

1 Definición de variables.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#) 

1.1 Variables.

- Son **símbolos** que permiten identificar la información que se almacena en la memoria de la computadora.
- Son **nombres** o **etiquetas** para los objetos que se crean en Python.
- Se crean con ayuda del operador de asignación `=`.
- No se tiene que establecer explícitamente el tipo de dato de la variable, pues esto se realiza de manera dinámica (tipado dinámico).

1.1.1 Ejemplos de variables válidas.

Los nombres de las variables: * pueden contener **letras**, **números** y **guiones bajos**, * deben comenzar con una letra o un guion bajo, * se distingue entre mayúsculas y minúsculas, es decir, `variable` y `Variable` son nombres diferentes.

A continuación se muestran algunos ejemplos.

```
_luis = "Luis Miguel de la Cruz"    # El nombre de la variable es _luis, el conte
LuisXV = "Louis Michel de la Croix"
luigi = 25
luis_b = 0b01110 # Binario
luis_o = 0o12376 # Octal
luis_h = 0x12323 # Hexadecimal

# Sensibilidad a mayúsculas y minúsculas
# los siguientes nombres son diferentes
pi = 3.14
PI = 31416e-4
Pi = 3.141592
```

Podemos ver el contenido de la variable usando la función `print()`:

```
# El contenido de cada variable se imprime en renglones
# diferentes debido a que usamos el argumento sep='\n'
```

```
print(_luis, LuisXV, luigi, luis_b, luis_o, luis_h, pi, PI, Pi, sep='\n')
```

```
Luis Miguel de la Cruz
Louis Michel de la Croix
25
14
5374
74531
3.14
3.1416
3.141592
```

NOTA. Para saber más sobre la función `print()` revisa la sección XXX.

Para saber el tipo de objeto que se creo cuando se definieron las variables anteriores, podemos hacer uso de la función `type()`:

```
print(type(_luis), type(LuisXV), type(luigi),
      type(luis_b), type(luis_o), type(luis_h),
      type(pi), type(PI), type(Pi), sep = '\n')
```

```
<class 'str'>
<class 'str'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'int'>
<class 'float'>
<class 'float'>
<class 'float'>
```

También es posible usar la función `id()` para conocer el identificador en la memoria de cada objeto como sigue:

```
print(id(_luis), id(LuisXV), id(luigi),
      id(luis_b), id(luis_o), id(luis_h),
      id(pi), id(PI), id(Pi), sep = '\n')
```

```
139672517675408
139672517886160
94157725269672
94157725269320
139672517639248
139672517630576
139672517623920
139672517638992
139672518259056
```

Observa que cada objeto tiene un identificador diferente. Es posible que un objeto tenga más de un nombre, por ejemplo

```
luiggi = _luis
```

La etiqueta o variable `luiggi` hace referencia al mismo objeto que la variable `_luis`, y eso lo podemos comprobar usando la función `id()`:

```
print(id(luiggi))  
print(id(_luis))
```

```
139672517675408
```

```
139672517675408
```

1.1.2 Ejemplos con Unicode.

Unicode: estándar para la codificación de caracteres, que permite el tratamiento informático, la transmisión y visualización de textos de muchos idiomas y disciplinas técnicas. Unicode intenta tener universalidad, uniformidad y unicidad. Unicode define tres formas de codificación bajo el nombre UTF (Unicode transformation format): UTF8, UTF16, UTF32. Véase <https://es.wikipedia.org/wiki/Unicode>

Python 3 utiliza internamente el tipo de datos `str` para representar cadenas de texto Unicode, lo que significa que se puede escribir y manipular texto en cualquier idioma sin preocuparte por la codificación. La compatibilidad con UTF-8 en Python significa que se puede leer, escribir y manipular archivos de texto en cualquier idioma, y también trabajar con datos provenientes de fuentes diversas, como bases de datos, API web, etc., que pueden contener texto en diferentes idiomas y codificaciones.

A continuación se muestran algunos ejemplos.

```
compañero = 'Luismi' # puedo usar la ñ como parte del nombre de la variable  
print(compañero)
```

Luismi

Los códigos Unicode de cada caracter se pueden dar en decimal o hexadecimal, por ejemplo para el símbolo π se tiene el código decimal `120587` y hexadecimal `0x1D70B`. La función `chr()` convierte ese código en el caracter correspondiente:

```
chr(0x1D70B)
```

```
'π'
```

```
chr(120587)
```

```
'π'
```

La función `ord()` obtiene el código Unicode de un caracter y lo regresa en decimal:

```
ord('π')
```

```
120587
```

Podemos usar la función `print()` para realizar una impresión con formato como sigue:

```
π = 3.141592
print('{:04d} \t {} = {}'.format(ord('π'), 'π', π)) # Impresión en decimal
print('{:04o} \t {} = {}'.format(ord('π'), 'π', π)) # Impresión en octal
print('{:04x} \t {} = {}'.format(ord('π'), 'π', π)) # Impresión en hexadecimal
```

```
120587    π = 3.141592
```

```
353413    π = 3.141592
```

```
1d70b     π = 3.141592
```

Podemos usar acentos:

```
México = 'El ombligo de la luna'
print(México)
```

```
El ombligo de la luna
```

Puedo saber el tipo de codificación que usa Python de la siguiente manera:

```
import sys
sys.stdout.encoding
```

```
'UTF-8'
```

También es posible obtener más información de los códigos unicode como sigue:

```
import unicodedata

u = chr(233) + chr(0x0bf2) + chr(6000) + chr(13231)
print('cadena : ', u)
print()
for i, c in enumerate(u):
    print('{} {:>5x} {:>3}'.format(c, ord(c), unicodedata.category(c)), end=" ")
    print(unicodedata.name(c))
```

```
cadena :  éϣϯ𐀀𐀀
```

é e9 Ll LATIN SMALL LETTER E WITH ACUTE
ௌ bf2 No TAMIL NUMBER ONE THOUSAND
Ბ 1770 Lo TAGBANWA LETTER SA
᳚ 33af So SQUARE RAD OVER S SQUARED

Véase: <https://docs.python.org/3/howto/unicode.html>

1.2 Asignación múltiple.

Es posible definir varias variables en una sola instrucción:

```
x = y = z = 25
```

```
print(type(x), type(y), type(z))
```

```
<class 'int'> <class 'int'> <class 'int'>
```

```
print(id(x), id(y), id(z))
```

```
94157725269672 94157725269672 94157725269672
```

Observa que se creó el objeto `25` de tipo `<class 'int'>` y los nombres `x`, `y` y `z` son etiquetas al mismo objeto, como se verifica imprimiendo el identificador de cada variable usando la función `id()`.

Podemos eliminar la etiqueta `x` con la función `del()`:

```
del(x)
```

Ahora ya no es posible hacer referencia al objeto `25` usando `x`:

```
print(x)
```

```
NameError: name 'x' is not defined
```

Pero si es posible hacer referencia al objeto `25` con los nombres `y` y `z`:

```
print(y,z)
```

```
25 25
```

Podemos hacer una asignación múltiples de objetos diferentes a variables diferentes:

```
x, y, z = 'eje x', 3.141592, 50
```

```
print(type(x), type(y), type(z))
```

```
<class 'str'> <class 'float'> <class 'int'>
```

```
print(id(x), id(y), id(z))
```

```
139672217233008 139672517638832 94157725270472
```

Como se observa, ahora las variables `x`, `y` y `z` hacen referencia a diferentes objetos, de distinto tipo.

1.2.1 Ejemplos de nombres NO válidos.

Los siguientes son ejemplos NO VALIDOS para el nombre de variables. Al ejecutar las celdas se obtendrá un error en cada una de ellas.

```
1luis = 20 # No se puede iniciar con un número
```

SyntaxError: invalid decimal literal (953519616.py, line 1)

```
luis$ = 8.2323 # No puede contener caracteres especiales
```

SyntaxError: invalid syntax (2653363214.py, line 1)

```
for = 35 # Algunos nombres ya están reservados
```

SyntaxError: invalid syntax (2521306807.py, line 1)

1.3 Palabras reservadas.

Tampoco es posible usar las palabras reservadas para nombrar variables. Podemos conocer las palabras reservadas como sigue:

```
help('keywords')
```

Here is a list of the Python keywords. Enter any keyword to get more help.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with

`await`
`break`

`finally`
`for`


`nonlocal`
`not`

`yield`

11 Manejo de excepciones.

Objetivo. ...

Funciones de Python: ... [Errors and Exceptions](#)

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

12 Tipos de errores:

En Python existen dos tipos de errores por los cuales un programa se detiene y no continua con su ejecución normal.

12.1 Errores de sintaxis.

Ocurren cuando no se escriben correctamente las expresiones y declaraciones, siguiendo la especificación de la interfaz de Python.

Por ejemplo:

```
# Escribimos a propósito 'printf' que es un nombre incorrecto.
printf('Hola mundo!')
```

NameError: name 'printf' is not defined

- Observa que el tipo de error se imprime cuando éste ocurre.
- En el caso anterior el error fue de tipo **NameError**, por lo que hay que revisar que todo esté correctamente escrito.

12.2 Errores provocados por excepciones.

Son errores lógicos que detienen la ejecución de un programa aún cuando la sintaxis sea la correcta.

Por ejemplo:

```
def raizCuadrada(numero):
    numero = float(numero)
    print("La raíz cuadrada del número {} es {}".format(numero, numero ** 0.5))
```

```
# Ejemplo correcto que se ejecuta sin problemas.
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0


```
# Ejemplo correcto, se calcula la raíz cuadrada de -1 en  
# El resultado es un número complejo. En este caso Python  
# se encarga de realizar las conversiones necesarias.  
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es $(6.123233995736766e-17+1j)$

```
# Ejemplo incorrecto. No es posible calcular la raíz cuadrada  
# de un número complejo, es una operación no definida.  
raizCuadrada(1+1j)
```

`TypeError: float() argument must be a string or a real number, not 'complex'`

En el ejemplo anterior se produce un error de tipo `TypeError`, es decir hay incompatibilidad con los tipos de datos que se están manipulando.

```
# Ejemplo incorrecto. No se puede calcular la raíz cuadrada  
# de una cadena.  
raizCuadrada("hola")
```

`ValueError: could not convert string to float: 'hola'`

En el ejemplo anterior se produce un error de tipo `ValueError`, es decir es decir hay un problema con el contenido del objeto.

13 Manejo de excepciones con: `try`, `except`, `finally`

Los errores que se pueden manejar, son aquellos errores lógicos como los presentados anteriormente en donde es posible “predecir” el tipo de error que puede ocurrir de acuerdo con la implementación que estamos realizando.

Todas las excepciones en Python son ejemplos concretos de una clase (*instance*) que se derivan de la clase principal [BaseExcepcion](#). Más detalles se pueden consultar [aquí](#).

Las excepciones se pueden capturar y manejar adecuadamente. Para ello se tienen las siguientes herramientas:

- `try`
- `except`
- `else`
- `finally`

Cuando se identifica una sección de código susceptible de errores, ésta puede ser delimitada con la expresión `try`. Cualquier excepción que ocurra dentro de esta sección de código podrá ser capturada y gestionada.

La expresión `except` es la encargada de gestionar las excepciones que se capturan. Si se utiliza sin mayor información, ésta ejecutará el código que contiene para todas las excepciones que ocurran.

En el ejemplo de la función `raizCuadrada()` podemos manejar las excepciones como sigue:

```
def raizCuadrada(numero):
    """
    Función que calcula la raíz cuadrada de un número.

    Parameters
    -----
    numero: int o float
    Valor al que se le desea calcular la raíz cuadrada.

    """
    # Intenta realizar el cálculo que está dentro de try
    try:
        numero = float(numero)
        print(f"La raíz cuadrada del número {numero} es {numero**0.5}")

    # Si ocurre una excepción se captura en el except
    except:
        # No se hace nada con la excepción (por el momento)
        pass

    print('Gracias por usar Python!')
```

Usando la nueva versión de la función `raizCuadrada()` intentemos ejecutarla con los ejemplos anteriores:

```
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0
Gracias por usar Python!.

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es (6.123233995736766e-17+1j)
Gracias por usar Python!.

```
raizCuadrada(1+1j)
```

Gracias por usar Python!.

```
raizCuadrada("hola")
```

Gracias por usar Python!.

Observa que ya no hay errores. Esto se debe a que la cláusula `except` captura todas las posibles excepciones, pero no hace nada, y aún con argumentos erróneos, no se sabe que hubo un error.

13.0.1 Gestión general de las excepciones

Ya que sabemos como capturar las excepciones, veamos cómo pueden ser tratadas para dar retroalimentación al usuario.

```
def raizCuadrada(numero):
    """
    Función que calcula la raíz cuadrada de un número.

    Parameters
    -----
    numero: int o float
    Valor al que se le desea calcular la raíz cuadrada.

    """
    # Variable Booleana para manejar las excepciones.
    ocurre_error = False

    # Intenta realizar el cálculo que está dentro de try
    try:
        numero = float(numero)
        print("La raíz cuadrada del número {} es {}".format(numero, numero ** 0.5))

    # Si ocurre una excepción se captura en el except
    except:
        ocurre_error = True

    # Cuando ocurre un error se hace lo siguiente:
    if ocurre_error:
        print("Cuidado, hubo una falla en el programa, no se pudo realizar el cálculo")
    else:
        print('Gracias por usar Python!')
```

```
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0
Gracias por usar Python!.

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es (6.123233995736766e-17+1j)
Gracias por usar Python!.

```
raizCuadrada(1+1j)
```

Cuidado, hubo una falla en el programa, no se pudo realizar el cálculo

```
raizCuadrada("hola")
```

Cuidado, hubo una falla en el programa, no se pudo realizar el cálculo

Observa que ahora se avisa al usuario que hubo un error al ejecutar la función por lo que el cálculo no se realizó. Sin embargo hace falta más información.

13.0.2 Gestión de las excepciones por su tipo.

La expresión `except` puede ser utilizada de forma tal que ejecute código dependiendo del tipo de error que ocurra. En este caso sabemos que pueden ocurrir dos tipos de errores: `TypeError` y `ValueError`. Entonces la nueva versión de la función `raizCuadrada()` es como sigue:

```
def raizCuadrada(numero):
    """
    Función que calcula la raíz cuadrada de un número.

    Parameters
    -----
    numero: int o float
    Valor al que se le desea calcular la raíz cuadrada.

    """
    # Variable Booleana para manejar las excepciones.
    ocurre_error = False

    # Intenta realizar el cálculo que está dentro de try
    try:
        numero = float(numero)
        print("La raíz cuadrada del número {} es {}".format(numero, numero ** 0.5))

    # En esta sección se trata la excepción de tipo TypeError
    except TypeError:
        ocurre_error = True
        print("Ocurrió un error de tipo: TypeError, verifique que los tipos sean")

    # En esta sección se trata la excepción de tipo ValueError
    except ValueError as detalles:
        ocurre_error = True
        print("Ocurrió un error de tipo ValueError, verifique el contenido de los")

    # En esta sección se tratan todas las otras posible excepciones
    except:
        ocurre_error = True
        print("Ocurrió algo misterioso")

    # Cuando ocurre un error se hace lo siguiente:
    if ocurre_error:
        print("Hubo una falla en el programa, no se pudo realizar el cálculo")
    else:
        print('Gracias por usar Python!.')
```

```
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0
Gracias por usar Python!.

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es (6.123233995736766e-17+1j)
Gracias por usar Python!.

```
raizCuadrada(1+4j)
```

Ocurrió un error de tipo: TypeError, verifique que los tipos sean compatibles.
Hubo una falla en el programa, no se pudo realizar el cálculo

```
raizCuadrada("hola")
```

Ocurrió un error de tipo ValueError, verifique el contenido de los argumentos.
Hubo una falla en el programa, no se pudo realizar el cálculo

Observa que ya se da mayor información sobre el tipo de error que ocurrió y el usuario puede saber que hacer como corregir los errores.

Todos los tipos de errores que existen en Python se pueden consulta en [Concrete exceptions](#).

13.0.3 Información del error

Se puede capturar toda la información del error para pasarla al usuario. Esto se hace como sigue:

```
def raizCuadrada(numero):  
    """  
    Función que calcula la raíz cuadrada de un número.  
  
