando estos parámetros sean operables entre sí, es decir, si intentaramos usar suma con dos parámetros no operables entre sí, tendríamos un error.

```
>>> suma(1, 'Python')
TypeError: Can't convert 'int' object to str implicitly
```

Es por eso que cuando trabajemos con funciones debemos ser muy cuidadosos al momento de pasar los parámetros, de lo contrario podría provocar errores en el proyecto.

Ese fue un ejemplo de una función extremadamente sencilla, si desea ver algunos ejemplos un poco más avanzados puede checar en la sección Ejemplos/Funciones.py de este repositorio.

Llamadas a funciones

Como ya hemos mencionado, las funciones son objetos decódigo ejecutable y reutilizable, sin embargo, aún no sabemos como mandarlas a llamar, para ello, pongamos un ejemplo sencillo, supongamos que tenemos el siguiente script:

```
def suma(a, b):
    resultado = a + b
    return resultado


def resta(a, b):
    resultado = a - b
    return resultado
```

Hasta este momento, tenemos dos funciones llamadas suma y resta, así que crearemos una tercera función que será la interfaz con el usuario, la que permitirá que el usuario escoja si se ejecuta una u otra.

```
def calculadora():
    opcion_del_usuario = input('¿Qué operación desea realizar?')

    if opcion_del_usuario == '+':
        print('Suma\n')
        numero_a = input('Escriba el primer número a sumar: ')
        numero_b = input('Escriba el segundo número a sumar: ')
        resultado_de_suma = suma(numero_a, numero_b)
        print(resultado_de_suma)

    elif opcion_del_usuario == '-':
        print('Resta\n')
        numero_a = input('Escriba el primer número a restar: ')
        numero_b = input('Escriba el segundo número a restar: ')
        resultado_de_resta = resta(numero_a, numero_b)
        print(resultado_de_resta)
```

Ahora hemos logrado mandar a llamar dos funciones dentro de una tercera función, así ya no tuvimos que reescribir el código dentro de la función `calculadora()`

Por último tenemos que llamar a nuestra función desde el `__main__` , es decir nuestro código principal, para ello usaremos un bloque de código muy empleado dentro de Python.

```
if __name__ == '__main__':
```

Pero ¿Qué significa esa línea de código? Esto está ligado al intérprete de Python, cuando este lee un archivo de código, **ejecuta *todo* el código** dentro de él. Todos los módulos en Python tienen un atributo llamado `__name__` que define el espacio de nombres en el que se está ejecutando, se usa para identificar a un módulo en el momento de las importaciones. Por otro lado, `__main__` es el nombre del ámbito en el que se ejecuta el código principal.

Dicha línea de código podría traducirse a lenguaje humano como: *Si estamos en el main del archivo, entonces ejecutar el bloque de código dentro*

Para ser un poco más gráficos la usaremos en el ejemplo de la calculadora básica anterior.

```
if __name__ == '__main__':
    calculadora()
```

Esto hará que cuando ejecutemos nuestro script desde la terminal, la función que se ejecutará será `calculadora()`

Recursividad

La recursividad es una estrategia de programación, en la cual mandamos a llamar a una función, dentro de la misma función. Al principio sonará y parecerá un tanto confuso, sin embargo, es bastante útil al momento de desarrollar algoritmos para la resolución de problemas.

Pongamos un ejemplo, calcularemos el factorial de un número con ciclos de iteraciones:

```
def  factorial_iteraciones(num):
    resultado = 1

    for i in range(num):
        resultado *= (i + 1)

    return resultado

if __name__ == '__main__':
    print(factorial_iteraciones(5))

# Resultado: 120
```

Dicho valor se pudo calcular de forma rápida, sin embargo, ¿fue la más eficiente? Calculemos el tiempo de ejecución de dicho algoritmo en el mismo ejemplo, añadiendo sólo unas cuantas líneas de código más.

```
from time import time
def  factorial_iteraciones(num):
    resultado = 1

    for i in range(num):
        resultado *= (i + 1)

    return resultado

if __name__ == '__main__':
    inicio = time() # Calcula el momento en el que inica el algoritmo
    print(factorial_iteraciones(5))
    final = time() # Calcula el momento en el que termina
    tiempo_de_ejecucion = final - inicio # El resultado es el tiempo final menos el tiempo inicial
    print(tiempo_de_ejecucion)

# Resultado: 120
# Tiempo de ejecución: .0000264 segundos
```

Como vemos, el algoritmo de cálculo del factorial tiene 4 líneas y tarda .0000264 segundos en ejecutarse, sin embargo, hay formas de hacerlo mucho más corto en extensión y ciertamente más rápido.

Veamos como calcular el factorial de un número de forma recursiva:

```
def factorial_recursivo(num):
    if num == 0:
        return 1
    else:
        return num * factorial_recursivo(num - 1)

if __name__ == '__main__':
    inicio = time() # Calcula el momento en el que inica el algoritmo

    factorial = factorial_recursivo(5)
    print(factorial)

    final = time() # Calcula el momento en el que termina
    tiempo_de_ejecucion = final - inicio # El resultado es el tiempo final menos el tiempo inicial

    print(tiempo_de_ejecucion)

# Resultado: 120
# Tiempo de ejecución: .0000238
```

Se puede apreciar un aumento leve en la velocidad de ejecución del algoritmo, sin embargo, aún podemos hacer más eficiente esta función, pero eso ya se verá dentro del curso.

Generadores

¿Qué es un generador?

Un generador puede ser descrito a groso modo como una función que nos regresará de poco a poco sus valores, es decir, cada vez que la llamemos nos va a regresar valores nuevos. Por ejemplo, una función que nos devuelva el siguiente número par, esto no se podría hacer sin los generadores, ya que estos mismos son infinitos.

¿Cómo se construye un generador?

Para poder construir un generador basta con usar la orden `yield`, esta orden lo que hará será regresar un valor tal como lo hace la orden `return`, pero además congelará la ejecución de la función hasta que la volvamos a llamar, para tenerlo un poco más claro, pongamos un ejemplo:

```
def multiplos_de_tres():
    index = 1
    while True:
        yield index*3
        index += 1

if __name__ == '__main__':
    for multiplo in multiplos_de_tres():
        print(multiplo)
```

En este fragmento de código estamos creando un generador, que nos regresará un valor, múltiplo de 3 y luego pausaará su ejecución hasta el momento que la volvamos a llamar. Recordemos que la palabra reservada `return` finaliza la función y nos regresa el control, sin embargo, la palabra reservada `yield` no finaliza del todo la función, pero de igual forma nos regresa el control, para poder continuar la ejecución de otros bloques de código sin la necesidad de perder el estado de ejecución de la función con yield.