    Parameters  
    -----  
    numero: int o float  
    Valor al que se le desea calcular la raíz cuadrada.  
  
    """  
    # Variable Booleana para manejar las excepciones.  
    ocurre_error = False  
  
    # Intenta realizar el cálculo que está dentro de try  
    try:  
        numero = float(numero)  
        print("La raíz cuadrada del número {} es {}".format(numero, numero ** 0.5))  
  
    # En esta sección se trata la excepción de tipo TypeError y se obtienen los c  
    except TypeError as info:  
        ocurre_error = True  
        print("Ocurrió un error (TypeError):", info)
```

```
# En esta sección se trata la excepción de tipo ValueError y se obtienen los
except ValueError as info:
    ocurre_error = True
    print("Ocurrió un error (ValueError):", info)

# En esta sección se tratan todas las otras posible excepciones
except:
    ocurre_error = True
    print("Ocurrió algo misterioso")

# Cuando ocurre un error se hace lo siguiente:
if ocurre_error:
    print("Hubo una falla en el programa, no se pudo realizar el cálculo")
else:
    print('Gracias por usar Python!')
```

```
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0
Gracias por usar Python!.

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es (6.123233995736766e-17+1j)
Gracias por usar Python!.

```
raizCuadrada(1+4j)
```

Ocurrió un error (TypeError): float() argument must be a string or a real number, not 'complex'
Hubo una falla en el programa, no se pudo realizar el cálculo

```
raizCuadrada("hola")
```

Ocurrió un error (ValueError): could not convert string to float: 'hola'
Hubo una falla en el programa, no se pudo realizar el cálculo

Observa que ahora además de conocer el tipo de error, también se muestra toda la información del error para que el usuario tome las acciones pertinentes.

13.0.4 finally

Esta sección se ejecuta siempre, sin importar si hubo una excepción o no.

```
def raizCuadrada(numero):
    """
    Función que calcula la raíz cuadrada de un número.
```

```

Parameters
-----
numero: int o float
Valor al que se le desea calcular la raíz cuadrada.

"""
# Variable Booleana para manejar las excepciones.
ocurre_error = False

# Intenta realizar el cálculo que está dentro de try
try:
    numero = float(numero)
    print("La raíz cuadrada del número {} es {}".format(numero, numero ** 0.5))

# En esta sección se trata la excepción de tipo TypeError y se obtienen los detalles
except TypeError as info:
    ocurre_error = True
    print("Ocurrió un error (TypeError):", info)

# En esta sección se trata la excepción de tipo ValueError y se obtienen los detalles
except ValueError as info:
    ocurre_error = True
    print("Ocurrió un error (ValueError):", info)

# En esta sección se tratan todas las otras posibles excepciones
except:
    ocurre_error = True
    print("Ocurrió algo misterioso")

# Cuando ocurre un error se hace lo siguiente:
finally:
    if ocurre_error:
        print("Hubo una falla en el programa, no se pudo realizar el cálculo")
    else:
        print('Gracias por usar Python!')

```

```
raizCuadrada(1)
```

La raíz cuadrada del número 1.0 es 1.0
Gracias por usar Python!.

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1.0 es (6.123233995736766e-17+1j)
Gracias por usar Python!.

```
raizCuadrada(1+4j)
```

Ocurrió un error (TypeError): float() argument must be a string or a real number, not 'complex'

Hubo una falla en el programa, no se pudo realizar el cálculo

```
raizCuadrada("hola")
```

Ocurrió un error (ValueError): could not convert string to float: 'hola'

Hubo una falla en el programa, no se pudo realizar el cálculo

13.0.5 Lanzar excepciones controladas.

Es posible presentar toda la información que genera la excepción y agregarle notas para el usuario. Para agregar notas usamos el método `add_note()` y para lanzar la excepción una vez controlada usamos `raise`. La siguiente versión de la función `raizCuadrada()` tiene al final una cláusula `else`, la cual se ejecuta cuando no ocurre ninguna excepción. En este caso, dentro del `try` realizamos el cálculo de la raíz cuadrada y en el `else` hacemos la impresión del resultado.

```
def raizCuadrada(numero):
    """
    Función que calcula la raíz cuadrada de un número.

    Parameters
    -----
    numero: int o float
    Valor al que se le desea calcular la raíz cuadrada.

    """

    # Intenta realizar el cálculo que está dentro de try
    try:
        numero_cuadrado = float(numero) ** 0.5

    # En esta sección se trata la excepción de tipo TypeError y se obtienen los c
    except TypeError as info:
        info.add_note("\n" + "-"*20)
        info.add_note(f"raizCuadrada{numero}: Para calcular una raíz cuadrada, el
        info.add_note("-"*20)
        raise # Lanzamos la excepción con toda la información

    # En esta sección se trata la excepción de tipo ValueError y se obtienen los
    except ValueError as info:
        info.add_note("\n" + "-"*20)
        info.add_note(f"raizCuadrada('{numero}')": Para calcular una raíz cuadrada
        info.add_note("-"*20)
        raise # Lanzamos la excepción con toda la información

    # En esta sección se tratan todas las otras posible excepciones
    except:
        print("Ocurrió algo misterioso")
```



```
else:  
    print("La raíz cuadrada del número {} es {}".format(numero, numero_cuadrada))
```

```
raizCuadrada(1)
```

La raíz cuadrada del número 1 es 1.0

```
raizCuadrada(-1)
```

La raíz cuadrada del número -1 es (6.123233995736766e-17+1j)

```
raizCuadrada(1+4j)
```

TypeError: float() argument must be a string or a real number, not 'complex'

```
raizCuadrada("hola")
```

ValueError: could not convert string to float: 'hola'

7 Entrada y salida estándar.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

8 Entrada estándar: `input()`

La entrada estándar para proporcionar información a un programa se realiza mediante la función `input()`.

Veamos algunos ejemplos:

```
# Cuando se ejecuta, se espera a que el usuario teclee algo y luego de <enter>
input()
```

34

'34'

```
# Se puede asignar el valor que se teclea a una variable
entrada = input()
```

34

```
# Ahora imprimimos el valor guardado
print(entrada)
```

34

```
# Se puede poner un mensaje para que el usuario sepa
# lo que se espera:
entrada = input('Teclea un valor entero :')
```

Teclea un valor entero : 5

```
# Lo que se lee siempre se transforma en una cadena de texto:
print(type(entrada))
print(entrada)
```

<class 'str'>

5

```
# Podemos hacer el 'casting' para transformar lo que teclea el usuario  
# en el tipo de dato requerido:  
entrada = int(input('Teclea un valor entero :'))
```

Teclea un valor entero : 5

```
print(type(entrada))  
print(entrada)
```

```
<class 'int'>  
5
```

Lo anterior se debe realizar con más cuidado e incluso usando declaraciones para capturar posibles errores del usuario al teclear un valor.

9 Salida estándar: `print()`

Existen varias formas de presentar la salida de un programa al usuario, la más común es en la pantalla (la salida estándar).

En todos los casos se desea un control adecuado sobre el formato de la salida. Para ello se tienen varias maneras de controlar esta salida:

9.1 Cadenas con formato `f` o `F`.

Este tipo de cadenas se forman anteponiendo una `f` o `F` al principio de la misma. De esta forma, es posible poner variables entre llaves `{}` dentro de la definición de la cadena.

```
nombre = 'LUIS MIGUEL'  
edad = 25  
f'Hola mi nombre es {nombre} y tengo {edad} años'
```

'Hola mi nombre es LUIS MIGUEL y tengo 25 años'

En el caso de números es posible agregar un formato:

```
import math  
print(f'El valor de PI es aproximadamente {math.pi:.10f}.') # {valor:formato}
```

El valor de PI es aproximadamente 3.1415926536.

También es posible alinear el texto de la salida.

```
na1 = 'Fulano'; n1 = 5_521_345_678  
na2 = 'Sutano'; n2 = 7_712_932_143
```

```
print(f'{na1:10} ==> {n1:15d}') # alineación del texto
print(f'{na2:10} ==> {n2:15d}') # alineación del texto
```

```
Fulano      ==>      5521345678
Sutano      ==>      7712932143
```

Para más información véase [Format Specification Mini-Language](#)

9.2 Método `format()`

Las cadenas tienen un método llamado `format()` que permite darle formato a la misma.

Veamos unos ejemplos:

```
print('El curso se llama "{}" y tenemos {} alumnos'.format('Python de cero a expe
```

El curso se llama "Python de cero a experto" y tenemos 100 alumnos

```
votos_a_favor = 42_572_654 # Este es un formato de número entero
votos_en_contra = 43_132_495 # que usa _ para separar los miles
total_de_votos = votos_a_favor + votos_en_contra
porcentaje = votos_a_favor / total_de_votos

# El primer dato se alinea al centro usando ^
# El segundo dato tendrá dos valores antes del punto y dos valores después.
print('{:^20} votos a favor ({:2.2%})'.format(votos_a_favor, porcentaje))
```

```
42572654          votos a favor (49.67%)
```

```
# Se pueden usar números para identificar los argumentos de format()
print('{0} y {1}'.format('el huevo', 'la gallina'))
print('{1} y {0:^20}'.format('el huevo', 'la gallina'))
```

el huevo y la gallina

la gallina y el huevo

```
# Se le puede dar nombre a los argumentos para que
# sea más fácil entender la salida
print('Esta {sustantivo} es {adjetivo}'.format(adjetivo='exquisita', sustantivo=
```

Esta comida es exquisita.

```
# Se pueden combinar números con nombres de argumentos
print('El {0}, el {1}, y el {otro}'.format('Bueno', 'Malo', otro='Feo'))
```

El Bueno, el Malo, y el Feo.

```
gatos = {'Siamés': 5, 'Siberiano': 4, 'Sphynx': 0}

# Podemos usar las características de los diccionarios
# para imprimir la salida.
print('Sphynx: {g[Sphynx]:d}; Siamés: {g[Siamés]:d}; Siberiano: {g[Siberiano]:d}')
```

Sphynx: 0; Siamés: 5; Siberiano: 4

```
# Una manera más entendible:
print('Sphynx: {Sphynx:d}; Siamés: {Siamés:d}; Siberiano: {Siberiano:d}'.format(*
```

Sphynx: 0; Siamés: 5; Siberiano: 4

```
for x in range(1, 11):
    print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
```

```
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000
```

Para más información véase [Format String Syntax](#)

9.3 Forma antigua de formatear la salida

Se puede seguir usando la forma antigua de la salida.

```
import math
print('El valor aproximado de pi es %5.6f.' % math.pi)
```


El valor aproximado de pi es 3.141593.



2 Expresiones y declaraciones.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

2.1 Expresiones

En matemáticas se define una expresión como una colección de símbolos que juntos expresan una cantidad, por ejemplo, el perímetro de una circunferencia es $2\pi r$.

En Python una **expresión** está compuesta de una combinación válida de valores, variables, operadores, funciones y métodos, que se puede evaluar y **da como resultado al menos un valor**.

En esencia, una **expresión es cualquier cosa que pueda ser evaluada y producir un resultado**.

Las expresiones pueden ser simples o complejas, pero en general, representan un valor único, por ejemplo:

```
a = 2**32
```

Véase más en [The Python language reference: Expressions](#) y [Python expressions](#).

Veamos algunos ejemplos:

Expresiones simples

```
23
```

23

```
5 + 3
```

8

```
a = 5  
a ** 2
```

25

Expresión que ejecuta una función

```
len('Hola mundo')
```

10

Expresiones usando operadores

```
# Otros ejemplos
x = 1
y = x + 2
z = y ** 3

print(x)
print(y)
print(z)

# Operación Booleana
7 == 2 * 2 * 2
```

1
3
27

False

Expresiones más complicadas

```
# Se combinan varias operaciones matemáticas
b = 2.14
c = 0.1 + 4j

(3.141592 * c + b) / a
```

(0.4908318400000001+2.5132736j)

Observa que en todos los ejemplos anteriores se produce al menos un valor como resultado de la ejecución de cada expresión.

2.2 Declaraciones

Una **declaración** (*statement*) se puede pensar como el elemento autónomo más corto de un lenguaje de programación. Un programa se forma de una secuencia que contiene una o más declaraciones. Una declaración contiene componentes internos, que pueden ser otras declaraciones y varias expresiones.

En términos simples, una **declaración** es una **instrucción que realiza una acción**.

Puede ser una asignación de valor a una variable, una llamada a una función, una estructura de control de flujo (como un ciclo o una condición), una definición de función, etc.

Véase más en [Simple statements](#), [Compound statements](#) y [Python statements \(wikipedia\)](#).

Veamos algunos ejemplos:

Declaración que hace una asignación

```
x = 0
```

Declaración usando un condicional

```
if x < 0:  
    pass
```

Declaración que realiza un ciclo

```
for i in range(0,5):  
    pass
```

Declaración de una función


```
def mult(a, b):  
    return a * b
```

Here is a note

13 Estructuras de datos concisas.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

14 Listas concisas

En matemáticas podemos definir un conjunto como sigue:

$$S = \{x^2 : x \in (0, 1, 2, \dots, 9)\} = \{0, 1, 4, \dots, 81\}$$

En Python es posible crear este conjunto usando lo que conoce como *list comprehensions* (generación corta de listas) como sigue:

```
S = [x**2 for x in range(10)]  
print(S)
```

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

Las listas concisas son usadas para construir listas de una manera muy concisa, natural y fácil, como lo hace un matemático. La forma precisa de construir listas concisas es como sigue:

```
[ expresion for i in S if predicado ]
```

Donde **expresion** es una expresión que se va a aplicar a cada elemento **i** de la secuencia **S**; opcionalmente, es posible aplicar el **predicado** antes de aplicar la **expresion** a cada elemento **i**.

14.1 Ejemplo 1.

Usando listas concisas, crear el siguiente conjunto:

$$M = \{\sqrt{x} : x \in (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) \text{ y } x \text{ es par}\} = \{\sqrt{2}, \sqrt{4}, \sqrt{6}, \sqrt{8}, \sqrt{10}\}$$

```
from math import sqrt  
  
M = [sqrt(x) for x in range(2,11) if x%2 == 0]  
print(M)
```

[1.4142135623730951, 2.0, 2.449489742783178, 2.8284271247461903, 3.1622776601683795]

En el ejemplo anterior se distingue lo siguiente:

1. La secuencia de entrada: `range(2,11)` (`[2, 3, 4, 5, 6, 7, 8, 9, 10]`).
2. La etiqueta `i` que representa los miembros de la secuencia de entrada.
3. La expresión de predicado: `if x % 2 == 0`.
4. La expresión de salida `sqrt(x)` que produce los elementos de la lista resultado, los cuales provienen de los miembros de la secuencia de entrada que satisfacen el predicado.

14.2 Ejemplo 2.

Obtener todos los enteros de la siguiente lista, elevarlos al cuadrado y poner el resultado en una lista:

```
lista = [1,'4',9,'luiggi',0,4,('mike','dela+')]

```

```
lista = [1,'4',9,'luiggi',0,4,('mike','dela+')]

```

```
resultado = [x**2 for x in lista if isinstance(x, int)]
print( resultado )

```

[1, 81, 0, 16]

14.3 Ejemplo 3.

Crear la siguiente lista:

$$V = (2^0, 2^1, 2^2, \dots, 2^{12}) = (1, 2, 4, 8, \dots, 4096)$$

```
[2**x for x in range(13)]

```

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]

14.4 Ejemplo 4.

Transformar grados Celsius en Fahrenheit y viceversa.

Celsius → Fahrenheit

```
c = [0, 22.5, 40, 100]
f = [(9/5)*t + 32 for t in c]
print(f)
```

[32.0, 72.5, 104.0, 212.0]

Fahrenheit → Celsius

```
cn = [(5/9)*(t - 32) for t in f]
print(cn)
```

[0.0, 22.5, 40.0, 100.0]

Observa que en ambos ejemplos la expresión es la fórmula de conversión entre grados.