Por otro lado, un generador también puede ser considerado como un objeto iterable, que no regreesará en cada función un valor, de nuevo, quizá esto no quede muy claro al principio, pero veremos algunos ejemplos que nos ayudarán a entenderlo mejor. Veamos las maneras de llenar una lista, por ejemplo, una lista que contenga los elementos de otra lista pero elevados al cuadrado:

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x_cuadrados = []

for elemento in x:
    cuadrado = elemento ** 2
    x_cuadrados.append(cuadrado)
```

En este caso ha sido efectiva nuestra solución al problema, sin embargo, no es la más eficiente, continuación, mostraré como reolver el mismo problema pero con el uso de un **generador**

```
x = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x_cuadrados = (elemento ** 2 for elemento in x)
lista_cuadrados = list(x_cuadrados)
```

Y listo, tenemos exactamente el mismo resultado de una forma más rápida y en menos líneas de código, pero, ¿Qué estamos haciendo en ese script?

Para empezar la linea: `x_cuadrados = (elemento ** 2 for elemento in x)` primero hablaremos de `elemento ** 2 for elemento in x` en esta parte del código iteraremos sobre una lista y al elemento sobre el cual nos encontramos, lo elevaremos al cuadrado, por último, este elemento lo agregaremos a un objeto de la clase generador.

Además, al llenar un generador, podemos incluir condiciones, pongamos de ejemplo, queremos llenar un generador con sólo los números múltiplos de 3 del 1 al 1000, primero lo haremos con un ciclo for

```
multiplos = []

for elemento in range(1000):
    if elemento % 3 == 0:
        multiplos.append(elemento)
```

El tiempo de ejecución de este código de aproximadamente de 0.000420 segundos, ahora veamos el mismo ejemplo pero con un generador esta vez

```
multiplos = (elemento for elemento in range(1000) if elemento % 3 == 0)
multiplos_lista = list(multiplos)
```

De nuevo, podemos ver que la cantidad de código se redujo bastante, y hasta en tiempo de ejecución resulta más eficiente, ya que en promedio es de 0.000146 segundos

Built-ins de Python

builtins

Un built-in es una función, constantes o clase ya definida dentro de los ficheros de Python, es decir, tenemos acceso todo el tiempo a esas funciones, constantes o clases sin tener que volver a codificarlas o en su defecto importarlas, aunque sí hay forma de importarlas.

El ejemplo más importante de estas funciones en Python3 es la famosa función `print()` que ya es una función que hemos usado bastante. Python cuenta con muchísimas funciones de este tipo, a lo largo de esta sección haremos una pequeña reseña de las funciones built-in más usadas.

Módulo builtins

Dentro de Python existe un módulo llamado `builtins` que es el código donde están almacenadas estás funciones, por ejemplo:

```
print('El curso de Python AM es cool')
# Output: 'El curso de Python AM es cool'
# Este es un ejemplo de como usamos la función print() de la forma tradicional
# Sin embargo podemos hacr lo siguiente y tendrá el mismo efecto:

import builtins
builtins.print('El curso de Python AM es cool')
# Output: 'El curso de Python AM es cool'
# Podríamos decir que builtins.print() es el nombre completo de nuestra función print()
```

Otras funciones built-in

abs(num)

Abs regresará el valor aboluto de un número.

```
>>> abs(10)
10
>>> abs(-14.6)
14.6
```

bin(num)

Bin convierte el parámetro que recibe en binario junto con el prefijo '0b'.

```
>>> bin(2)
'0b10'
>>> bin(-10)
'-0b1010'
```

chr(num)

Chr regresa el caracter en Unicode correspondiente al número que recibe como parámetro.

```
>>> chr(5359)
'ᓯ'
>>> chr(4360)
'ᑈ'
```

dir([objeto])

Dir regresará la lista de los nombres llamables en el scope local en caso de no recibir parámetros, en caso de recibir un objeto regresará una lista con los nombres de los métodos y atributos del objeto.

```
>>> dir() # Caso sin parámetros
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>> # Definimos una función
>>> def count(lista):
...     counter = 0
...     for i in lista:
...         counter += 1
...     return counter
```

```
>>> # Hacemos dir() de nuevo
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'count']
>>> # Y ahora aparece nuestra nueva función count en la lista, ya que ahora está dentro del scope, lo mismo pasará si
>>> mi_lista = [1, 2, 3, 4, 5, 6, 7]
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'count', 'mi_lista']
>>>
>>>
>>> # ¿Qué pasa si le pasamos un parámetro a dir()?
>>> dir(mi_lista)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__', '__format__', '
>>> # Nos enlistó todos los métodos y atributos que podemos usar con nuestra lista.
```

divmod(dividendo, divisor)

Divmod regresa una tupla cuyo primer miembro es el resultado truncado de la división de ambos parámetros y el segundo es el módulo de ambos.

```
>>> divmod(30, 3)
(10, 0)
>>> divmod(18.5, 4.68)
(3.0, 4.460000000000001)
```

eval(expresión)

Eval recibe una expresión o cadena y se evalua como una expresión de Python

```
>>> x = 15
>>> eval('x ** 2')
225
```

exec(expresión)

Exec recibe una orden de Python y la ejecuta, es útil para la ejecución dinámica de scripts

```
>>> exec('print("Soy una cadena impresa dinámicamente :D")')
"Soy una cadena impresa dinámicamente :D"
```

hex(num)

Hex recibe un número decimal y regresa su equivalente en código hexadecimal junto con el prefijo '0x'

```
>>> hex(16)
'0x10'
>>> hex(151651)
'0x25063'
```

id(objeto)

Id recibe un objeto y regresa la unidad de dirección de memoria de dicho objeto

```
>>> id(15)
94196848721984
>>> id([1, 2, 3]])
140704700168904
```

```
>>> id(True)
94196848546272
```

Este valor es único de cada objeto.

input(prompt)

Input recibe una cadena que mostrará en la consola y regresa el parámetro que el usuario escriba como una cadena

```
>>> input('Escribe un valor: ')
Escribe un valor: 15
'15'
```

int(objeto)

Int recibe un objeto y lo castea a un entero siempre y cuando sea compatible

```
>>> int('15151')
15151
>>> int(16.845)
16
```

len(objeto)

Len recibe un objeto como parámetro y regresa el número de elementos en él, el objeto debe ser una secuencia, una lista, tupla, un range, o una colección como un diccionario, set o frozenset

```
>>> len('Sígueme en Twitter :3')
21
>>> len(['uno', 'dos', 'tres])
3
>>> len(((1, 2, 3), 'otro elemento', 16))
3
```

En fin, encontraremos un sin fin funciones builtin de Python que podrán ayudarnos en cualquier momento, ya sea importando el módulo o simplemente usándolas como lo hemos hecho hasta ahora.

Decoradores

¿Qué es un decorador?

Un decorador es una función que recibe como parámetro una función y altera el comportamiento de dicha función, retornando otra función. Para crear un decorador podemos hacerlo de la siguiente manera:

```
def agujero_negro(func): #Un decorador que hará que la función que reciba regrese None
    def hacerla_none():
        pass
    return None
```

Listo, ahora tenemos nuestro constructor, pero para implementarlo habrá que hacer uso de cierta estructura, que es la siguiente:

```
@agujero_negro
def suma(a, b):
    return a + b
```

Una vez que hemos hecho esto, nustra función `suma` ahora en vez de regresar la suma de ambos números, regresará un None. Este ejemplo es muy sencillo, sin embargo es una forma de entender el concepto de Decorador, que no es más que un **más que un envoltorio de una función** por lo tanto, debe retornar una función. Chequemos otro pequeño ejemplo, un poco más coomplejo para intentar que el concepto quede más claro.

```
def html(func):
    contador = 0
    def box(*args, **kwargs):
        return "%s" % (func(*args, **kwargs))
    return box

@html
def saludo(nombre, visitas):
    return "Hola %s eres el visitante numero %d!" % (nombre, visitas)

print(saludo("Aldo", 500))
```

Ahora, hagamos un ejemplo con un logger para ver que parámetros le hemos pasado a una función:

```
def logger(func):
    def box(*args, **kwargs):
        print("los argumentos son: %s, %s" % (args, kwargs))
        result = func(*args, **kwargs)
        return result
    return box

@logger
def suma(a, b):
    return a+b

@logger
def resta(a, b):
    return a-b

print(suma(3,2))
print(resta(3,2))
```

Listas por comprensión

Formas de crear listas

Una lista por comprensión es una forma mucho más rápida y eficiente de llenar una lista, es decir, supongamos que tenemos que llenar una ista de números del 1 al 10000, ¿Recuerdas lo que hicimos con los generadores? Pues será muy similar, por ejemplo tenemos varias formas de hacer una lista del 1 al 10000, la más difícil, larga e ineficiente es llenarla a mano:

```
x = [1, 2, 3, 4, 5, 6, ..., 100000]
```

¿Imaginas el tiempo que tardaríamos en eso? Aún así hay otras cuantas formas de hacerlo, por ejemplo:

```
x = []
i = 0
while i <= 10001:
    x.append(i)
    i += 1
```

Esta forma, sí, en efecto será mejor que la anterior, pero no es la más eficiente, podemos también crear un generador, o sólo crear una llamada **lista por comprensión**

```
x = [i for i in range(1, 10001)]
```

Y listo, tenemos nuestra lista creada, en una forma mucho más rápida y eficiente, sin mencionar que sale en una sola línea de código ¿Qué pasa si necesitamos una lista con todos los números primos del 1 al 10000?, creemos unas cuantas fucniones para lograrlo

```
def es_primo(numero):
    for i in range(2,numero):
        if (numero%i)==0:
            return False
    return True
# Ya hemos creado una función que regresa True en caso de que el número sea primo y False en caso contrario
# Ahora, agreguémoslo a una lista por comprensión
```

Tanto en las listas por comprensión como en los generadores se pueden agregar condicionales, con una sintaxis igual:

```
primos = [i for i in range(1, 10001) if es_primo(i) == True]
```

Esta es una de las tantas ventajas de las listas por comprensión, el primer elemento será el elemento a agregar en la lista, el ciclo for delimitará el rango y los valores que tomará el iterador, y el condicional será quien se encargue de verificar si el valor se agrega a la lista o no, veamos, supongamos que tenemos una lista de números del 40000 al 50000, dichos números, deben ser insertados en una lista, pero en forma de cadena, no de entero, y encima sólo si los números son pares, además de que en caso de ser impares, se debe sustituir con un 1, solucionemos esto, con una lista de comprensión. En este caso, al agregar un nuevo condicional, tendremos que modificar un poco la sintaxis:

```
def numeros_a_cadena(lista_de_numeros):
    lista_de_cadenas = [str(i) if i % 2 == 0 else str(1) for i in lista_de_numeros]
```

Donde `str(i)` será el valor que se irá agregando a nuestra lista, `if i % 2 == 0 else str(1)` controará si se agrega el número o si se agrega un '1' y `for i in lista_de_numeros` serán nuestras iteraciones en una lista, estás listas de comprensión ahorran tiempo tanto al desarrollador del proyecto como a quien deba utilizar el código, ya que su sintáxis hace más sencilla la lectura del algoritmo.

Lambdas

###Funciones anónimas Como hemos podido ver, Python es fan de hacer código en una sola línea, para serle fiel al Zen de Python, por lo tanto, admite una sintaxis interesante llamada Lambda, inspirada en lenguajes como LISP, que pueden usarse en cualquier lado donde se necesite una función. Hasta ahora, hemos visto las funciones de esta forma:

```
def cuadrado(x):
    return x ** 2

f = cuadrado(8)
# f = 64
```

Sin embargo, una forma más rápida y reusumida de declararla sería así:

```
cuadrado = lambda x: x ** 2
f = cuadrado(3)
f = 9
```

O también de la siguiente forma si lo que se desea es ahorrar líneas de código:

```
f = (lambda x: x ** 2)(3)
# f = 3
```

Es debido a esta última sintáxis que se les conoce a las funciones lambda como funciones anónimas, ya que no cuentan con un nombre o identificador como las que habíamos visto hasta ahora. Comparemos la sintáxis con la anterior que habíamos visto para definir funciones `lambda` será el equivalente en cierta forma de `def` que será la palabra reservada que nos permitirá comenzar a definir una función. `x:` sería al equivalente de los parámetros `(x)` , eso quiere decir, que si nuestra funcioón lambda usará más de un parámetro, se puede hacer, de la siguiente forma: `x, y:` que sería equivalente a `(x, y)` Y por último `x ** 2` será el código ejecutable dentro de la función, ya incluyendo el return que finalizará la función y retornará un valor.

Para aplicar este tipo de funciones sobre una lista u objeto iterable hagamos un ejemplo, supongamos que teneos una tupla con una cantidad de temperaturas en escala de Celsius, y tenemos la necesidad de pasarlas a escala de Farenheit.