15 Anidado de listas concisas.

15.1 Ejemplo 5.

Crear la siguiente lista:

$$M = \{\sqrt{x} \mid x \in S \text{ y } x \text{ impar}\}$$

con

$$S = \{x^2 : x \in (0 \dots 9)\} = \{0, 1, 4, \dots, 81\}$$

Este ejemplo se puede realizar como sigue:

```
S = [x**2 for x in range(0,10)]
M = [sqrt(x) for x in S if x%2]
print('S = {}'.format(S))
print('M = {}'.format(M))
```

S = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

M = [1.0, 3.0, 5.0, 7.0, 9.0]

Sin embargo, es posible anidar las listas concisas como sigue:

```
M = [sqrt(x) for x in [x**2 for x in range(0,10)] if x%2]
print('M = {}'.format(M))
```

```
M = [1.0, 3.0, 5.0, 7.0, 9.0]
```

15.2 Ejemplo 6.

Sea una matriz identidad de tamaño $n \times n$:

$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{bmatrix}$$

En Python esta matriz se puede representar por la siguiente lista:

```
[[1,0,0, ..., 0],
 [0,1,0, ..., 0],
 [0,0,1, ..., 0],
 .....
 [0,0,0, ..., 1]]
```

- Usando *list comprehensions* anidados se puede obtener dicha lista:

```
n = 8
[[1 if col == row else 0 for col in range(0,n)] for row in range(0,n)]
```

```
[[1, 0, 0, 0, 0, 0, 0, 0],
 [0, 1, 0, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 1, 0],
 [0, 0, 0, 0, 0, 0, 0, 1]]
```

Observa que en este caso la expresión de salida es una lista concisa:

```
[1 if col == row else 0 for col in range(0,n)] .
```

Además, la expresión de salida de esta última lista concisa es el operador ternario: `1 if col == row else 0`

15.3 Ejemplo 7.

Calcular números primos en el rango `[2,50]`.

En este ejercicio se usa el algoritmo conocido como criba de Eratóstenes. Primero se encuentran todos aquellos números que tengan algún múltiplo. En este caso solo vamos a buscar en el intervalo `[2, 50]`.

La siguiente lista concisa calcula los múltiplos de `i` (prueba cambiando el valor de `i` a 2, 3, 4, 5, 6, 7) y observa el resultado.

```
i = 4
[j for j in range(i*2, 50, i)]
```

`[8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48]`

Ahora, para cambiar el valor de `i` a 2, 3, 4, 5, 6, 7 con una lista concisa se puede hacer lo siguiente:

```
[i for i in range(2,8)]
```

`[2, 3, 4, 5, 6, 7]`

Usando las dos listas concisas creadas antes, generamos todos aquellos números en el intervalo `[2, 50]` que tienen al menos un múltiplo (y que por lo tanto no son primos)

```
noprimos = [j for i in range(2,8) for j in range(i*2, 50, i)]
print('NO primos: \n{}'.format(noprimos))
```

NO primos:

`[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 10, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 21, 28, 35, 42, 49]`

Para encontrar los primos usamos una lista concisa verificando los números que faltan en la lista de `noprimos`, esos serán los números primos que estamos buscando:

```
primos = [x for x in range(2,50) if x not in noprimos]
print('Primos: \n{}'.format(primos))
```

Primos:

`[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]`

Juntando todo:

```
[x for x in range(2,50) if x not in [j for i in range(2,8) for j in range(i*2, 50)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

15.4 Listas concisas con elementos no numéricos.

Las listas también pueden contener otro tipo de elementos, no solo números. Por ejemplo:

```
mensaje = 'La vida no es la que uno vivió, sino la que uno recuerda'
```

```
print(mensaje)
```

La vida no es la que uno vivió, sino la que uno recuerda

```
palabras = mensaje.split()
print(palabras,end='')
```

```
['La', 'vida', 'no', 'es', 'la', 'que', 'uno', 'vivió,', 'sino', 'la', 'que', 'uno',
'recuerda']
```

Vamos a crear una lista cuyos elementos contienen cada palabra de la lista anterior en mayúsculas, en forma de título y su longitud, estos tres elementos agregados en una tupla:

```
tabla = [(w.upper(), w.title(), len(w)) for w in palabras]
print(tabla)
```

```
[('LA', 'La', 2), ('VIDA', 'Vida', 4), ('NO', 'No', 2), ('ES', 'Es', 2), ('LA', 'La', 2),
('QUE', 'Que', 3), ('UNO', 'Uno', 3), ('VIVIÓ,', 'Vivió,', 6), ('SINO', 'Sino', 4),
('LA', 'La', 2), ('QUE', 'Que', 3), ('UNO', 'Uno', 3), ('RECUERDA', 'Recuerda', 8)]
```

16 Conjuntos concisos

Al igual que las listas concisas, también es posible crear conjuntos usando los mismos principios, la única diferencia es que la secuencia que resulta es un objeto de tipo **set**.

Definición.

```
{expression(variable) for variable in input_set [predicate][, ...]}
```

1. **expression** : Es una expresión **opcional** de salida que produce los miembros del nuevo conjunto a partir de los miembros del conjunto de entrada que satisfacen el **predicate**.
2. **variable** : Es una variable **requerida** que representa los miembros del conjunto de entrada.

3. `input_set`: Representa la secuencia de entrada. (**requerido**).
4. `predicate`: Expresión **opcional** que actúa como un filtro sobre los miembros del conjunto de entrada.
5. `[, ...]`: Otra *comprehension* anidada **opcional**.

16.1 Ejemplo 8.

Supongamos que deseamos organizar una lista de nombres de tal manera que no haya repeticiones, que los nombres tengan más de un carácter y que su representación sea con la primera letra mayúscula y las demás minúsculas. Por ejemplo, una lista aceptable sería:

```
nombres = ['Luis', 'Juan', 'Angie', 'Pedro', 'María', 'Diana']
```

Leer una lista de nombres del archivo `nombres` y procesarlos para obtener una lista similar a la descrita.

```
# Abrimos el archivo en modo lectura
archivo = open('nombres','r')

# Leemos la lista de nombres y los ponemos en una lista
lista_nombres = archivo.read().split()

# Vemos la lista de nombres
print(lista_nombres)
```

```
['A', 'LuCas', 'Sidronio', 'Michelle', 'a', 'ANGIE', 'Luis', 'lucas', 'MICHelle',
'Pedro', 'PEPE', 'Manu', 'luis', 'diana', 'sidronio', 'pepe', 'a', 'a', 'b']
```

```
# Procesamos las palabras como se requiere
nombres_set = {nombre[0].upper() + nombre[1:].lower()
               for nombre in lista_nombres
               if len(nombre) > 1 }
print(nombres_set)
```

```
{'Pepe', 'Lucas', 'Angie', 'Sidronio', 'Pedro', 'Manu', 'Luis', 'Diana', 'Michelle'}
```

```
# Transformamos el conjunto a una lista
nombres = list(nombres_set)
print(nombres)
```

```
['Pepe', 'Lucas', 'Angie', 'Sidronio', 'Pedro', 'Manu', 'Luis', 'Diana', 'Michelle']
```

16.2 Ejemplo 9.

Observa los siguientes ejemplos de conjuntos concisos y explica su funcionamiento.

```
{s for s in [1, 2, 1, 0]}
```

{0, 1, 2}

```
{s**2 for s in [1, 2, 1, 0]}
```

{0, 1, 4}

```
{s**2 for s in range(10)}
```

{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

```
{s for s in range(10) if s % 2}
```

{1, 3, 5, 7, 9}

```
{(m, n) for n in range(2) for m in range(3, 5)}
```

{(3, 0), (3, 1), (4, 0), (4, 1)}

17 Dicionarios concisos

- Es un método para transformar un diccionario en otro diccionario.
- Durante esta transformación, los objetos dentro del diccionario original pueden ser incluidos o no en el nuevo diccionario dependiendo de una condición.
- Cada objeto en el nuevo diccionario puede ser transformado como sea requerido.

Definición.

```
{key:value for (key,value) in dictionary.items()}
```


17.1 Ejemplo 9.

Duplicar el valor (*value*) de cada entrada (*item*) de un diccionario:

Recuerda como funcionan los diccionarios:

```
dicc = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
print(dicc.keys()) # Función para obtener las claves
print(dicc.values()) # Función para obtener los valores
print(dicc.items()) # Función para obtener los items
```

```
dict_keys(['a', 'b', 'c', 'd'])
dict_values([1, 2, 3, 4])
dict_items([('a', 1), ('b', 2), ('c', 3), ('d', 4)])
```

Para crear el diccionario del ejemplo hacemos lo siguiente:

```
# Definición del diccionario
dicc = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}

# Duplicación de los valores del diccionario
dicc_doble = {k:v*2 for (k,v) in dicc.items()}

# Mostramos el resultado
print(dicc)
print(dicc_doble)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
{'a': 2, 'b': 4, 'c': 6, 'd': 8, 'e': 10}
```

17.2 Ejemplo 10.

Duplicar la clave (*key*) de cada entrada (*item*) del diccionario:

```
dict1_keys = {k*2:v for (k,v) in dict1.items()}
print(dict1_keys)
```

```
{'aa': 1, 'bb': 2, 'cc': 3, 'dd': 4}
```

17.3 Ejemplo 11.

Crear un diccionario donde la clave sea un número divisible por 2 en un rango de 0 a 10 y sus valores sean el cuadrado de la clave.

```
# La forma tradicional
numeros = range(11)
dicc = {}

for n in numeros:
    if n%2==0:
        dicc[n] = n**2

print(dicc)
```

```
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

```
# Usando dict comprehensions
dicc_smart = {n:n**2 for n in numeros if n%2 == 0}

print(dicc_smart)
```

```
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

17.4 Ejemplo 12.

Intercambiar las claves y los valores en un diccionario.

```
a_dict = {'a': 1, 'b': 2, 'c': 3}
{value:key for key, value in a_dict.items()}
```

```
{1: 'a', 2: 'b', 3: 'c'}
```

```
# OJO: No siempre es posible hacer lo anterior.
a_dict = {'a': [1, 2, 3], 'b': 4, 'c': 5}
{value:key for key, value in a_dict.items()}
```

```
TypeError: unhashable type: 'list'
```

17.5 Ejemplo 14.

Convertir Fahrenheit a Celsius y viceversa.

```
# Usando map, lambda y diccionarios
[32.0, 72.5, 104.0, 212.0]
fahrenheit_dict = {'t1':32.0, 't2':72.5, 't3':104.0, 't4':212.0}

celsius = list(map(lambda f: (5/9)*(f-32), fahrenheit_dict.values()))

celsius_dict = dict(zip(fahrenheit_dict.keys(), celsius))

print(celsius_dict)
```

```
{'t1': 0.0, 't2': 22.5, 't3': 40.0, 't4': 100.0}
```

```
# Usando dict comprehensions !
celsius_smart = {k:(5/9)*(v-32) for (k,v) in fahrenheit_dict.items()}
print(celsius_smart)
```

```
{'t1': 0.0, 't2': 22.5, 't3': 40.0, 't4': 100.0}
```

17.6 Ejemplo 15.

Dado un diccionario, cuyos valores son enteros, crear un nuevo diccionario cuyos valores sean mayores que 2.

```
a_dict = {'a':1, 'b':2, 'c':3, 'd':4, 'e':5, 'f':6, 'g':7, 'h':8}
print(a_dict)

a_dict_cond = { k:v for (k,v) in a_dict.items() if v > 2 }
print(a_dict_cond)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8}
{'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8}
```

17.7 Ejemplo 16.

Dado un diccionario, cuyos valores son enteros, crear un nuevo diccionario cuyos valores sean mayores que 2 y que además sean pares.

```
a_dict_cond2 = { k:v for (k,v) in a_dict.items() if (v > 2) and (v % 2) == 0}
print(a_dict_cond2)
```

```
{'d': 4, 'f': 6, 'h': 8}
```

17.8 Ejemplo 17.

Dado un diccionario, cuyos valores son enteros, crear un nuevo diccionario cuyos valores sean mayores que 2 y que además sean pares y divisibles por 3.

```
# La forma tradicional
a_dict_cond3_loop = {}

for (k,v) in a_dict.items():
    if (v>=2 and v%2 == 0 and v%3 == 0):
        a_dict_cond3_loop[k] = v

print(a_dict_cond3_loop)
```

```
{'f': 6}
```

```
# Usando dict comprehensions
a_dict_cond3 = {k:v for (k,v) in a_dict.items() if v>2 if v%2 == 0 if v%3 == 0}

print(a_dict_cond3)
```

```
{'f': 6}
```

17.9 Ejemplo 18.

Apartir de un diccionario con valores enteros, identificar los valores pares y los impares, y sustituir los valores por etiquetas 'par' e 'impar' según corresponda.

```
print(a_dict)
a_dict_else = { k:('par' if v%2==0 else 'impar') for (k,v) in a_dict.items() }
```

```
print(a_dict_else)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6, 'g': 7, 'h': 8}
```

```
{'a': 'impar', 'b': 'par', 'c': 'impar', 'd': 'par', 'e': 'impar', 'f': 'par', 'g':  
'impar', 'h': 'par'}
```

17.10 Ejemplo 19.

Crear un diccionario cuyos valores sean diccionarios.

```
# con dict comprehensions
anidado = {'primero':{'a':1}, 'segundo':{'b':2}, 'tercero':{'c':3}}
pi = 3.1415
float_dict = {e_k:{i_k:i_v*pi for (i_k, i_v) in e_v.items()} for (e_k, e_v) in anidado.items()}
print(float_dict)
```

```
{'primero': {'a': 3.1415}, 'segundo': {'b': 6.283}, 'tercero': {'c': 9.4245}}
```

```
# La forma tradicional sería:
anidado = {'primero':{'a':1}, 'segundo':{'b':2}, 'tercero':{'c':3}}
pi = 3.1415
for (e_k, e_v) in anidado.items():
    for (i_k, i_v) in e_v.items():
        e_v.update({i_k: i_v * pi})

anidado.update({e_k:e_v})

print(anidado)
```

```
{'primero': {'a': 3.1415}, 'segundo': {'b': 6.283}, 'tercero': {'c': 9.4245}}
```

17.11 Ejemplo 20.

Eliminar números duplicados de una lista.

```
numeros = [i for i in range(1,11)] + [i for i in range(1,6)]
numeros
```

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

```
# Una manera es:
numeros_unicos = []
for n in numeros:
    if n not in numeros_unicos:
        numeros_unicos.append(n)
numeros_unicos
```

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

```
# Otra forma mas pythonica!
numeros_unicos_easy = list(set(numeros))
numeros_unicos_easy
```

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

17.12 Ejemplo 21.

Eliminar objetos duplicados de una lista de diccionarios.

```
datos = [
    {'id': 10, 'dato': '...'},
    {'id': 11, 'dato': '...'},
    {'id': 12, 'dato': '...'},
    {'id': 10, 'dato': '...'},
    {'id': 11, 'dato': '...'},
]

print(datos)
```

[{'id': 10, 'dato': '...'}, {'id': 11, 'dato': '...'}, {'id': 12, 'dato': '...'}, {'id': 10, 'dato': '...'}, {'id': 11, 'dato': '...'}]

```
# La forma tradicional
objetos_unicos = []
for d in datos:
    dato_existe = False
    for ou in objetos_unicos:
        if ou['id'] == d['id']:
            dato_existe = True
            break
    if not dato_existe:
        objetos_unicos.append(d)
```

```
print(objetos_unicos)
```

```
[{'id': 10, 'dato': '...'}, {'id': 11, 'dato': '...'}, {'id': 12, 'dato': '...'}]
```

```
# Una mejor manera.  
objetos_unicos_easy = { d['id']:d for d in datos }.values()  
  
print(list(objetos_unicos_easy))
```

```
[{'id': 10, 'dato': '...'}, {'id': 11, 'dato': '...'}, {'id': 12, 'dato': '...'}]
```

17.13 Ejemplo 22.

Sea un diccionario que tiene como claves letras minúsculas y mayúsculas, y como valores números enteros:

```
mcase = {'z':23, 'a':30, 'b':21, 'A':78, 'Z':4, 'C':43, 'B':89}
```

- Sumar los valores que corresponden a la misma letra, mayúscula y minúscula.
- Construir un diccionario cuyas claves sean solo letras minúsculas y sus valores sean la suma antes calculada.