```
celsius = (39.2, 36.5, 37.3, 38, 37.8)

Farenheits = map(lambda x: (float(9)/5 * x + 32), celsius)
```

Con la función map() se hace un objeto iterable del tipo map, este objeto se llenará iterando sobre el segundo parámetro, en este caso `celsius` y sustituyendo el parámetro de x de lambda con esos valores.

Programación orientada a objetos

La Programación Orientada a Objetos es otro de los paradigmas que podemos encontrar en el lenguaje Python. Este tipo de paradigma hace una abstracción del mundo real, lo que nos permite llevar la lógica de nuestro programa de una manera distinta en como lo hacíamos en Programación estructurada. Este paradigma está basado en varias técnicas como lo es el *encapsulamiento*, la *herencia*, el *polimorfismo* y la *abstracción*, las cuales también conocemos como los **Pilares de la programación orientada a objetos**.

Para poder adentrarnos dentro de este paradigma, primero debemos definir varios de los conceptos que estaremos utilizando a lo largo de esta sección. Estos conceptos son los siguientes:

- Clase: Es una plantilla o "molde" para la creación de objetos que cuentan con características similares. En ella se definen los atributos (características o estado) y los métodos (acciones) de los objetos.
- Objeto: Es una instancia de una clase. Se refiere a una entidad que tiene datos particulares (atributos) y tiene formas para operar sobre ellos (métodos). Al igual que en la vida real, un objeto puede interactuar con otro mediante sus métodos
- Atributos: Son aquellas características individuales que ayudan a diferenciar un objeto de otro.
- Método: Son las acciones que puede realizar nuestro objeto. Estos determinan cómo tienen que actuar y también pueden enviar mensajes a otros objetos solicitando una acción o información.

Ya que conocemos a qué se refiere cada uno de estos conceptos, podemos pasar a la parte de codificación en Python.

Clases y objetos

Hemos mencionado con anterioridad que al hablar de una clase, hacemos referencia a una plantilla o a un molde, entonces, imaginemos que todas las personas en un salón de clases provenimos de una clase llamada *Persona*. Con lo anterior estamos diciendo que todas las personas dentro del salón, tendrán características y acciones similares. Para tener más claro esta analogía, pasémosla a código en Python. Para crear una clase en Python, tendremos que hacer uso de la palabra *class* seguido del nombre de la clase, por ejemplo, véase el siguiente código:

```
class Persona:
        nombre = "Rodrigo"
        edad = 21
        genero = "masculino"

        def saludar(self):
                print("Hola a todos")
```

En el código anterior estamos diciendo que tenemos una clase Persona la cual tiene tres atributos: *nombre*, *edad* y *genero*; además también tiene un método el cual se llama *saludar()* que únicamente imprime en pantalla un saludo. Hay varias cosas que debemos de aclarar antes continuar...

- En primer lugar, notarás que el nombre de la clase se ha escrito con la primer letra mayúscula. En realidad esto no es completamente necesario, cualquiera puede crear una clase con un nombre escrito en minúsculas y no obtendremos error alguno, sin embargo, las reglas de codificación (leer el apartado de introducción a Python) nos dicen que los nombres de clases deben iniciar siempre con mayúsculas. Lo anterior también ayuda a los programadores a identificar a una clase de alguna otra función cualquiera.
- Los atributos son muy fáciles de identificar, se encuentran inmediatamente al iniciar el bloque de identación de la clase, y es como si se declarara una variable como lo hemos hecho en lecciones anteriores, sólo le asignamos un valor y eso es todo.
- Por último, vemos que tenemos los métodos y si observamos bien, es la misma estructura que se utilizó en programación funcional, sin embargo, hay algo nuevo... Vemos que ahora, en los parámetros que recibe, tiene una palabra que dice *self*. Esta palabra SIEMPRE debemos incluirla en nuestros métodos, ya que es la que nos va a ayudar a diferenciar un objeto de otro. Como recordaremos las funciones podíamos mandarlas a llamar únicamente escribiendo su nombre y los paréntesis, pero con la palabra *self* le indicaremos que se utilizará mediante un objeto que nosotros creemos. Es importante mencionar que *self* como tal no es una palabra reservada, lo que quiere decir que, en vez de *self* podríamos poner otro nombre como *hola* y no habría ningún problema, sin embargo, *self* se utiliza como un estándar, para saber qué es a lo que nos estamos refiriendo.

Partiendo del código anterior, entonces, ¿cómo creo un objeto y hago uso de sus atributos y métodos? Para eso veamos la siguiente parte del código:

```
persona1 = Persona() #Creación del objeto

#Accediendo a los atributos
print("El nombre de la persona 1 es: ", persona1.nombre)
print("La edad de la persona 1 es: ", persona1.edad)
print("El genero de la persona 1 es: ", persona1.genero)

#Ejecutando el método de la clase
persona1.saludar()
```

Para la creación de un objeto, lo único que tenemos que tener una variable cualquiera (en este caso *persona1*) y con el operador "=" le indicamos que será un objeto de la clase Persona(). Para acceder a los atributos y métodos lo hacemos mediante el operador punto ("."), pero... ¿Se observa el problema? Si nosotros seguimos creando más objetos notaremos que ¡todos tienen los mismos valores en sus atributos! Para arreglar esto, podemos hacer uso de un método especial llamado ***init()***. Este método nos servirá para que, al momento de crear nuestro objeto, podamos indicarle los valores de sus atributos. A este método podemos llamarle también constructor.

> El método **init**() es uno de los llamados métodos mágicos, sin embargo, esto se estudiará un poco más adelante.

El método init() (Constructor)

Veamos un ejemplo de la implementación del método **init**():

```
class Alumno: # Clase
```

```
        def __init__(self, nombre, apellido, calificaciones): # Constructor inicializa datos al momento de crear un ob
                self.nombre = nombre
                self.apellido = apellido
                self.calificaciones = calificaciones

        def mostrarNombre(self):
                print("Mi nombre es: " + self.nombre + " " + self.apellido)

        def calificacionFinal(self):
                suma = 0
                for valor in self.calificaciones:
                        suma += valor
                promedio = suma/len(self.calificaciones)
                return promedio
# Creacion de objetos - Crear instancias

alumno1 = Alumno("Rodrigo", "Vivas", [6,9,6])
alumno2 = Alumno("Ricardo", "Hernández", [8,7,10,2])

# Mandar a llamar atributos o metodos, se hace con el operador "."

print(alumno1.nombre)
print(alumno1.apellido)
alumno1.mostrarNombre()
print("Calif final: ", alumno1.calificacionFinal())

print(alumno2.nombre)
print(alumno2.apellido)
alumno2.mostrarNombre()
print("Calif final: ", alumno2.calificacionFinal())
```

Vemos que con el método **init**() no declaramos los atributos inmediatamente después de la creación de la clase, sino que los estamos declarando dentro del método en sí.Aquí pueden surgir varias preguntas, como por ejemplo, *¿A qué se refiere eso de* `self.nombre = nombre` *?*. Veamos, al estar trabajando en este paradigma y dentro de nuestras clases más específicamente, podemos llegar a confundirnos entre cuáles son los atributos y cuales son variables externas a la clase. Para solucionar lo anterior, vamos a identificar a todos los atributos de la clase utilizando la palabra *self* antes de su nombre, por lo que *self.nombre* está haciendo referencia a que es un atributo de la clase.

Ya que quedó claro cuáles son los atributos de la clase, ahora observemos que **init**() recibe diferentes parámetros: uno es el que siempre debemos de usar (*self*) y los otros son los parámetros que va a recibir cuando nosotros creemos el objeto. Habiendo dicho esto, no queda más que explicar que, cuando nosotros tenemos una línea de código como `self.nombre = nombre` le estamos diciendo a nuestro programa que a nuestro atributo de la clase (*self.nombre*) le vamos a asignar un valor que le pase el usuario (`nombre`).

Lo anterior mencionado sucede cuando nosotros implementamos las siguientes líneas de código:

```
alumno1 = Alumno("Rodrigo", "Vivas", [6,9,6])
alumno2 = Alumno("Ricardo", "Hernández", [8,7,10,2])
```

*¿Se le pueden poner nombres diferentes a los parámetros del método **init**() para distinguirlos todavía mejor de los atributos?* ¡Claro que se puede! Sin embargo, muchas veces es mejor mantener el mismo nombre. Imaginemos la situación en que nosotros tenemos 10 atributos y va a recibir también 10 valores dentro del método... Si pasamos el código a otro programador (o incluso si nosotros dejamos el programa por un tiempo y después volvemos) puede causar confusión el tener tantas variables, por lo que si utilizamos el mismo nombre, puede ser mucho más fácil identificar qué parámetro le corresponde a "X" atributo.

Métodos de clase y estáticos

Cuando nosotros declaramos métodos dentro de una clase, para hacer uso de ellos debemos crear una instancia de una clase, es decir, un objeto mediante el cuál podemos mandarlos a llamar; por esta razón esos métodos son denominados "Métodos de instancia". *¿Qué*

hago si yo no quiero crear un objeto para llamar a un método? Esto es posible mediante los llamados métodos de clase y métodos estáticos. Sus definiciones son las siguientes:

- *Métodos de clase*: Son aquellos que no necesitan de una instancia para ser llamados y además pueden acceder a los atributos de la clase.
- *Métodos estáticos*: No necesitan instancias para ser llamados pero con la restricción de que no pueden acceder a los atributos de la clase.

Veamos el caso de los métodos de clase con el siguiente código:

```
class Perro:
        def __init__(self, nombre):
                self.nombre = nombre

        @classmethod
        def crear(cls, nombre):
                return Perro(nombre)

animal = Perro.crear("Lucho")
print(animal.nombre)
```

El método que haremos de clase será *crear()* el cual nos devuelve un objeto de tipo Perro y además nos pide el nombre de éste. Observa lo que tiene escrito antes de él... ¿Te recuerda a algo? Efectivamente, se trata de un decorador (`@classmethod`) con el que ya cuenta Python al momento de su instalación. Con tal decorador ya no estaremos haciendo referencia a una instancia cada que lo llamemos y en vez de eso, sólo hacemos referencia a la clase (por eso su primer parámetro es cls). Fíjate que en este método, además, está accediendo a un atributo de la clase, el cual es el nombre del perro.

Si este método no fuera de clase, tendríamos que hacer lo siguiente:

```
objeto = Perro("Leon")
animal = objeto.crear("Lucho)
```

Pero ya que es un método de clase, únicamente basta con poner el nombre de la clase y el método, así como se muestra a continuación

```
animal = Perro.crear("Lucho")
```

Incluso observa que al momento de usar `Perro.crear("Lucho")` no se ha creado un objeto como tal.

Si fuera el caso en que yo quiero crear un método, al cual yo quiero acceder mediante la clase y además, que pueda acceder a mis atributos, tendría que crear un *método estático*. Para hacerlo, también debo de utilizar un decorador, pero ahora será `@staticmethod` , así como se observa en el siguiente código:

```
class Joven:

        def __init__(self, nombre):
                self.nombre = nombre

        @staticmethod
        def saludar():
                return "hola"

print(Joven.saludar())
```

En este caso ya no pondremos ningún parámetro en el método saludar (el cual es estático) y únicamente podremos llamarlo con el nombre de su clase y del método, así como se muestra en la última impresión de pantalla.

Métodos mágicos

Hasta el momento hemos trabajado con métodos como len(), print(), entre muchos otros. ¿Cuál es el gran problema? Si recordarán, muchos de esos métodos sólo sirven con ciertos tipos de datos. ¿Qué tengo que hacer si a esos métodos les quiero agregar más funcionalidad? Para eso existen los métodos mágicos, los cuales son métodos que nos van a permitir alterar comportamientos que parecían implícitos, estos métodos ya vienen definidos en Python. Todo en Python es un objeto (Se estudiará esta parte mejor en el tema de Herencia) por lo que todos nuestros tipos de datos provienen de una clase como tal, y muchas de esas clases implementan métodos mágicos para definir algún tipo de comportamiento.

Veamos, por ejemplo, la siguiente clase:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento

p = Pelicula("El Padrino", 175, 1972)
```

¿Qué pasaría si intento ejecutar la siguiente instrucción?

```
print(p)
```

Lo único que nos devolvería ese código sería lo siguiente:

```
<__main__.Pelicula object at 0x01819490>
```

¿Cómo le digo a mi programa que quiero que me imprima un mensaje cuando pase un objeto de una clase "X" a un print? Para eso justamente nos servirán los métodos mágicos. Para el caso de la función print() (que nos recibe una cadena), tendremos que añadirle funcionalidad mediante el método **str**(). Veamos su implementación:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento
                print("Se ha creado la pelicula", self.titulo)

        #Redefinimos el método string
        def __str__(self):
                return "{0} lanzada en {1} con una duración de {2} minutos".format(self.titulo, self.lanzamiento, self

p = Pelicula("El Padrino", 175, 1972)

print(p)
```

Vemos que el método **str**() lo que re define es al método str() (El cual se vio en la lección de cadenas) Esto quiere decir que si nuestro objeto lo intentamos utilizar como si fuera una cadena, devolverá el mensaje que nosotros le implementemos. Lo que devuelve el código en este caso sería:

```
El Padrino lanzada en 1972 con una duración de 175 minutos
```

Ya que se ha explicado el método **srt**(), veamos ahora otro método mágico. Veamos qué es lo que pasa ahora si ponemos el siguiente código:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento

p = Pelicula("El Padrino", 175, 1972)

print(len(p))
```

Si nosotros intentamos ejecutar el siguiente código, nos debe salir el siguiente error:

```
TypeError: object of type 'Pelicula' has no len()
```