```
mcase = {'z':23, 'a':30, 'b':21, 'A':78, 'Z':4, 'C':43, 'B':89}  
mcase_freq = {k.lower() :  
               mcase.get(k.lower(), 0) + mcase.get(k.upper(), 0)  
               for k in mcase.keys()}  
print(mcase_freq)
```

```
{'z': 27, 'a': 108, 'b': 110, 'c': 43}
```

17.14 Ejemplo 23.

Es posible usar las listas y diccionarios concisos para revisar la lista de archivos de un directorio y sus características.

```
import os, glob  
metadata = [(f, os.stat(f)) for f in glob.glob('*.ipynb')]  
metadata
```

```
[('Pensando_como_pythonista_1.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172678637, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=29456, st_atime=1705689214, st_mtime=1705689214,
  st_ctime=1705689214)),
 ('Pythonico_es_mas_bonito_1.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172678648, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=39608, st_atime=1705689214, st_mtime=1705689214,
  st_ctime=1705689214)),
 ('Pythonico_es_mas_bonito_2.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172678651, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=29765, st_atime=1705689214, st_mtime=1705689214,
  st_ctime=1705689214)),
 ('T02_Expr_Decla.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172811400, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=8386, st_atime=1709740171, st_mtime=1709683460,
  st_ctime=1709683460)),
 ('T03_TiposBasico_Operadores.ipynb',
  os.stat_result(st_mode=33188, st_ino=2173059438, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=35254, st_atime=1710044492, st_mtime=1709321664,
  st_ctime=1709321664)),
 ('T04_Cadenas.ipynb',
  os.stat_result(st_mode=33188, st_ino=2174792627, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=18314, st_atime=1709510973, st_mtime=1709323123,
  st_ctime=1709323123)),
 ('T00_Otros.ipynb',
  os.stat_result(st_mode=33188, st_ino=2173586078, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=6513, st_atime=1710044528, st_mtime=1710044528,
  st_ctime=1710044528)),
 ('T01_Etiquetas_y_Palabras_Reservadas.ipynb',
  os.stat_result(st_mode=33188, st_ino=2175783890, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=20284, st_atime=1709740171, st_mtime=1709666919,
  st_ctime=1709666919)),
 ('T05_Estructura_de_Datos.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172811396, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=44066, st_atime=1709567068, st_mtime=1709566559,
  st_ctime=1709566559)),
 ('T06_Control_de_flujo.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172811404, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=27294, st_atime=1709598006, st_mtime=1709598006,
  st_ctime=1709598006)),
 ('zT10_Comprehensions.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172811425, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=49575, st_atime=1710099347, st_mtime=1710099347,
  st_ctime=1710099347)),
 ('zT11_IteradoresGeneradores.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172811428, st_dev=2097240, st_nlink=1,
  st_uid=1000, st_gid=100, st_size=11449, st_atime=1710044551, st_mtime=1710024222,
  st_ctime=1710024222)),
 ('zT12_Decoradores.ipynb',
  os.stat_result(st_mode=33188, st_ino=2172812549, st_dev=2097240, st_nlink=1,
```



```

st_uid=1000, st_gid=100, st_size=13349, st_atime=1709600117, st_mtime=1705689214,
st_ctime=1709596220)),
('zT13_BibliotecaEstandar.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172812551, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=15600, st_atime=1709679822, st_mtime=1709679822,
st_ctime=1709679822)),
('T08_Archivos_Gestores_de_contexto.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172811408, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=5865, st_atime=1709600072, st_mtime=1709600072,
st_ctime=1709600072)),
('T07_Entrada_salida_estandar.ipynb',
 os.stat_result(st_mode=33188, st_ino=2173013626, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=12347, st_atime=1709599786, st_mtime=1709599786,
st_ctime=1709599786)),
('T09_Funciones_y_docstring.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172811410, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=33782, st_atime=1709774198, st_mtime=1709668427,
st_ctime=1709668427)),
('T11_Excepciones.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172811414, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=34820, st_atime=1709740171, st_mtime=1709680686,
st_ctime=1710038069)),
('T10_LambdaExpressions.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172811418, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=15325, st_atime=1710043865, st_mtime=1710043865,
st_ctime=1710043865)),
('T12_IterablesMapFilter.ipynb',
 os.stat_result(st_mode=33188, st_ino=2172811416, st_dev=2097240, st_nlink=1,
st_uid=1000, st_gid=100, st_size=32836, st_atime=1710090534, st_mtime=1710090534,
st_ctime=1710090534))]

```

```
metadata_dict = {f:os.stat(f) for f in glob.glob('*.ipynb')}
```

```
metadata_dict.keys()
```

```

dict_keys(['Pensando_como_pythonista_1.ipynb', 'Pythonico_es_mas_bonito_1.ipynb',
'Pythonico_es_mas_bonito_2.ipynb', 'T02_Expr_Decla.ipynb',
'T03_TiposBasico_Operadores.ipynb', 'T04_Cadenas.ipynb', 'T00_Otros.ipynb',
'T01_Etiquetas_y_Palabras_Reservadas.ipynb', 'T05_Estructura_de_Datos.ipynb',
'T06_Control_de_flujo.ipynb', 'zT10_Comprehensions.ipynb',
'zT11_IteradoresGeneradores.ipynb', 'zT12_Decoradores.ipynb',
'zT13_BibliotecaEstandar.ipynb', 'T08_Archivos_Gestores_de_contexto.ipynb',
'T07_Entrada_salida_estandar.ipynb', 'T09_Funciones_y_docstring.ipynb',
'T11_Excepciones.ipynb', 'T10_LambdaExpressions.ipynb', 'T12_IterablesMapFilter.ipynb'])

```

```
metadata_dict['T02_Expr_Decla.ipynb'].st_size
```

8386

4 Cadenas.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#) 

4.1 Definición de cadenas

Para definir una cadena se utilizan comillas simples `'`, comillas dobles `"` o comillas triples `"""` o `'''`.

```
simples = 'este es un ejemplo usando \' \' '
print(simples)

dobles = "este es un ejemplo usando \" \" "
print(dobles)

triples1 = '''este es un ejemplo usando \'\' \'\' \'\'
print(triples1)

triples2 = """este es un ejemplo usando \"\" \"\" \"\"
print(triples2)
```

```
este es un ejemplo usando ' '
este es un ejemplo usando " "
este es un ejemplo usando ''' '''
este es un ejemplo usando """ """
```

Observa que para poder imprimir `'` dentro de una cadena definida con `'` es necesario usar el caracter `\` antes de `'` para que se imprima correctamente. Lo mismo sucede en los otros ejemplos.

Es posible imprimir `'` sin usar el caracter `\` si la cadena se define con `"` y viceversa, veamos unos ejemplos:

```
# La cadena puede tener ' dentro de " ... "
poema = "Enjoy the moments now, because they don't last forever"
print(poema)
```

```
Enjoy the moments now, because they don't last forever
```

```
# La cadena puede tener " dentro de ' ... '
titulo = 'Python "pythonico" '
print(titulo)
```

```
Python "pythonico"
```

```
# La cadena puede tener " y ' dentro de ''' ... '''
queja = """
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
"""
print(queja)
```

Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela

```
# La cadena puede tener " y ' dentro de """ ... """
queja = """
Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela
"""
print(queja)
```

Desde muy niño
tuve que "interrumpir" 'mi' educación
para ir a la escuela

4.2 Indexación de las cadenas.

La indexación de las cadenas permite acceder a diferentes elementos, o rangos de elementos, de una cadena.

- Todos los elementos de una cadena se numeran empezando en **0** y terminando en **N**, el cual representa el último elemento de la cadena.
- También se pueden usar índices negativos donde **-1** representa el último elemento y **-(N+1)** el primer elemento.

Veamos la siguiente tabla:

cadena :	M	u	r	c	i	é	l	a	g	o
índice +:	0	1	2	3	4	5	6	7	8	9
índice -:	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
ejemplo = 'Murciélago'
```

```
ejemplo[0]
```

'M'

```
ejemplo[5]
```

'é'

```
ejemplo[9]
```

'o'

```
len(ejemplo) # Longitud total de la cadena
```

10

```
ejemplo[-1]
```

'o'

```
ejemplo[-5]
```

'é'

```
ejemplo[-10]
```

'M'

4.3 Inmutabilidad de las cadenas

Los elementos de las cadenas no se pueden modificar:

```
ejemplo[5] = "e"
```

TypeError: 'str' object does not support item assignment

```
cadena='''  
esta es una  
oración  
larga  
'''
```

```
print(type(cadena))
```

```
<class 'str'>
```

```
len(cadena)
```

```
27
```

```
cadena[0]
```

```
'\n'
```

```
cadena[-1]
```

```
'\n'
```

```
cadena[5] = 'h'
```

TypeError: 'str' object does not support item assignment

4.4 Acceso a porciones de las cadenas (*slicing*)

Se puede obtener una subcadena a partir de la cadena original. La sintaxis es la siguiente:

```
cadena[Start:End:Stride]
```

Start: Índice del primer carácter para formar la subcadena.

End: Índice (menos uno) que indica el carácter final de la subcadena.

Stride: Salto entre elementos.

```
ejemplo[:] # Cadena completa
```

```
'Murciélago'
```

```
ejemplo[0:5] # Elementos del 0 al 4
```

```
'Murci'
```

```
ejemplo[::2] # Todos los elementos, con saltos de 2
```

```
'Mrilg'
```

```
ejemplo[1:8:2] # Los elementos de 1 a 7, con saltos de 2
```

```
'ucéa'
```

```
ejemplo[::-1] # La cadena en reversa
```

```
'ogaléicruM'
```

4.5 Operaciones básicas con cadenas

Los operadores: `+` y `*` están definidos para las cadenas.

```
'Luis' + ' ' + 'Miguel' # Concatenación
```

```
'Luis Miguel'
```

```
'ABC' * 3 # Repetición
```

```
'ABCABCABC'
```

4.6 Funciones aplicables sobre las cadenas

Existen métodos definidos que se pueden aplicar a las cadenas. Véase [Common string operations](#) para más información.

```
ejemplo = 'murcielago'
```

```
ejemplo.capitalize()
```

```
'Murcielago'
```

```
print(ejemplo)
print(ejemplo.center(20, '-'))
print(ejemplo.upper())
print(ejemplo.find('e'))
print(ejemplo.count('g'))
print(ejemplo.isprintable())
```

```
murcielago
-----murcielago-----
MURCIELAGO
5
1
True
```

4.7 Construcción de cadenas con variables

```
edad = 15
nombre = 'Pedro'
apellido = 'Páramo'
peso = 70.5
```

Concatenación y casting.

```
datos = nombre + apellido + 'tiene' + str(15) + 'años y pesa ' + str(70.5)
datos
```

```
'PedroPáramotiene15años y pesa 70.5'
```

Método `format()`

```
datos = '{} {} tiene {} años y pesa {}'.format(nombre, apellido, edad, peso)
datos
```

```
'Pedro Páramo tiene 15 años y pesa 70.5'
```

Cadenas formateadas (*f-string*, *formatted string literals*)


```
datos = f'{nombre} {apellido} tiene {edad} años y pesa {peso}'
datos
```

```
'Pedro Páramo tiene 15 años y pesa 70.5'
```


10 Funciones lambda.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

11 Programación funcional.

- Paradigma de programación basado en el uso de funciones, entendiendo el concepto de función según su definición matemática, y no como los subprogramas de los lenguajes imperativos.
- Tiene sus raíces en el cálculo lambda (un sistema formal desarrollado en los años 1930 para investigar la definición de función, la aplicación de las funciones y la recursión).
- Muchos lenguajes de programación funcionales pueden ser vistos como elaboraciones del cálculo lambda.
- Las funciones que se usan en este paradigma son *funciones puras*, es decir, que no tienen efectos secundarios, que no manejan datos mutables o de estado.
- Lo anterior está en contraposición con la programación imperativa.
- Uno de sus principales representantes es el lenguaje Haskell, que compite en belleza, elegancia y expresividad con Python.
- Los programas escritos en un estilo funcional son más fáciles de probar y depurar.
- Por su característica modular facilita el cómputo concurrente y paralelo.
- El estilo funcional se lleva muy bien con los datos, permitiendo crear algoritmos y programas más expresivos para trabajar en *Big Data*.

12 Lambda expressions

- Una expresión Lambda (*Lambda expressions*) nos permite crear una función “anónima”, es decir podemos crear funciones *ad-hoc*, **sin** la necesidad de definir una función propiamente con el comando **def**.
- Una expresión Lambda o función anónima, es una expresión simple, no un bloque de declaraciones.
- Solo hay que escribir el resultado de una expresión en vez de regresar un valor explícitamente.
- Dado que se limita a una expresión, una función anónima es menos general que una función normal **def**.

Por ejemplo, para calcular el cuadrado de un número podemos escribir la siguiente función:

```
def square_v1(n):  
    """  
    Calcula el cuadrado de n y lo regresa. Versión 1.0  
    """  
    result = n**2  
    return result
```

```
print(square_v1(5))
```

Se puede reducir el código anterior como sigue:

```
def square_v2(n):  
    """  
    Calcula el cuadrado de n y lo regresa. Versión 2.0  
    """  
    return n**2
```

```
print(square_v2(5))
```

Se puede reducir aún más, pero puede llevarnos a un mal estilo de programación. Por ejemplo:

```
def square_v3(n): return n**2
```

```
print(square_v3(5))
```

Definición. La sintáxis de una expresión lambda en Python (función lambda o función anónima) es muy simple:

```
lambda argument_list: expression
```

1. La lista de argumentos consiste de objetos separados por coma.
2. La expresión es cualquiera que sea válida en Python.

Se puede asignar la función a una etiqueta para darle un nombre.

12.1 Ejemplo 1.

Función anónima para el cálculo del cuadrado de un número.

```
# Se crea una función anónima  
lambda n: n**2
```

Para poder usar la función anterior debe estar en un contexto donde pueda ser ejecutada o podemos darle un nombre como sigue:

```
# La función anónima se llama ahora cuadrado()  
cuadrado = lambda num: num**2
```

```
# Usamos la función cuadrado()  
print(cuadrado(7))
```

12.2 Ejemplo 2.

Escribir una función lambda para calcular el cubo de un número usando la función lambda que calcula el cuadrado.

```
# Construimos la función cubo() usando la función cuadrado()  
cubo = lambda n: cuadrado(n) * n
```

```
cubo(5)
```

12.3 Ejemplo 3.

Construir una función que genere funciones para elevar un número *a* a una potencia *n*.

Este ejemplo nos permite mostrar que es posible combinar la definición de funciones normales de Python con las funciones lambda.

```
def potencia(n):  
    return lambda a: a ** n # regresa una función lambda
```

```
# Creamos dos funciones.  
cuadrado = potencia(2) # función para elevar al cuadrado  
cubo = potencia(3) # función para elevar al cubo
```

```
print(cuadrado(5))  
print(cubo(2))
```

12.4 Ejemplo 4.

Escribir una función lambda para multiplicar dos números.

En este ejemplo vemos como una función lambda puede recibir dos argumentos.

```
mult = lambda a, b: a * b
```

```
print(mult(5,3))
```

12.5 Ejemplo 5.

Checar si un número es par.

En este ejemplo usamos el operador ternario para probar una condición.

```
esPar = lambda n: False if n % 2 else True
```

```
print(esPar(2))  
print(esPar(3))
```

12.6 Ejemplo 6.

Obtener el primer y último elemento de una secuencia, la cual puede ser una cadena, una lista y una tupla.

```
primer_ultimo= lambda s: (s[0], s[-1])
```

```
# Cadena  
primer_ultimo('Pythonico')
```

```
# Lista  
primer_ultimo([1,2,3,4,5,6,7,8,9])
```

```
# Tupla
primer_ultimo( (1.2, 3.4, 5.6, 8.4) )
```

12.7 Ejemplo 7.

Escribir en reversa una secuencia que puede ser una cadena, una lista y una tupla.

```
c = 'Pythonico'

reversa = lambda l: l[::-1]

print(c)
print(reversa(c))
```

13 Funciones puras e impuras

- La programación funcional busca usar funciones *puras*, es decir, que no tienen efectos secundarios, no manejan datos mutables o de estado.
- Estas funciones puras devuelven un valor que depende solo de sus argumentos.

Por ejemplo, podemos construir funciones que hagan un cálculo aritmético el cuál solo depende de sus entradas y no modifica otra cosa:

```
# La siguiente es una función pura
def pura(x, y):
    return (x + 2 * y) / (2 * x + y)

pura(1,2)
```

1.25

```
# La siguiente es una función lambda pura
lambda_pura = lambda x,y: (x + 2 * y) / (2 * x + y)

lambda_pura(1,2)
```

1.25

El que sigue es un ejemplo de una función impura que tiene efectos colaterales en la [lista](#):

```
# Esta es una función impura
lista = []

def impura(arg):
    potencia = 2
    lista.append(arg) # Se modifica la lista
    return arg ** potencia

impura(5)

print(lista)
```

[5]

Lo anterior también puede suceder usando funciones lambda:

```
# podemos crear funciones lambda impuras :o
lambda_impura = lambda l, arg : (l.append(arg), arg**2)
```

```
print(lambda_impura(lista,5))
lista
```

(None, 25)

[5, 5]


Una buena práctica del estilo funcional es evitar los efectos secundarios, es decir, **que nuestras funciones NO modifiquen los valores de sus argumentos.**



12 Iterables, Mapeo, Filtros y Reducciones.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

13 Iterables

- En Python existen objetos que contienen secuencias de otros objetos (listas, tuplas, diccionarios, etc).
- Estos objetos se pueden recorrer usando ciclos **for ... in ...**.
- A estos objetos se les conoce también como **iterables** (objetos iterables, secuencias iterables, contenedores iterables, conjunto iterable, entre otros).

13.1 Ejemplo 1.

Crear una cadena, una lista, una tupla, un diccionario, un conjunto y leer un archivo; posteriormente recorrer cada uno de estos iterables usando un ciclo **for**:

```
mi_cadena = "pythonico"
mi_lista = ['p','y','t','h','o','n','i','c','o']
mi_tupla = ('p','y','t','h','o','n','i','c','o')
mi_dict = {'p':1,'y':2,'t':3,'h':4,'o':5,'n':6,'i':7,'c':8,'o':9}
mi_conj = {'p','y','t','h','o','n','i','c','o'}
mi_archivo = open("mi_archivo.txt")

print('\nCadena:', end=' ')
# Recorremos la cadena e imprimimos cada elemento
for char in mi_cadena:
    print(char, end=' ')

print('\nLista:', end=' ')
# Recorremos la lista e imprimimos cada elemento
for element in mi_lista:
    print(element, end=' ')

print("\nTupla: ", end='')
# Recorremos la tupla e imprimimos cada elemento
for element in mi_tupla:
    print(element, end=' ')
```

```

print("\nDiccionario (claves): ", end='')
# Recorremos el diccionario e imprimimos cada clave
for key in mi_dict.keys():
    print(key, end=' ')

print("\nDiccionario (valores): ", end='')
# Recorremos el diccionario e imprimimos cada valor
for key in mi_dict.values():
    print(key, end=' ')

print("\nConjunto: ", end='')
# Recorremos el conjunt e imprimimos cada elemento
for s in mi_conj:
    print(s, end = ' ')

print("\nArchivo: ")
# Recorremos el archivo e imprimimos cada elemento
for line in mi_archivo:
    print(line, end = '')

```

Cadena: p y t h o n i c o
 Lista: p y t h o n i c o
 Tupla: p y t h o n i c o
 Diccionario (claves): p y t h o n i c
 Diccionario (valores): 1 2 3 4 9 6 7 8
 Conjunto: y h t c i n o p
 Archivo:
 p
 y
 t
 h
 o
 n
 i
 c
 o

Observa el caso del diccionario y del conjunto: * Diccionario: cuando hay claves repetidas, se sustituye el último valor que toma la clave ('0':9). * Conjunto: los elementos se ordenan, y no se admiten elementos repetidos.

14 Mapeo.

En análisis matemático, un *Mapeo* es una regla que asigna a cada elemento de un primer conjunto, un único elemento de un segundo conjunto:

map

$$\begin{bmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-1} \end{bmatrix} \longrightarrow \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_{n-1} \end{bmatrix}$$

14.1 map

En Python existe la función `map(function, sequence)` cuyo primer parámetro es una función la cual se va a aplicar a una secuencia, la cual es el segundo parámetro. El resultado será una nueva secuencia con los elementos obtenidos de aplicar la función a cada elemento de la secuencia de entrada.

14.2 Ejemplo 2.

Crear el siguiente mapeo con una lista, una tupla, un conjunto

$$f(x) = x^2$$

$$\begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 \\ 1 \\ 4 \\ 9 \\ 16 \end{bmatrix}$$

```
# Primero definimos la función
def square(x):
    """
    Calcula el cuadrado de x.
    """
    return x**2

# Luego definimos las secuencias
l = [0,1,2,3,4]
t = (0,1,2,3,4)
s = {0,1,2,3,4}

# Ahora creamos los mapeos
lmap = map(square, l)
tmap = map(square, t)
smap = map(square, s)

# Checamos el tipo de cada mapeo
print(type(lmap), type(tmap), type(smap))
```

```
print('Lista {}'.format(l))
print('Mapeo {}'.format(list(lmap)))

print('Tupla {}'.format(t))
print('Mapeo {}'.format(tuple(tmap)))

print('Conj {}'.format(s))
print('Mapeo {}'.format(set(smap)))
```

```
<class 'map'> <class 'map'> <class 'map'>
```

```
Lista [0, 1, 2, 3, 4]
```

```
Mapeo [0, 1, 4, 9, 16]
```

```
Tupla (0, 1, 2, 3, 4)
```

```
Mapeo (0, 1, 4, 9, 16)
```

```
Conj {0, 1, 2, 3, 4}
```

```
Mapeo {0, 1, 4, 9, 16}
```

Observa que el resultado del mapeo es un objeto de tipo `<class 'map'>` por lo que debemos convertirlo en un tipo que pueda ser desplegado para imprimir.

14.3 Ejemplo 3.

Crear un mapeo para convertir grados Fahrenheit a Celsius y viceversa:

```
def toFahrenheit(T):
    """
    Transforma los elementos de T en grados Farenheit.
    """
    return (9/5)*T + 32

def toCelsius(T):
    """
    Transforma los elementos de T en grados Celsius.
    """
    return (5/9)*(T-32)
```

Celsius → Fahrenheit

```
# Lista original con los datos
c = [0, 22.5, 40, 100]

# Construimos el mapeo y lo nombramos en `fmap`.
fmap = map(toFahrenheit, c)
```

```
# Imprimos a lista original y el mapeo
print(c)
print(list(fmap))
```

```
[0, 22.5, 40, 100]
[32.0, 72.5, 104.0, 212.0]
```

NOTA. Solo se puede usar el mapeo una vez, si vuelves a ejecutar la celda anterior el resultado del mapeo estará vacío. Para volverlo a generar debes ejecutar la celda donde se construye el mapeo.

Lo anterior se puede realiza en una sola línea: crear el mapeo, convertir a lista e imprimir

```
print(list(map(toFahrenheit,c)))
```

```
[32.0, 72.5, 104.0, 212.0]
```

Fahrenheit → Celsius

```
# Lista original con los datos
f = [32.0, 72.5, 104.0, 212.0]

# Conversión en una sola línea
print(list(map(toCelsius, f)))
```

```
[0.0, 22.5, 40.0, 100.0]
```

14.4 Ejemplo 4.

Crear un mapeo para sumar los elementos de tres listas que contienen números enteros.

NOTA. La función `map()` se puede aplicar a más de un conjunto iterable, siempre y cuando los iterables tengan la misma longitud y la función que se aplique tenga los parámetros correspondientes.

```
def suma(x,y,z):
    """
    Suma los números x, y, z.
    """
    return x+y+z

# Tres listas con enteros
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]
```

```
# Aplicación del mapeo
print(list(map(suma, a,b,c)))
```

[15, 18, 21, 24]

15 Filtrado.

- Filtrar es un procedimiento para seleccionar cosas de un conjunto o para impedir su paso libremente.
- En matemáticas, un filtro es un subconjunto especial de un conjunto parcialmente ordenado.

filter

$$\begin{array}{ccc} \left[\begin{array}{c} s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_{n-1} \end{array} \right] & \begin{array}{c} \text{True} \\ \rightarrow \\ \\ \text{True} \\ \rightarrow \\ \\ \text{True} \\ \rightarrow \end{array} & \left[\begin{array}{c} - \\ f_1 \\ - \\ f_2 \\ f_{m-1} \end{array} \right] \end{array}$$

15.1 filter.

En Python existe la función `filter(function, sequence)` cuyo primer parámetro es una función la cual se va a aplicar a una secuencia, la cual es el segundo parámetro. La función debe regresar un objeto de tipo Booleano: `True` o `False`. El resultado será una nueva secuencia con los elementos obtenidos de aplicar la función a cada elemento de la secuencia de entrada.

15.2 Ejemplo 5.

Usando la función `filter()`, encontrar los números pares en una lista.

```
def esPar(n):
    """
    Función que determina si un número es par o impar.
    """
    if n%2 == 0:
        return True
    else:
        return False
```

```
# Probamos la función
print(esPar(10))
```

```
print(esPar(9))
```

True
False

```
# Creamos una lista de números, del 0 al 19
numeros = list(range(20))
print(numeros)
```

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

```
# Aplicamos el filtro
print(list(filter(esPar, numeros)))
```

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

15.3 Ejemplo 6.

Encontrar los números pares en una lista que contiene elementos de muchos tipos.

Paso 1. Creamos la lista.

```
# Creamos la lista
lista = ['Hola', 4, 3.1416, 3, 8, ('a', 2), 10, {'x': 1.5, 'y': 12} ]
print(lista)
```

['Hola', 4, 3.1416, 3, 8, ('a', 2), 10, {'x': 1.5, 'y': 12}]

Paso 2. Escribimos una función que verifique si una entrada es de tipo `int`.

```
def esEntero(i):
    """
    Función que determina si un número es entero.
    """
    if isinstance(i, int): # Checamos si la entrada es de tipo int
        return True
    else:
        return False
```

```
print(esEntero("Hola"))
print(esEntero(1))
print(esEntero(1.))
```

False
True
False

Una forma alternativa, más Pythonica, de construir la función `esEntero()` es la siguiente:

```
def esEntero(i):
    return True if isinstance(i, int) else False
```

Paso 3. Usamos la función `esPar()` para encontrar los pares de la lista.

```
print(list(filter(esPar, list(filter(esEntero, lista)))))
```

[4, 8, 10]

Observa que se aplicó dos veces la función `filter()`, la primera para determinar si el elemento de la lista es entero usando la función `esEntero()`, la segunda para determinar si el número es par.

15.4 Ejemplo 7.

Encontrar los números primos en el conjunto $\{2, \dots, 50\}$.

```
# Función que crear una
def noPrimo():
    """
    Determina la lista de números que no son primos en el
    rango [2, 50]
    """
    np_list = []
    for i in range(2, 50):
        for j in range(i*2, 50, i):
            np_list.append(j)
    return np_list

no_primo = noPrimo()

print("Lista de números NO primos en el rango [2, 50] \n{}".format(no_primo))

def esPrimo(number):
    """
    Determina si un número es primo o no.
    """
    np_list = noPrimo()
    if(number not in np_list):
        return True
```

```
# Creación de la lista de números enteros de 2 a 50
numeros = list(range(2,50))

# Calculamos los primos usando filter(), con
# la función esPrimo() y la lista números.
primos = list(filter(esPrimo, numeros))

print("\nNúmeros primos en el rango [2, 50] \n {}".format(primos))
```

Lista de números NO primos en el rango [2, 50]

[4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 10, 15, 20, 25, 30, 35, 40, 45, 12, 18, 24, 30, 36, 42, 48, 14, 21, 28, 35, 42, 49]

Números primos en el rango [2, 50]

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

16 Reducción.

- **Reducción** : Disminuir *algo* en tamaño, cantidad, grado, importancia, ..
- La operación de reducción es útil en muchos ámbitos, el objetivo es reducir un conjunto de objetos en un objeto más simple.

Una reducción se hace como sigue:

Dada la función $f()$ y la secuencia $[s_1, s_2, s_3, s_4]$ se tiene que

$$\begin{array}{ccc} \left[\underbrace{s_1, s_2}_{a=f(s_1, s_2)}, s_3, s_4 \right] & \implies & f(\underbrace{f(\underbrace{f(s_1, s_2), s_3}_b), s_4}_c) \\ \underbrace{b=f(a, s_3)} & & \underbrace{a} \\ \underbrace{c=f(b, s_4)} & & \underbrace{b} \end{array}$$

16.1 filter.

En Python existe la función `reduce(function, sequence)` cuyo primer parámetro es una función la cual se va a aplicar a una secuencia, la cual es el segundo parámetro. La función debe regresar un objeto que es el resultado de la reducción.

16.2 Ejemplo 8.

Calcular la siguiente serie:

$$1 + 2 + \dots + n = \sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Si $n = 4$ entonces $1+2+3+4 = 10$

```
# La función reduce() debe importarse del módulo functools
from functools import reduce
```

```
# Creamos la lista
nums = [1,2,3,4]
print(nums)

# Calculamos la serie usando reduce y una función lambda
suma = reduce(lambda x, y: x + y, nums)
print(suma)
```

```
[1, 2, 3, 4]
10
```

```
# Se pueden usar arreglos de numpy
import numpy as np

# Construimos un arreglo de 1's
a = np.ones(20)
print(a)

suma = reduce(lambda x, y: x + y, a)
print(suma)
```

```
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
20.0
```

16.3 Ejemplo 9.

Calcular la siguiente serie:

$$\sum_{i=1}^n \frac{1}{i} = \sum_{i=1}^n \frac{1}{i}$$

```
numeros = [3,4,5]
result = reduce(lambda x, y: 1/x + 1/y, numeros)
print(result)
```

```
1.9142857142857144
```


16.4 Ejemplo 10.

Calcular el máximo de una lista de números.

```
numeros = [23,5,23,56,87,98,23]
maximo = reduce(lambda a,b: a if (a > b) else b, numeros)
print(maximo)
```

98

16.5 Ejemplo 11.

Calcular el factorial de un número.