Recordemos que el método len() podíamos utilizarlo con listas, porque como tal, las listas sí tiene una longitud, es decir, una cantidad de elementos. Pero para el caso de un objeto... ¿Cuál será su longitud? Nosotros como humanos claramente podríamos decir: "Obviamente quiero la duración de la película", pero recordemos que a la hora de programar, nosotros tenemos que ser muy explícitos con nuestra computadora. Entonces, si se quiere utilizar el método len(), tendremos que implementar en nuestra clase el método mágico **len**(). Veamos el siguiente código para ver cómo se hace:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento

        #Re definimos el método len()
        def __len__(self):
                return self.duracion

p = Pelicula("El Padrino", 175, 1972)

print("La duración de la película es: ",len(p))
```

Observa que la implementación del método len() dentro de la clase, únicamente nos retorna la duración, y es justamente lo que queremos. Si ejecutamos el programa, ahora sí nos debería dar la siguiente salida:

```
La duración de la película es: 175
```

Veamos entonces un último método mágico. Hemos visto que el método **init**() nos funciona para poder inicializar a un objeto, dándole valores a sus atributos. En Python también existe su método opuesto, es decir, un destructor de objetos. Este método destructor, SIEMPRE es llamado al finalizar nuestro programa, sólo que nosotros no podemos verlo porque no nos manda ningún mensaje. Este método mágico es el llamado **del**(), así que vamos a añadirle un poco más de funcionalidad. Observa el siguiente código:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento

        #También existe un método destructor que se ejecuta al momento de eliminar una instancia
        # Destructor de clase

        def __del__(self):
                print("Se está borrando la película ",self.titulo)

p = Pelicula("El Padrino", 175, 1972)
```

En la implementación del método destructor, únicamente le pusimos una impresión de pantalla que nos diga cuál es la película que se está borrando. Ejecutando este código se obtiene lo siguiente:

```
Se está borrando la película El Padrino
```

En ningún momento se ejecutó un método para poder eliminar al objeto, por lo que se demuestra que el objeto se destruye al finalizar el programa. Si nosotros quisiéramos eliminar al objeto en otra parte de nuestro código, basta con utilizar la función del(), así como se muestra en el siguiente código:

```
class Pelicula:
        def __init__(self, titulo, duracion, lanzamiento):
                self.titulo = titulo
                self.duracion = duracion
                self.lanzamiento = lanzamiento

        #También existe un método destructor que se ejecuta al momento de eliminar una instancia
        # Destructor de clase

        def __del__(self):
                print("Se está borrando la película ",self.titulo)


p = Pelicula("El Padrino", 175, 1972)
del(p)
print("Todavía no termina el programa, pero se borró el objeto")
```

Y con este método obtenemos el mismo resultado, pero antes de finalizar el programa:

```
Se está borrando la película El Padrino
Todavía no termina el programa, pero se borró el objeto
```

Existen muchos más métodos mágicos que nosotros podemos implementar, sin embargo, tomaría un buen rato explicarlos por aquí. Dentro de esta liga encontrarás otros de los métodos mágicos que existen en Python.

Herencia

Cuando se habla de herencia, quizá recordemos nuestras clases de Biología y pase por nuestra mente el hecho de tener características similares a nuestros padres. En programación, este concepto no cambia del todo.

Si se habla de Herencia dentro de la programación, se dice que una clase puede heredar tanto sus atributos, como sus métodos de una clase padre. Esto nos puede servir porque habrán muchos objetos que realicen acciones muy parecidas a los objetos de otra clase, así que para evitarnos la repetición de código, es mejor que los métodos y atributos se vayan heredando.

Imaginemos pues, una clase llamada Animal. Un animal puede hacer una acción como lo es comer o correr, y eso aplica para cualquier animal. Entonces si nosotros creamos una clase más específica, como lo es la clase Perro, estamos de acuerdo que el perro también puede comer y correr, más aparte realizar otra cantidad de acciones. Habiendo dicho lo anterior, vamos a hacer que el Perro herede los métodos y atributos de la clase animal. En Python, esto podemos lograrlo de la siguiente manera:

```
class Animal:
        color = ""
        nombre = ""

        def __init__(self, nombre, color):
                self.nombre = nombre
                self.color = color
```

```
        def correr(self):
                print("El animal está corriendo y es de color",self.color)

        def comer(self):
                print(self.nombre, "esta comiendo")

class Perro(Animal):
        def ladrar(self):
                print("Wooof!")

perro = Perro("Lucho", "blanco")
perro.comer()
perro.correr()
perro.ladrar()
```

Para heredar entonces de una clase, bastará con que nosotros creemos nuestra nueva clase, y al lado del nombre de esta, abrir paréntesis y escribir la clase de la cuál va a tomar atributos y métodos. En este caso, la herencia se da en esta línea:

```
class Perro(Animal):
```

Además, observa que el objeto perro que se creo, está utilizando los métodos comer y correr, sin que estén definidos como tal en la clase, al igual que los atributos de nombre y color que está en el constructor padre. Esto es porque al momento de heredar, también puede utilizar su método constructor y los métodos definidos. La salida de este código sería:

```
Lucho esta comiendo
El animal está corriendo y es de color blanco
Wooof!
```

¿Qué pasaría si nosotros tuviéramos constructores tanto en la clase padre como en la clase hija? Veamos, por ejemplo, el siguiente código:

```
class Figura:
        def __init__(self, nombre):
                self.nombre = nombre

class Cuadrado(Figura):
        def __init__(self,nombre, lados):
                super().__init__(nombre)
                self.lados = lados

figuras = Cuadrado("Rodrigo",4)
print(figuras.nombre)
print(figuras.lados)
```

Si nosotros intentáramos crear un objeto con un sólo constructor, nos marcaría un error, porque no sabría cuál de ellos tomar y además puede marcar el error de que faltan argumentos. Para evitar problemas, en el constructor de la clase hija, debemos de hacer uso del método super(), que en este caso le diremos que vamos a heredar su método **init**() con su atributo nombre, y de esta manera, nosotros ya no tendremos que hacer una nueva asignación como lo es `self.nombre = nombre` . Hay que fijarse muy bien que al segundo constructor, también le pasamos de parámetro un nombre, pues es el valor que le vamos a asignar al constructor heredado.

Multiherencia

A diferencia de otros lenguajes de programación como Java o C#, Python soporta lo que se conoce como Multiherencia, esto quiere decir que una clase, puede heredar atributos o métodos, de dos o más clases. A pesar de que esto no sea tan utilizado en Python, debemos de saber que es posible.

Por ejemplo, veamos el caso de un Perro de nuevo, en donde ahora va a heredar los métodos de dos clases: Mascota y Canino.

```
class Mascota:
        def descansar(self):
                print("La mascota esta descansando")

class Canino:
        def ladrar(self):
                print("Woof!")

class Perro(Mascota,Canino):
        def __init__(self, nombre):
                self.nombre = nombre
                print("Soy un perro")

        def jugar(self):
                print("Wooooof! :P")

perrito = Perro("Duque")
perrito.descansar()
perrito.ladrar()
perrito.jugar()
```

La implementación es igual que se hizo con la herencia múltiple, sólo que ahora, para poder heredar de las dos clases se escribió la siguiente línea:

```
class Perro(Mascota,Canino):
```

Es decir, sólo se separan mediante comas (,) las clases de las cuales va a heredar. Si ejecutamos este código, obtendremos el siguiente resultado.

```
Soy un perro
La mascota esta descansando
Woof!
Wooooof! :P
```

El principio de operación es el mismo que el de herencia múltiple con la diferencia que aquí son más clases de las cuales heredar.

Polimorfismo

El origen de este término viene del griego: "poli-" -> Muchos, "morfo" -> Formas

El polimorfismo se refiere a la capacidad de los objetos de responder a una misma función de una manera diferente. Hagamos una analogía: Estamos de acuerdo que tanto un perro como un gato pueden emitir sonidos, sin embargo... *¿Lo hacen de la misma manera?* Evidentemente no, mientras que un gato maúlla (o hace "Miau"), el perro ladra (o hace "Woof"). Es decir que los dos podrían tener un métodos llamado igual que sea `emitir_sonido()` pero ambos objetos lo harán de manera diferente.

Veamos el ejemplo codificado:

```
class Gato:
        def emitir_sonido(self):
                print("MIAU!")

class Perro:
        def emitir_sonindo(self):
                print("WOOF!")

def sonido(animal):
        animal.emitir_sonido()
```

```
gato = Gato()
perro = Perro()

sonido(perro)
sonido(gato)
```

Para esto se crearon dos clases: Perro y Gato, cada una con su respectivo método emitir_sonido() que imprimen mensaje diferentes. Fuera de la creación de las clases, se crea una función llamada *sonido()* la cual recibe un objeto animal (es decir, de alguna de las dos clases) y ejecuta su método emitir_sonido().

Después de eso creamos un objeto para cada una de las clases y los pasamos al método sonido(). La salida que obtenemos es:

```
WOOF!
MIAU!
```

Verificando así, que los objetos se comportan de diferente manera, aún teniendo el mismo método.

Protección de atributos y métodos

Cuando hablamos de protección de atributos y métodos no nos referimos a la protección de un "ataque informático", sino más bien, a la protección de no poder acceder a ellos a través de otra clase. En otros lenguajes de programación (como Java), podremos encontrar los modificadores de acceso *public* y *private*.

Lo que hacen los modificadores de acceso determinan desde qué clase se puede acceder a un determinado elemento. El modificador de acceso *public* permite que tanto sus atributos, como sus métodos, puedan ser accedidos a través de otra clase cualquiera. El modificador de acceso *private* únicamente permite el acceso a atributos y métodos dentro de la MISMA clase.

En Python no se cuenta con los modificadores de acceso como tal, sin embargo, podemos simular ese comportamiento añadiendo dos guiones bajos (__) antes del nombre del método o del atributo. Veamos un ejemplo codificado:

```
class Ejemplo:
        __atributo_privado ="Soy un atributo inalcanzable"

e = Ejemplo()

print(e.__atributo_privado)
```

Nuestros conocimientos de Python nos dicen que este código no tendría ningún error, porque es la manera correcta de definir, clases, atributos, objetos e incluso es la manera correcta de acceder a ellos. Pero si lo ejecutamos obtenemos lo siguiente:

```
AttributeError: 'Ejemplo' object has no attribute '__atributo_privado'
```

Vemos que como tal, no nos dice que es un atributo inaccesible fuera de la clase, sino que nos marca un error de que no existe. Por eso en Python se dice que es únicamente una simulación.

Si quisiéramos acceder a este atributo, tendríamos que hacerlo mediante un método que no tuviera los dos guiones bajos, es decir, un método *público*:

```
class Ejemplo:
        __atributo_privado ="Soy un atributo inalcanzable"

        def atributo_publico(self):
                return self.__atributo_privado

e = Ejemplo()
```

```
#print(e.__atributo_privado)
print(e.atributo_publico())
```

De esta manera, ya nos devuelve lo siguiente:

```
Soy un atributo inalcanzable
```

Lo mismo pasa con los métodos, si lo precedemos con dos guiones bajos, no podremos acceder a ellos, a menos que sea mediante otro método:

```
class Ejemplo:
        def __metodo_privado(self):
                print("Soy un método inalcanzable desde fuera")

        def metodo_publico(self):
                return self.__metodo_privado()

e = Ejemplo()

#e.__metodo_privado()
e.metodo_publico()
```

Si intentamos ejecutar directamente el método privado, nos mandaría el mismo error que obtuvimos anteriormente.

Esto puede llegar a ser de utilidad si es que no queremos que se modifique como tal un atributo o método de una clase, a menos que sea a través de un método. Imaginemos que le damos nuestro código a otro programador y no sabe que NO debe de modificar un atributo, y al estar haciendo varias operaciones con él, sin quererlo, alteró su valor. Con los métodos damos la certeza de que, efectivamente, se quiere modificar el valor, dándole así un poco más de seguridad a nuestra información dentro del código.

Programación modular, paquetes y documentación

Módulos

Existen varios métodos para crear módulos pero la forma más sencilla es crear un archivo con extensión `.py` que contenga un conjunto de funciones y/o de clases. Un módulo puede ser importado por otro programa para hacer uso de su funcionalidad, de igual forma que se hace con aquellos módulos que pertenecen a la biblioteca estándar de Python.

La carga de un módulo compilado es más rápida. Los archivos compilados tienen la extensión `.pyc` y tienen la ventaja de que son independientes de la plataforma.

Ejemplo de uso del modulo `math` que ofrece funciones específicas para operaciones matemáticas:

```
>>> import math # Importamos el módulo math
>>> a = math.pow(2, 3) # pow(), potencia de un número
>>> print(a)
8.0
>>> b = math.sin(math.pi/2) # sin(), calcula el seno trigonométrico, el parámetro es en radianes
>>> print(b)
1.0
>>> c = math.sqrt(144) # sqrt(), calcula la raíz cuadrada de un número
>>> print(c)
12.0
>>> print(math.pi) # atributo pi, constante pi con valor 3.141592...
3.141592653589793
```

> Nota: Para importar usamos la palabra reservada `import`

Para importar módulos hay varias formas.

1. Importar todo el módulo

```
>>> import math
```

Para acceder a sus métodos o atributos, tenemos que poner el nombre del módulo, luego un . y al final el método

```
>>> potencia = math.pow(1,0)
>>> print(potencia)
1.0
```

2. Importar los métodos o funciones que vayamos a ocupar, esto lo hacemos con las palabras reservadas `from` e `import`

```
>>> from math import pow
>>> potencia = pow(2,4)
>>> print(potencia)
16.0
```

Para utilizar esos métodos, basta con llamarlos como cualquier otra función. Esto no es muy recomendable hacerlo, ya que al momento de importar varios módulos pueda que un método se llame igual a otro y entre en conflicto.