17 Más ejemplos Pythonicos.

17.1 Ejemplo 12.

Convertir grados Fahrenheit a Celsius y viceversa combinando `map()` con `lambda`.

```
c = [0, 22.5, 40,100]

# Conversión a Fahrenheit
f = list(map(lambda T: (9/5) * T + 32, c))
print(f)

# Conversión a Celsius
print(list(map(lambda T: (5/9)*(T - 32), f)))
```

[32.0, 72.5, 104.0, 212.0]

[0.0, 22.5, 40.0, 100.0]

17.2 Ejemplo 13.

Sumar tres arreglos combinando `map()` con `lambda`.

```
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

print(list(map(lambda x,y,z : x+y+z, a,b,c)))
```

[15, 18, 21, 24]

17.3 Ejemplo 14.

Encontrar todos los números pares de una lista combinando `filter()` con `lambda`.

```
# Lista de números
nums = [0, 2, 5, 8, 10, 23, 31, 35, 36, 47, 50, 77, 93]

# Aplicación de filter y lambda
result = filter(lambda x : x % 2 == 0, nums)

print(list(result))
```

[0, 2, 8, 10, 36, 50]

17.4 Ejemplo 15.

Encontrar todos los números primos en el conjunto $\{2, \dots, 50\}$ combinando combinando `filter()` con `lambda`.

```
# Lista de números de 2 a 50
nums = list(range(2, 51))

# Cálculo de los números primos usando
# filter y lambda
for i in range(2, 8):
    nums = list(filter(lambda x: x == i or x % i, nums))

print(nums)
```

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

17.5 Ejemplo 16.

Contar el número de caracteres de un texto, combinando `reduce()`, `map()` y `lambda`.

```
# Contar los caracteres de una cadena
texto = 'Hola Mundo'

palabras = texto.split()
print(palabras)

# Conteo de caracteres
print(reduce(lambda x,y: x+y, list(map(lambda palabras: len(palabras), palabras))
```

['Hola', 'Mundo']

9

```
# Contar los caracteres de un texto en un archivo
archivo = open('QueLesQuedaALosJovenes.txt', 'r')

suma = 0
for linea in archivo:
    palabras = linea.split()
    suma += reduce(lambda x,y: x+y, list(map(lambda palabras: len(palabras), palabras))
print(suma)
archivo.close()
```

824

Lo anterior se puede realizar si construir una función que cuenta los caracteres de una lista de cadenas:

```
def cuentaCaracteres(palabras):
    return reduce(lambda x,y: x+y, list(map(lambda palabras: len(palabras), palabras))
```

```
texto = 'Hello Motto'.split()
cuentaCaracteres(texto)
```

10

```
# Contar los caracteres de un texto en un archivo
archivo = open('QueLesQuedaALosJovenes.txt', 'r')

suma = 0
```

```
for linea in archivo:
    palabras = linea.split()
    suma += cuentaCaracteres(palabras)
print(suma)
archivo.close()
```

824

17.6 Ejemplo 17.

La siguiente función es impura porque modifica la `lista`:

```
def cuadradosImpuros(lista):
    for i, v in enumerate(lista):
        lista[i] = v ** 2
    return lista

numeros_originales = list(range(5))
print(numeros_originales)
print(cuadradosImpuros(numeros_originales))
print(numeros_originales)
```

La salida del código anterior es el siguiente:

```
[0, 1, 2, 3, 4]
[0, 1, 4, 9, 16]
[0, 1, 4, 9, 16]
```

Escribe una versión pura de la función `cuadradosImpuros(lista)` usando `map()` y `lambda`.

Una manera alternativa es la siguiente:

```
numeros_originales = list(range(5))
print(numeros_originales)
print(list(map(lambda x: x ** 2, numeros_originales)))
print(numeros_originales)
```


```
[0, 1, 2, 3, 4]
[0, 1, 4, 9, 16]
[0, 1, 2, 3, 4]
```



8 Gestión de archivos y gestores de contexto.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

9 Gestión de archivos

La función para gestionar archivos es `open()`. Toma dos parámetros: el nombre del archivo y el modo. Existen cuatro diferentes modos:

- `"r"` - Read - Default value. Abre el archivo para lectura. Se produce un error si el archivo no existe.
- `"a"` - Append - Abre el archivo para agregar información. Si el archivo no existe, lo crea.
- `"w"` - Write - Abre el archivo para escritura. Si el archivo no existe, lo crea. Si el archivo existe, lo sobrescribe.
- `"x"` - Create - Crea el archivo, regresa un error si el archivo existe.

Adicionalmente se puede especificar si el archivo se abre en modo texto o binario:

- `"t"` - Text - Valor por omisión.
- `"b"` - Binary

```
# Abrimos el archivo gatos.txt en modo escritura
# Ojo: si el archivo existe, lo sobrescribe.
f = open('gatos.txt', 'w')
```

```
# Construimos un diccionario con información
gatos = ['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
peso_minimo = [2.3, 3.5, 5.4, 2.5]
dicc = dict(zip(gatos, peso_minimo))
print(dicc)
```

```
{'Persa': 2.3, 'Sphynx': 3.5, 'Ragdoll': 5.4, 'Siamés': 2.5}
```

```
# Guardamos cada elemento del diccionario en el archivo
for i in dicc:
    f.write('{:>10} {:>10} \n'.format(i, dicc[i]))

# Es importante siempre cerrar el archivo cuando ya no se use.
f.close()
```

```
# Ahora abrimos el archivo en modo lectura
# El archivo debe existir, de otro modo se genera un error.
```

```
f = open('gatos.txt', 'r')
for i in f:
    print(i)

f.close()
```

Persa	2.3
Sphynx	3.5
Ragdoll	5.4
Siamés	2.5

10 Gestores de contexto

Permiten asignar y liberar recursos justo en el momento requerido. Se usa mayormente con `with`, veamos un ejemplo:

```
with open('contexto_ejemplo', 'w') as archivo_abierto:
    archivo_abierto.write('En este ejemplo, todo se realiza con un gestor de cont
```

Lo que realiza la operación anterior es: 1. Abre el archivo `contexto_ejemplo`. 2. Escribe algo en el archivo, 3. Cierra el archivo. Si ocurre algún error mientras se escribe en el archivo, entonces se intenta cerrar el archivo.

El código es equivalente a:


```
archivo_abierto = open('contexto_ejemplo', 'w')
try:
    archivo_abierto.write('En este ... contexto!')
finally:
    archivo_abierto.close()
```

Es posible implementar nuestros propios gestores de contexto. Para ello se requiere un conocimiento más profundo de Programación Orientada a Objetos.

15 Decoradores.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

16 Definición.

- Se denomina decorador a la persona dedicada a diseñar el interior de oficinas, viviendas o establecimientos comerciales con criterios estéticos y funcionales.
- En Python, un decorador es una función para modificar otra función.
 - Recibe una función.
 - Regresa otra función.
- Los decoradores son herramientas bonitas y útiles de Python.

16.1 Ejemplo 1.

La función `print_hello()` imprime `Hola mundo pythonico`.

```
def print_hello():  
    print('{:^30}'.format('Hola mundo pythonico'))
```

Crear un decorador que agregue colores al mensaje.

```
def print_hello():  
    print('{:^30}'.format('Hola mundo pythonico'))
```

```
# Uso normal de la función  
print_hello()
```

Hola mundo pythonico

```
from colorama import Fore, Back, Style  
  
# Decorador  
def mi_decorador1(f):  
  
    # La función que hace el decorado.
```

```
def envoltura():
    linea = '-' * 30
    print(Fore.BLUE + linea + Style.RESET_ALL)
    print(Back.GREEN + Fore.WHITE, end='')

    f() # Ejecución de la función

    print(Style.RESET_ALL, end='')
    print(Fore.BLUE + linea + Style.RESET_ALL)

# Regresamos la función decorada
return envoltura

# Decorando la función.
print_hello_colored = mi_decorador1(print_hello) # Funcion decorada

# Ahora se ejecuta la función decorada.
print_hello_colored()
```

```
-----
Hola mundo pythonico
-----
```

16.2 Ejemplo 2.

La función `print_message(m)` imprime el mensaje que recibe como parámetro.

```
def print_message(m):
    print('{:^30}'.format(m))
```

Modificar el decorador creado en el ejemplo 1 para que se pueda recibir el parámetro `m`.

```
# Decorador
def mi_decorador2(f):

    # La función que hace el decorado.
    # Ahora recibe un parámetro
    def envoltura(m):
        linea = '-' * 30
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print(Back.GREEN + Fore.WHITE, end='')

        f(m) # Ejecución de la función

        print(Style.RESET_ALL, end='')
        print(Fore.BLUE + linea + Style.RESET_ALL)
```



```
# Regresamos la función decorada
return envoltura
```

```
# La función se puede decorar en su definición como sigue
@mi_decorador2
def print_message(m):
    print('{:^30}'.format(m))
```

```
# Entonces se puede usar la función con su nombre original
print_message('bueno, bonito y barato')
```

```
-----
bueno, bonito y barato
-----
```

16.3 Ejemplo 3.

Decorar las funciones `sin()` y `cos()` de la biblioteca `math`.

```
def mi_decorador3(f):

    def coloreado(x):

        # Construimos una cadena coloreada con el
        # resultado de la evaluación de f(x)
        res = Fore.GREEN + f.__name__
        res += '(' + Style.BRIGHT + str(x) + Style.RESET_ALL + Fore.GREEN + ')' =
        res += f'{f(x)}'

        # Imprimimos el resultado
        linea = '-' * 80
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print('{:^80}'.format(res))
        print(Fore.BLUE + linea + Style.RESET_ALL)

    return coloreado

from math import sin, cos

sin = mi_decorador3(sin)
cos = mi_decorador3(cos)

for f in [sin, cos]:
    f(3.141596)
```

```
sin(3.141596) = -3.3464102065883993e-06
```

```
cos(3.141596) = -0.99999999999944008
```

16.4 Ejemplo 4.

Decorar funciones con un número variable de argumentos.

```
from random import random, randint, choice, choices

def mi_decorador4(f):
    def envoltura(*args, **kwargs):

        # Construimos una cadena coloreada con el
        # resultado de la evaluación de f(x)
        res = Fore.GREEN + f.__name__
        res += '(' + Style.BRIGHT + f'{args},{kwargs}' + Style.RESET_ALL + Fore.C
        res += f'{f(*args, **kwargs)}'

        # Imprimimos el resultado
        linea = '-' * 80
        print(Fore.BLUE + linea + Style.RESET_ALL)
        print('{:^80}'.format(res))
        print(Fore.BLUE + linea + Style.RESET_ALL)

    return envoltura

random = mi_decorador4(random)
randint = mi_decorador4(randint)
choice = mi_decorador4(choice)
choices = mi_decorador4(choices)

random()
randint(3, 8)
choice([4, 5, 6])

p = [x for x in range(10)]
choices(p, k=3)
```

```
random((),{}) = 0.4390656899525458
```

```
randint((3, 8),{}) = 3
```

```
choice([4, 5, 6], {}, {}) = 5
```

```
choices([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], {}, {'k': 3}) = [5, 3, 9]
```

16.5 Ejemplo 5.

Crear un decorador que calcule el tiempo de ejecución de una función.

```
import time

def crono(f):
    """
    Regresa el tiempo que toma en ejecutarse la funcion.
    """
    def tiempo():
        t1 = time.perf_counter()
        f()
        t2 = time.perf_counter()
        return 'Elapsed time: ' + str((t2 - t1)) + "\n"
    return tiempo

@crono
def miFuncion():
    numeros = []
    for num in (range(0, 10000)):
        numeros.append(num)
    print('\nLa suma es: ' + str((sum(numeros))))

print(miFuncion())
```

La suma es: 49995000

Elapsed time: 0.0012546591460704803

16.6 Ejemplo 6.

Detener la ejecución por un tiempo antes que una función sea ejecutada.

```

from time import sleep

def sleepDecorador(function):

    def duerme(*args, **kwargs):
        sleep(1)
        return function(*args, **kwargs)
    return duerme

@sleepDecorador
def imprimeNumero(num):
    return num

for num in range(1, 6):
    print(imprimeNumero(num), end = ' ')

print('\n --> happy finish!')

```

```

1 2 3 4 5
--> happy finish!

```

16.7 Ejemplo 7.

Crear un decorador que cheque que el argumento de una función que calcula el factorial, sea un entero positivo.

```

def checaArgumento(f):
    def checador(x):
        if type(x) == int and x > 0:
            return f(x)
        else:
            raise Exception("El argumento no es un entero positivo")
    return checador

@checaArgumento
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)

for i in range(1,10):
    print(i, factorial(i))

```

```
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
```

```
print(factorial(-1))
```

Exception: El argumento no es un entero positivo

16.8 Ejemplo 8.

Contar el número de llamadas de una función.

```
def contadorDeLlamadas(func):

    def cuenta(*args, **kwargs):
        cuenta.calls += 1
        return func(*args, **kwargs)

    # Variable estática que lleva la cuenta
    cuenta.calls = 0

    return cuenta

@contadorDeLlamadas
def suma(x):
    return x + 1

@contadorDeLlamadas
def mulp1(x, y=1):
    return x*y + 1

print('Llamadas a suma = {}'.format(suma.calls))

for i in range(4):
    suma(i)

mulp1(1, 2)
mulp1(5)
mulp1(y=2, x=25)
```

```
print('Llamadas a suma = {}'.format(suma.calls))  
print('Llamadas a multp1 = {}'.format(mulp1.calls))
```

```
Llamadas a suma = 0  
Llamadas a suma = 4  
Llamadas a multp1 = 3
```

16.9 Ejemplo 9.

Decorar una función con diferentes saludos.

```
def buenasTardes(func):  
    def saludo(x):  
        print("Hola, buenas tardes, ", end='')  
        func(x)  
    return saludo  
  
def buenosDias(func):  
    def saludo(x):  
        print("Hola, buenos días, ", end='')  
        func(x)  
    return saludo  
  
@buenasTardes  
def mensaje1(hora):  
    print("son las " + hora)  
  
mensaje1("3 pm")  
  
@buenosDias  
def mensaje2(hora):  
    print("son las " + hora)  
  
mensaje2("8 am")
```

```
Hola, buenas tardes, son las 3 pm  
Hola, buenos días, son las 8 am
```

16.10 Ejemplo 10.

El ejemplo anterior se puede realizar como sigue:

```
def saludo(expr):  
    def saludoDecorador(func):  
        def saludoGenerico(x):  
            print(expr, end='')  
            func(x)  
        return saludoGenerico  
    return saludoDecorador  
  
@saludo("Hola, buenas tardes, ")  
def mensaje1(hora):  
    print("son las " + hora)  
  
mensaje1("3 pm")  
  
@saludo("Hola, buenos días, ")  
def mensaje2(hora):  
    print("son las " + hora)  
  
mensaje2("8 am")  
  
@saludo("καλημερα ")  
def mensaje3(hora):  
    print(" <--- en griego " + hora)  
  
mensaje3(" :D ")
```

Hola, buenas tardes, son las 3 pm


Hola, buenos días, son las 8 am

καλημερα <--- en griego :D

6 Control de flujo.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

En Python existen declaraciones que permiten controlar el flujo de un programa para realizar acciones complejas. Entre estas declaraciones tenemos las siguientes:

- `while`
- `for`
- `if`
- `match`

Junto con estas declaraciones generalmente se utilizan las siguientes operaciones lógicas cuyo resultado puede ser `True` o `False`:

Python	Significado
<code>a == b</code>	¿son iguales <code>a</code> y <code>b</code> ?
<code>a != b</code>	¿son diferentes <code>a</code> y <code>b</code> ?
<code>a < b</code>	¿ <code>a</code> es menor que <code>b</code> ?:
<code>a <= b</code>	¿ <code>a</code> es menor o igual que <code>b</code> ?
<code>a > b</code>	¿ <code>a</code> es mayor que <code>b</code> ?
<code>a >= b</code>	¿ <code>a</code> es mayor o igual que <code>b</code> ?
<code>not A</code>	El inverso de la expresión <code>A</code>
<code>A and B</code>	¿La expresión <code>A</code> y la expresión <code>B</code> son verdaderas?
<code>A or B</code>	¿La expresión <code>A</code> o la expresión <code>B</code> es verdadera?:

7 while

Se utiliza para repetir un conjunto de instrucciones mientras una expresión sea verdadera:

```
while expresión:  
    código ...
```


Por ejemplo:

```
a = 0 # Inicializamos a en 0

print('Inicia while') # Instrucción fuera del bloque while

while a < 5: # Mientras a sea menor que 5 realiza lo siguiente:
    print(a) # Imprime el valor de a
    a += 1   # Incrementa el valor de a en 1

print('Finaliza while') # Instrucción fuera del bloque while
```

- Como se observa, el código después de `while` tiene una sangría (*indentation*): las líneas de código están recorridas hacia la derecha.
- Este espacio en blanco debe ser al menos de uno, pero pueden ser más.
- Por omisión, en JupyterLab (y algunos otros editores, se usan 4 espacios en blanco para cada línea de código dentro del bloque.
- El número de espacios en blanco se debe mantener durante todo el bloque de código.
- Cuando termina el sangrado, es decir las líneas de código ya no tienen ningún espacio en blanco al inicio, se cierra el bloque de código, en este caso el `while`.
- El uso de una sangría para organizar los bloques de código lo hace Python para que el código sea más entendible.
- **Ejemplos válidos:**

```
while a < 5:
    print(a)
    a += 1
```

```
while a < 5:
    print(a)
    a += 1
```

- **Ejemplos NO válidos**

```
while a < 5:
    print(a)
a += 1
```

```
while a < 5:
print(a)
a += 1
```

7.1 Ejemplo 1.

Los números de Fibonacci, denotados con F_n forman una secuencia tal que cada número es la suma de dos números precedentes e inicia con el 0 y el 1. Matemáticamente se escribe como:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_n &= F_{n-1} + F_{n-2} \quad \text{para } n > 1.\end{aligned}$$

La secuencia es entonces: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Vamos a calcular esta secuencia usando la instrucción `while`:

```
a, b = 0, 1 # Definimos los primeros dos elementos:

while a < 1000:      # Mientras a sea menor que 1000 realiza lo siguiente:
    print(a, end=', ') # Imprime a y b (separados por una coma)
    a, b = b, a+b     # Calcula los siguientes dos elementos
```

8 if, elif, else

Esta declaración permite ejecutar un código dependiendo del resultado de una o varias expresiones lógicas. La estructura es como sigue:

```
if expresion1:
    codigo1 ...
elif expresion2:
    codigo2 ...
elif expresion3:
    codigo3 ...
else:
    codigo4
```

Si la `expresion1` es verdadera, entonces se ejecuta el `codigo1`, en otro caso se evalúan las siguientes expresiones y dependiendo de cuál es verdadera se ejecuta el código correspondiente. Cuando ninguna de las expresiones es verdadera, entonces se ejecuta el código de la sección `else`, es decir el `codigo4`.