Podemos importar más de una función o método. Solo basta con separarlos por comas:

```
>>> from math import pow,sqrt
```

3. Importar todos los métodos, esto lo hacemos con un asterisco `*`. La diferencia con importar todo el módulo, es que en este caso, para utilizar los métodos, no es necesario anteponer el nombre del módulo.

```
>>> from math import *
```

Namespaces y Alias

Cuando se hace referencia a algún elemento que pertenece a un módulo importado se indica previamente su espacio de nombres (namespaces) seguido de un punto “.” como en el ejemplo anterior: math.pow.

Hay veces que los nombres de módulos y/o funciones están muy largo, python nos ofrece la opción de acortar esos nombres dándoles un `alias`. Para hacer eso usamos la palabra reservada `as`.

```
>>> from math import factorial as fact
>>> print(fact(3)) # Imprime el factorial de 3
6
```

Al usar el método, ya solo tenemos que usarlo conforme al alias que le pusimos.

Creación y uso de un módulo

En el siguiente ejemplo se crea el módulo `moduloseries.py` y un programa `programa.py` lo importa y hace uso de la función sumadesde1aN (que para un número n, suma todo los números que van desde 1 a n).

moduloseries.py:

```
def sumadesde1aN(pnumero):
    ''' Suma todos los números consecutivos desde 1 hasta
    el número expresado. Si el valor es menor que 1 devuelve 0 '''
    ntotal= 0
    if pnumero >0:
        for nnum in range(1,pnumero+1):
            ntotal = ntotal + nnum

    return ntotal

__version__ = '1.0'
```

programa.py:

```
import moduloseries

valor = 10
print('Suma desde 1 a '+str(valor)+':',
      moduloseries.sumadesde1aN(valor))
print('Versión:', moduloseries.__version__)
print('Nombre:', moduloseries.__name__)
print('Doc:', moduloseries.sumadesde1aN.__doc__)
```

También sería posible importar la función y la versión del módulo, así:

```
from moduloseries import sumadesde1aN, __version__

# Importa las funciones pero no la __version__
from moduloseries import *
```

Paquetes

Las variables suelen ir dentro de las funciones, las llamadas variables globales son las que están dentro de los módulos. Los módulos suelen organizarse en un tipo de carpetas especiales que se llaman paquetes. Dentro de estas carpetas deben existir necesariamente un archivo llamado **init**.py, aunque esté vacío. No es obligatorio que todos los módulos pertenezcan a un paquete.

Un paquete puede contener a otros subpaquetes y éstos, también, módulos. Cuando se importa un módulo es posible indicar su espacio de nombres:

```
# Importa módulo no empaquetado:
import modulo

# Importa módulo del paquete indicado:
import paquete.modulo

# importa módulo del subpaquete/paquete:
import paquete.subpaquete.modulo
```

Documentación con Pydoc

Docstrings

Las cadenas de documentación o `docstrings` son textos que se escriben entre triples comillas dentro de los programas para documentarlos. Cuando se desarrolla un proyecto donde colaboran varias personas contar con información clara y precisa que facilite la comprensión del código es imprescindible y beneficia a todos los participantes y al propio proyecto.

Las funciones, clases y módulos deben ir convenientemente documentados. La información de las docstrings estará disponible cuando se edite el código y, también, durante la ejecución de los programas:

```
def area_trapecio(BaseMayor, BaseMenor, Altura):
    '''area_trapecio: Calcula el área de un trapecio.
    area_trapecio = (BaseMayor + BaseMenor) * Altura / 2'''
    return (BaseMayor + BaseMenor) * Altura / 2

print(area_trapecio(10,4,4))  # Resultado: 28
print(area_trapecio.__doc__)
```

La sentencia print(area_trapecio.**doc**) muestra la información de la docstrings:

area_trapecio: Calcula el área de un trapecio. area_trapecio=(BaseMayor+BaseMenor)*Altura/2

Para recuperar la información de las docstrings en el modo interactivo utilizaremos la función help(); y en la línea de comandos haremos uso del comando pydoc3:

```
>>>help(math) -> Desde el interprete de python
```

```
$ pydoc3 math -> Desde la línea de comandos
```

Es frecuente agrupar varias funciones en un mismo archivo (módulo) acompañadas con información escrita suficiente que explique cómo utilizarlas, para que pueda ser consultada en cualquier momento por los programadores.

En el siguiente archivo (geometriaplana.py) se incluyen varias funciones para calcular el área de algunas figuras geométricas; de forma que puedan utilizarse en cualquier programa que escribamos con posterioridad, Las funciones incluyen documentación que explica brevemente para qué sirven y especifican los argumentos que utilizan:

geometriaplana.py:

```
'''Funciones de geometría plana
   Para el cálculo del área de las siguientes figuras:

'''

from math import pi

def area_cuadrado(Lado):
    '''Función area_cuadrado: Calcula área de un cuadrado.
    area_cuadrado = Lado ** 2 '''
    return (Lado ** 2)

def area_rectangulo(Base, Altura):
    '''Función area_rectangulo: Calcula área de un rectángulo.
    area_rectangulo = Base * Altura '''
    return (Base * Altura)

def area_triangulo(Base, Altura):
    '''Función area_triangulo: Calcula área de un triángulo.
    area_triangulo = Base * Altura / 2 '''
    return (Base * Altura / 2)

def area_paralelogramo(Base, Altura):
    '''Función area_paralelogramo: Calcula área de un paralelogramo.
    area_paralelogramo = Base * Altura '''
    return (Base * Altura)

def area_trapecio(BaseMayor, BaseMenor, Altura):
    '''Función area_trapecio: Calcula área de un trapecio.
    area_trapecio = (BaseMayor + BaseMenor) * Altura / 2'''
```

```
    return (BaseMayor + BaseMenor) * Altura / 2

def area_circulo(Radio):
    '''Función area_circulo: Calcula área de un circulo.
    area_ciruculo = Pi * Radio ** 2 '''
    return (pi * Radio ** 2)
```

Ver la documentación en la consola

Para ver la documentación de un módulo hecho por nosotros, podemos ir a la carpeta en donde se encuentre nuestro archivo y posteriormente poner el comando `pydoc3` y el nombre del archivo **sin extensión py**.

```
$ pydoc3 geometriaplana
```

Guardar la documentación en formato HTML

Para generar un archivo con extensión `.html`, el comando `pydoc` tiene una opción la cual es `-w`. La opción `-w` hará que se cree un archivo con la documentación en formato HTML.

```
$ pydoc3 -w geometriaplana
```