Observa que se siguen las mismas reglas de sangrado que en el `while`.

Veamos un ejemplo:

```
# Modifica los valores de a y b, y observa el resultado
a = 10
b = 20
```

```
if a < b:
    print('a es menor que b')
elif a > b:
    print('a es mayor que b')
elif a == b:
    print('a es igual a b')
else:
    print('Esto nunca pasa')
```

Las expresiones pueden ser más complejas:

```
# Modifica los valores de a y b, y observa el resultado
a = 10
b = 20
if (a < b) or (a > b):
    print(f'a = {a}, b = {b}')
```

9 Operador ternario

Este operador permite evaluar una expresión lógica y generar un valor para un resultado **True** y otro diferente para un resultado **False**; todo esto se logra en una sola línea de código como sigue:

```
resultado = valor1 if expresion else valor2
```

Por ejemplo:

```
# Usa valores para c = 1, 2, 4, 4, 5, 6, 20 y observa el resultado
c = 1
r = c if c > 5 else 0
print(r)
```

10 for

Permite iterar sobre el contenido de cualquier secuencia (cadena, lista, tupla, conjunto, diccionario, archivo, ...). La forma de esta declaración es como sigue:

```
for i in secuencia:
    codigo
```

Las reglas de sangrado se siguen en esta declaración.

Por ejemplo:

```
gatos = ['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
```

```
for i in gatos:
    print(i)
```

10.1 Función zip

La función `zip(s1, s2, ...)` permite combinar dos o más secuencias en una sola; genera tuplas con los elementos de: las secuencias y va iterando sobre ellas.

Por ejemplo

```
# Dos listas de la misma longitud
gatos = ['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
origen = ['Irán', 'Toronto', 'California', 'Tailandia']
print(gatos)
print(origen)

# Combinamos las listas en una sola secuencia
print('\n(Raza, Origen)')
print('-'*20)
for t in zip(gatos, origen):
    print(t)
```

```
['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
['Irán', 'Toronto', 'California', 'Tailandia']
```

```
(Raza, Origen)
```

```
-----
('Persa', 'Irán')
('Sphynx', 'Toronto')
('Ragdoll', 'California')
('Siamés', 'Tailandia')
```

```
# Se puede extraer la información de cada secuencia:
for g, o in zip(gatos, origen):
    print('La raza {} proviene de {}'.format(g, o))
```

```
La raza Persa proviene de Irán
La raza Sphynx proviene de Toronto
La raza Ragdoll proviene de California
La raza Siamés proviene de Tailandia
```

10.2 Conversión de zip a list, tuple, set, dict

Estrictamente `zip` es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo `zip` a alguna otra secuencia básica de datos de Python.

```
z = zip(gatos, origen)

# Verificar el tipo de zip
print(type(z))
print(z)
```

```
<class 'zip'>
<zip object at 0x7f0598954a40>
```

```
lista = list(zip(gatos, origen))
tupla = tuple(zip(gatos, origen))
conj = set(zip(gatos, origen))
dicc = dict(zip(gatos, origen)) # Solo funciona para dos secuencias

print(lista)
print(tupla)
print(conj)
print(dicc)
```

```
[('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Ragdoll', 'California'), ('Siamés',
'Tailandia')]
(('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Ragdoll', 'California'), ('Siamés',
'Tailandia'))
{('Persa', 'Irán'), ('Sphynx', 'Toronto'), ('Siamés', 'Tailandia'), ('Ragdoll',
'California')}
```

```
{'Persa': 'Irán', 'Sphynx': 'Toronto', 'Ragdoll': 'California', 'Siamés': 'Tailandia'}
```

10.3 Función **enumerate**

Permite enumerar los elementos de una secuencia. Genera tuplas con el número del elemento y el elemento de la secuencia.

Por ejemplo:

```
print(gatos)

# Enumeramos la secuencia
print('\n(Numero, Raza)')
print('-'*20)

for t in enumerate(gatos):
    print(t)
```

```
['Persa', 'Sphynx', 'Ragdoll', 'Siamés']
```

```
(Numero, Raza)
```

```
-----
```

```
(0, 'Persa')  
(1, 'Sphynx')  
(2, 'Ragdoll')  
(3, 'Siamés')
```

```
for i, g in enumerate(gatos):  
    print(i, g)
```

```
0 Persa  
1 Sphynx  
2 Ragdoll  
3 Siamés
```

Lo anterior permite usar el indexado para acceder a los elementos de una secuencia:

```
for i, g in enumerate(gatos):  
    print(i, gatos[i])
```

```
0 Persa  
1 Sphynx  
2 Ragdoll  
3 Siamés
```

10.4 Conversión de `enumerate` a `list`, `tuple`, `set`, `dict`

Estrictamente `enumerate` es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo `enumerate` a alguna otra secuencia básica de datos de Python:

```
e = enumerate(gatos)  
  
# Verificar el tipo de enumerate  
print(type(e))  
print(e)
```

```
<class 'enumerate'>  
<enumerate object at 0x7f0598472d90>
```

```
lista = list(enumerate(gatos))  
tupla = tuple(enumerate(gatos))  
conj = set(enumerate(gatos))  
dicc = dict(enumerate(gatos))  
  
print(lista)  
print(tupla)  
print(conj)  
print(dicc)
```

```
[(0, 'Persa'), (1, 'Sphynx'), (2, 'Ragdoll'), (3, 'Siamés')]
((0, 'Persa'), (1, 'Sphynx'), (2, 'Ragdoll'), (3, 'Siamés'))
{(1, 'Sphynx'), (0, 'Persa'), (3, 'Siamés'), (2, 'Ragdoll')}
```

```
{0: 'Persa', 1: 'Sphynx', 2: 'Ragdoll', 3: 'Siamés'}
```

10.5 Funcion range

Esta función genera una secuencia iterable con un inicio, un final y un salto:

```
range(start, stop, step)
```

La secuencia irá desde **start** hasta **stop-1** en pasos de **step**. Por ejemplo:

```
for i in range(1,20): # Por omisión step = 1
    print(i, end= ', ')
```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,

```
for i in range(1,20, 2):
    print(i, end= ', ')
```

1, 3, 5, 7, 9, 11, 13, 15, 17, 19,

```
for i in range(20, 1, -2): # El paso puede ser negativo
    print(i, end= ', ')
```

20, 18, 16, 14, 12, 10, 8, 6, 4, 2,

Usando **range()** se puede acceder a una secuencia mediante el indexado:

```
N = len(gatos) # Longitud de la lista gatos

for i in range(0, N):
    print(i, gatos[i])
```

```
0 Persa
1 Sphynx
2 Ragdoll
3 Siamés
```

10.6 Conversión de **range** a **list**, **tuple**, **set**

Estrictamente **range** es una clase que define un tipo dentro de Python, por lo que es posible convertir del tipo **range** a alguna otra secuencia básica de datos de Python:

```
N = len(gatos) # Longitud de la lista gatos
r = range(0,N)

# Verificar el tipo de range
print(type(r))
print(r)
```

```
<class 'range'>
range(0, 4)
```

```
lista = list(range(0,N))
tupla = tuple(range(0,N))
conj = set(range(0,N))

print(lista)
print(tupla)
print(conj)
```

```
[0, 1, 2, 3]
(0, 1, 2, 3)
{0, 1, 2, 3}
```

10.7 break, continue, else, pass

Estas son palabras clave que se pueden usar en ciclos `while` o `for`: * `break`: terminar el ciclo más interno. * `continue`: saltarse a la siguiente iteración sin terminar de ejecutar el código que sigue. * `else`: **NO** se ejecuta el código de esta cláusula si el ciclo es finalizado por el `break`. * `pass`: no hacer nada y continuar.

Veamos algunos ejemplos:

```
# Se itera por una lista de palabras, cuando se encuentra
# la letra 'h' se termina el ciclo interno y se continua con
# la siguiente palabra.
for palabra in ["Hola", "mundo", "Pythonico"]:
    print('Palabra: ', palabra)
    for letra in palabra:
        print('\t letra: ', letra)
        if letra == "h":
            break
```

```
Palabra:  Hola
    letra:  H
    letra:  o
    letra:  l
    letra:  a
Palabra:  mundo
    letra:  m
```



```

letra: u
letra: n
letra: d
letra: o
Palabra: Pythonico
letra: P
letra: y
letra: t
letra: h

```

```

# Se itera por una lista de palabras, cuando se encuentra
# la letra 'h' se termina el ciclo interno y se continua con
# la siguiente palabra. La cláusula 'else' se ejecuta si no
# se encuentra la letra.
for palabra in ["Hola", "mundo", "Pythonico"]:
    print('Palabra: ', palabra)
    for letra in palabra:
        print('\t letra: ', letra)
        if letra == "h":
            break
    else:
        print('No encontré la letra "h"')

```

```

Palabra: Hola
letra: H
letra: o
letra: l
letra: a
No encontré la letra "h"
Palabra: mundo
letra: m
letra: u
letra: n
letra: d
letra: o
No encontré la letra "h"
Palabra: Pythonico
letra: P
letra: y
letra: t
letra: h

```

```

# Esta declaración pass no hace nada. Se usa principalmente
# para cuestiones de desarrollo de código a un nivel abstracto.
i = 0
while i > 10:
    pass

```

```
# La siguiente función calcula la secuencia de Fibonacci
def fib(n):
    #     print(i, end=" ")
    pass

# En este punto del programa requiero el uso de la función fib(n):

fib(100000) #
```

10.7.1 Más ejemplos.

```
# Calcula números primos usando la
# criba de Eratóstenes
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'igual a ', x, '*', n//x)
            break
    else:
        print(n, 'es un número primo')
```

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

```
# Determina números pares e impares
for num in range(2, 10):
    if num % 2 == 0:
        print("Número par ", num)
        continue
    print("Número impar", num)
```

```
Número par  2
Número impar 3
Número par  4
Número impar 5
Número par  6
Número impar 7
Número par  8
Número impar 9
```

```
# Checa la clave de un usuario. Después de tres
# intentos fallidos termina. Si se da la clave
```

```
# correcta (despedida) se termina.  
suma = 0  
while suma < 3:  
    entrada = input("Clave:")  
    if entrada == "despedida":  
        break  
    suma = suma + 1  
    print("Intento %d. \n " % suma)  
print("Tuviste {} intentos fallidos.".format(suma))
```

Clave: despedida

Tuviste 0 intentos fallidos.

11 match (desde la versión 3.10)

Similar al switch de lenguajes como C, C++, Java.

```
def http_error(status):  
    match status:  
        case 400:  
            return "Bad request"  
        case 404:  
            return "Not found"  
        case 418:  
            return "I'm a teapot"  
        case _:  
            return "Something's wrong with the internet"
```

```
# Modifica el valor del argumento y observa lo que sucede  
http_error(500)
```

"Something's wrong with the internet"

```
# Modifica los valores de la siguiente tupla y observa el resultado  
point = (0,0)  
  
match point:  
    case (0, 0):  
        print("Origin")  
    case (0, y):  
        print(f"Y={y}")  
    case (x, 0):  
        print(f"X={x}")  
    case (x, y):  
        print(f"X={x}, Y={y}")
```

```
case _:  
    raise ValueError("Not a point")
```

Origin

Para más detalles véase [match Statements](#).

3 Tipos de datos básicos y operadores.

Objetivo. Explicar el concepto de variable, etiqueta, objetos y como se usan mediante algunos ejemplos.

Funciones de Python: - `print()`, `type()`, `id()`, `chr()`, `ord()`, `del()`

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under

[Attribution-ShareAlike 4.0 International](#)   

3.1 Tipos y operadores

En Python se tienen tres tipos de datos básicos principales:

Tipo	Ejemplo
Númerico	13, 3.1416, 1+5j
Cadena	"Frida", "Diego"
Lógico	True, False

3.2 Tipos numéricos

En Python se tienen tres tipos de datos numéricos: 1. Enteros 2. Reales 3. Complejos

A continuación se realiza una descripción de estos tipos numéricos. Más información se puede encontrar aquí: [Numeric types](#).

1. Enteros

Son aquellos que carecen de parte decimal. Para definir un entero hacemos lo siguiente:

```
entero = 13
```

Cuando se ejecuta la celda anterior, se crea el objeto `13` cuyo nombre es `entero`. Podemos imprimir el valor de `entero` y su tipo como sigue:

```
print(entero)
print(type(entero))
```

```
13
<class 'int'>
```

Es posible obtener más información del tipo `int` usando la biblioteca `sys`:

```
import sys
sys.int_info
```

```
sys.int_info(bits_per_digit=30, sizeof_digit=4, default_max_str_digits=4300,
str_digits_check_threshold=640)
```

2. Reales

Son aquellos que tienen una parte decimal. Para definir un número real (flotante) se hace como sigue:

```
pi = 3.141592
```

Cuando se ejecuta la celda anterior, se crea el objeto `3.141592` cuyo nombre es `pi`. Podemos imprimir el valor de `pi` y su tipo como sigue:

```
print(pi)
print(type(pi))
```

```
3.141592
<class 'float'>
```

```
# para obtener más información:
sys.float_info
```

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

3. Complejos

Son aquellos que tienen una parte real y una parte imaginaria, y ambas partes son números reales. Para definir un número complejo se hace como sigue:

```
complejo = 12 + 5j # La parte imaginaria lleva una j al final
```

Cuando se ejecuta la celda anterior, se crea el objeto `12 + 5j` cuyo nombre es `complejo`. En este caso, el contenido de `complejo` tiene dos partes: la real y la imaginaria. Podemos imprimir el valor de `complejo` y su tipo como sigue:

```
print(complejo)
print(type(complejo))
```

```
(12+5j)
<class 'complex'>
```

```
complejo.imag # accedemos a la parte imaginaria
```

5.0

```
complejo.real # accedemos a la parte real
```

12.0

```
complejo.conjugate() # calculamos el conjugado del número complejo.
```

(12-5j)

Nota: observa que hemos aplicado el método `conjugate()` al objeto `complejo`, esto es posible debido a que existe la clase `<class 'complex'>` en Python, y en ella se definen atributos y métodos para los objetos de esta clase. Más acerca de programación orientada a objetos la puedes ver en esta sección XXX.

3.2.1 Operadores aritméticos

Para los tipos numéricos descritos antes, existen operaciones aritméticas que se pueden aplicar sobre ellos. Veamos:

```
# Suma
1 + 2
```

```
# Resta
5 - 32
```

```
# Multiplicación
3 * 3
```

```
# División
3 / 2
```

```
# Potencia
81 ** (1/2)
```

3.2.2 Precedencia de operadores.

La aplicación de los operadores tiene cierta precedencia que está definida en cada implementación del lenguaje de programación. La siguiente tabla muestra el orden en que se aplicarán los operadores en una expresión.

Nivel	Categoría	Operadores
7	exponenciación	<code>**</code>
6	multiplicación	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>

Nivel	Categoría	Operadores
5	adición	<code>+</code> , <code>-</code>
4	relacional	<code>==</code> , <code>!=</code> , <code><=</code> , <code>>=</code> , <code>></code> , <code><</code>
3	logicos	<code>not</code>
2	logicos	<code>and</code>
1	logicos	<code>or</code>

Como se puede ver, siempre se aplican primero los operadores aritméticos (niveles 7,6, y 5), luego los relacionales (nivel 4) y finalmente los lógicos (3, 2 y 1).

Más acerca de este tema se puede ver aquí: [Operator precedence](#).

A continuación se muestran ejemplos de operaciones aritméticas en donde se resalta est precedencia:

```
# Precedencia de operaciones: primero se realiza la multiplicación
1 + 2 * 3 + 4
```

11

```
# Es posible usar paréntesis para modificar la precedencia: primero la suma
(1 + 2) * (3 + 4)
```

21

```
# ¿Puedes explicar el resultado de acuerdo con la precedencia
# descrita en la tabla anterior?
6/2*(2+1)
```

9.0

3.2.3 Operaciones entre tipos diferentes

Es posible combinar operaciones entre tipos de números diferentes. Lo que Python hará es promover cada número al tipo más sofisticado, siendo el orden de sofisticación, de menos a más, como sigue: `int`, `float`, `complex`.

```
a = 1 # un entero
b = 2 * 3j # un complejo
a + b # resultará en un complejo
```


3.2.4 Operadores de asignación

Existen varios operadores para realizar asignaciones: `=`, `+=`, `-=`, `*=`, `/=`, `**=`, `%=`. La forma de uso de estos operadores se muestra en los siguientes ejemplos:

```
etiqueta = 1.0
suma = 1.0
suma += etiqueta # Equivalente a : suma = suma + etiqueta
print(suma)
```

```
etiqueta = 4
resta = 16
resta -= etiqueta # Equivalente a : resta = resta - etiqueta
print(resta)
```

```
etiqueta = 2
mult = 12
mult *= etiqueta # Equivalente a : mult = mult * etiqueta
print(mult)
```

```
etiqueta = 5
div = 50
div /= etiqueta # Equivalente a : divide = divide / etiqueta
print(div)
```

```
etiqueta = 2
pot = 3
pot **= etiqueta # Equivalente a : pot = pot ** etiqueta
print(pot)
```

```
etiqueta = 5
modulo = 50
modulo %= etiqueta # Equivalente a : modulo = modulo % etiqueta
print(modulo)
```

3.3 Tipos lógicos

Es un tipo utilizado para realizar operaciones lógicas y puede tomar dos valores: `True` o `False`.

```
bandera = True
print(type(bandera))
```

3.3.1 Operadores relacionales

Cuando se aplica un operador relacional a dos expresiones, se realiza una comparación entre dichas expresiones y se obtiene como resultado un tipo lógico. **True** o **False**.

Los operadores relacionales que se pueden usar son: **==**, **!=**, **<=**, **>=**, **>**, **<**. A continuación se muestran algunos ejemplos:

```
35 > 562 # ¿Es 35 mayor que 562?
```

False

```
32 >= 21 # ¿Es 32 mayor o igual que 21?
```

True

```
12 < 34 # ¿Es 12 menor que 34?
```

True

```
12 <= 25 # ¿Es 12 menor o igual que 25?
```

True

```
5 == 5 # ¿Es 5 igual a 5?
```

True

```
23 != 23 # ¿Es 23 diferente de 23?
```

False

```
'aaa' == 'aaa' # Se pueden comparar otros tipos de datos
```

True

```
5 > len('5')
```

True

3.3.2 Operaciones lógicas.

Los operadores lógicos que se pueden usar son: **not** , **and** y **or** . Veamos algunos ejemplos

```
(5 < 32) and (63 > 32)
```

True

Debido a la precedencia de operadores, no son necesarios los paréntesis en la operaciones relacionales de la expresión anterior (véase tabla ...):

```
5 < 32 and 63 > 32
```

True

Aunque a veces el uso de paréntesis hace la lectura del código más clara:

```
(2.32 < 21) and (23 > 63)
```

False

```
(32 == 32) or (5 < 31)
```

True

```
(32 == 21) or (31 < 5)
```

False

```
not True
```

False

```
not (32 != 32)
```

True

3.3.2.1 Comparación entre números flotantes.

La comparación entre números de tipo flotante debe realizarse con cuidado, veamos el siguiente ejemplo:

```
(0.4 - 0.3) == 0.1
```

False

El cálculo a mano de $(0.4 - 0.3)$ da como resultado 0.1 ; pero en una computadora este cálculo es aproximado y depende de las características del hardware (exponente, mantisa, base, véase). En Python el resultado de la operación $(0.4 - 0.3)$ es diferente de 0.1 , veamos:

```
print(0.4 - 0.3)
```

```
0.10000000000000003
```

Python ofrece herramientas que permiten realizar una mejor comparación entre números de tipo flotante. Por ejemplo la biblioteca `math` contiene la función `isclose(a, b)` en donde se puede definir una tolerancia mínima para que las dos expresiones, `a` y `b` se consideren iguales (*cercanas*), por ejemplo:

```
import math
math.isclose((0.4 - 0.3), 0.1)
```

```
True
```

Se recomienda revisar el manual de [math.isclose\(\)](#) y el de [numpy.isclose\(\)](#) para comparación de arreglos con elementos de tipo flotante.

3.4 Fuertemente Tipado.

Python es fuertemente tipado, lo que significa que el tipo de un objeto no puede cambiar repentinamente; se debe realizar una conversión explícita si se desea cambiar el tipo de un objeto.

Esta característica también impide que se realicen operaciones entre tipos no compatibles.

Veamos unos ejemplos:

```
lógico = True
real    = 220.0
entero  = 284
complejo = 1+1j
cadena  = 'numeros hermanos'
```

```
lógico + real # Los tipos son compatibles
```

```
221.0
```

```
lógico + complejo # Los tipos son compatibles
```

```
(2+1j)
```

```
cadena + real # Los tipos no son compatibles
```

TypeError: can only concatenate str (not "float") to str

3.5 Conversión entre tipos (*casting*)

Es posible transformar un tipo en otro tipo compatible; a esta operación se lo conoce como *casting*.

3.5.1 Función `int()`

Transforma objetos en enteros, siempre y cuando haya compatibilidad.

```
cadena = '1000'  
print(type(cadena))  
entero = int(cadena)  
print(type(entero))  
print(entero)
```

```
<class 'str'>  
<class 'int'>  
1000
```

```
flotante = 3.141592  
entero = int(flotante) # Trunca la parte decimal  
print(entero)
```

3

```
complejo = 4-4j  
entero = int(complejo) # Tipos NO COMPATIBLES
```

TypeError: int() argument must be a string, a bytes-like object or a real number, not 'complex'

```
entero = int(True)  
print(entero)
```

1

```
print(1 == True)
```

True

Función `str()`

Transforma objetos en cadenas, siempre y cuando haya compatibilidad.

```
entero = 1000
print(type(entero))
cadena = str(entero)
print(type(cadena))
print(cadena)
```

```
<class 'int'>
<class 'str'>
1000
```

```
complejo = 5+1j
print(complejo)
print(type(complejo))
cadena = str(complejo)
print(cadena)
print(type(cadena))
```

```
(5+1j)
<class 'complex'>
(5+1j)
<class 'str'>
```

Función `float()`

Transforma objetos en flotantes, siempre y cuando haya compatibilidad.

```
cadena = '3.141592'
print(cadena)
print(type(cadena))
real = float(cadena)
print(real)
print(type(real))
```

```
3.141592
<class 'str'>
3.141592
<class 'float'>
```

```
float(33)
```

```
33.0
```

```
float(False)
```

```
0.0
```

```
float(3+3j) # NO hay compatibilidad
```

`TypeError: float() argument must be a string or a real number, not 'complex'`

En general, si existe el tipo `<class 'MiClase'>`, donde `MiClase` puede ser un tipo de dato definido dentro de Python, alguna biblioteca o creada por el usuario, es posible realizar el *casting* del objeto `a` al tipo `<class 'MiClase'>` haciendo: `MiClase(a)` siempre y cuando haya compatibilidad.

3.6 Constantes

Python contiene una serie de constantes integradas a las que no se les puede cambiar su valor.

Más detalles se pueden encontrar en: [Built-in Constants](#)

Las principales constantes son las siguientes:

- `False`: de tipo Booleano.
- `True`: de tipo Booleano.
- `None`: El único valor para el tipo `NoneType`. Es usado frecuentemente para representar la ausencia de un valor, por ejemplo cuando no se pasa un argumento a una función.
- `NotImplemented`: es un valor especial que es regresado por métodos binarios especiales (por ejemplo `__eq__()`, `__lt__()`, `__add__()`, `__rsub__()`, etc.) para indicar que la operación no está implementada con respecto a otro tipo.
- `Ellipsis`: equivalente a `...`, es un valor especial usado mayormente en conjunción con la sintaxis de *slicing* de arreglos.
- `__debug__`: Esta constante es verdadera si Python no se inició con la opción `-O`.

Las siguiente constantes son usadas dentro del intérprete interactivo (no se pueden usar dentro de programas ejecutados fuera del intérprete).

- `quit` (code=None)
- `exit` (code=None)
- `copyright`
- `credits`
- `license`

```
copyright()
```

Copyright (c) 2001–2023 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995–2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991–1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.

```
license()
```

A. HISTORY OF THE SOFTWARE

=====

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <https://www.cwi.nl>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <https://www.cnri.reston.va.us>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.


All Python releases are Open Source (see <https://opensource.org> for the Open Source Definition). Historically, most, but not all, Python

Hit Return for more, or q (and Return) to quit: q

14 Iteradores y Generadores.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

15 Iteradores

- Como vimos en la sección XXXX, en Python existen objetos que contienen secuencias de otros objetos (listas, tuplas, diccionarios, etc).
- La mayoría de los objetos contenedores se pueden recorrer usando un ciclo **for ... in ...**. Este es un estilo claro y conveniente que impregna el universo de Python.

Por ejemplo:

```
mi_cadena = "abcd"

print("\nIteración sobre una cadena: ", end='')
for char in mi_cadena:
    print(char, end=' ')
```

Iteración sobre una cadena: a b c d

Notas importantes: - La instrucción **for** llama a la función **iter()** que está definida dentro del objeto **contenedor**. - La función **iter()** regresa como resultado un objeto **iterador** que define el método **__next__()**, con el que se puede acceder a los elementos del objeto contenedor, uno a la vez. - Cuando no hay más elementos, **__next__()** lanza una excepción de tipo **StopIteration** que le dice al ciclo **for** que debe terminar. - Se puede ejecutar al método **__next__()**, al iterador, usando la función de biblioteca **next()**.

Por ejemplo:

```
iterador = iter(mi_cadena) # Obtenemos un iterador para la cadena
print(type(iterador)) # Obtenemos el tipo del iterador
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: a
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: b
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: c
print(next(iterador)) # Aplicamos __next__() al iterador para obtener: d
```

<class 'str_ascii_iterator'>

a
b

c
d

Cuando ya llegamos al final de la secuencia e intentamos aplicar `__next__()` obtenemos una excepción:

```
next(iterador) # Sobrepasó los elementos, se obtiene la excepción StopIteration
```

`StopIteration`:

Observa que cuando se hace el recorrido de la cadena usando el ciclo `for` no se produce ninguna excepción debido a que maneja la excepción para terminar el proceso adecuadamente.

Se puede crear un iterador y aplicarle la función `next()` a cualquier secuencia, por ejemplo a una lista

```
# Creación de una lista
cuadradosI = [x*x for x in range(10)]
print(cuadradosI)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
# Recorriendo la lista usando un iterador en una lista concisa:
iterador = iter(cuadradosI)
[next(iterador) for x in range(10)]
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Estos objetos iterables son manejables y prácticos debido a que se pueden usar tantas veces como se desee, pero se almacenan todos los valores en memoria y esto no siempre es conveniente, sobre todo cuando se tienen muchos valores.

16 Generadores

- Los objetos **generadores** son iteradores.
- Pero solo se puede iterar sobre ellos una sola vez. Esto es porque los generadores no almacenan todos los valores en memoria, ellos generan los valores al vuelo.
- Un generador se crea como sigue:

```
(expresion for x in secuencia)
```

donde `expresion` es una expresión válida de Python que genera los elementos del generador; `x` es un elemento al que se le aplica la `expresion` y `secuencia` es cualquier secuencia válida en Python.

Por ejemplo:

```
# Un generador simple
gen = (x for x in range(3))
```

```
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen)) # Produce una excepción de tipo StopIteration
```

```
0
1
2
```

StopIteration:

```
# Creamos el generador
cuadradosG = (x*x for x in range(10))
print(type(cuadradosG))

# Recorremos el generador en un ciclo for
for i in cuadradosG:
    print(i, end=' ')
```

```
<class 'generator'>
0 1 4 9 16 25 36 49 64 81
```

En el ejemplo anterior tenemos: - genera el 0, es usado y lo olvida - genera el 1, es usado y lo olvida - genera el 4, es usado y lo olvida - etcétera.

Un generador solo se puede usar una vez, pues va calculando sus valores uno por uno e inmediatamente los va olvidando. Si intentamos utilizar una vez más el generador, ya no obtendremos nada:

```
for i in cuadradosG:    # Este ciclo no imprimirá nada por que
    print(i, end=' ')   # el generador ya se usó antes
```

Observa que no se produce un error porque estamos usando el generador, que ya ha sido usado con anterioridad, dentro del ciclo `for`.

17 Yield

- Es una palabra clave que suspende la ejecución de una función y envía un valor de regreso a quien la ejecuta, pero retiene la información suficiente para reactivar la ejecución de la función donde se quedó. Si la función se vuelve a ejecutar, se reanuda desde donde se detuvo la última vez.
- Esto permite al código producir una serie de valores uno por uno, en vez de calcularlos y regresarlos todos.
- Una función que contiene la declaración `yield` se le conoce como función generadora.

Por ejemplo:

```
# Función generadora
def generadorSimple():
    print('yield 1 : ', end=' ')
    yield 1
    print('yield 2 : ', end=' ')
    yield 2
    print('yield 3 : ', end=' ')
    yield 3

# Se construye un generador
gen = generadorSimple()

# Se usa el generador
print('Primera ejecución de la función generadora: {}'.format(next(gen)))
print('Segunda ejecución de la función generadora: {}'.format(next(gen)))
print('Tercera ejecución de la función generadora: {}'.format(next(gen)))
```

```
yield 1 : Primera ejecución de la función generadora: 1
yield 2 : Segunda ejecución de la función generadora: 2
yield 3 : Tercera ejecución de la función generadora: 3
```

Si se intenta usar una vez más el generador obtendremos una excepción de tipo **StopIteration**:

```
print('Cuarta ejecución de la función generadora: {}'.format(next(gen)))
```

StopIteration:

Notas importantes. - **yield** es usada como un **return**, excepto que la función regresa un objeto **generador**. - Las funciones generadoras regresan un objeto generator. - Los objetos generadores pueden ser usados en ciclos **for ... in ...** o **while**.

Entonces, una función generadora regresa un objeto **generador** que es iterable, es decir, se puede usar como un **iterador**.

```
def construyeUnGenerador(v):
    for i in range(v):
        yield i*i

# Se construye una función generadora
cuadradosY = construyeUnGenerador(10)
print(type(cuadradosY))

for i in cuadradosY:
    print(i)
```

```
<class 'generator'>
```

```
0
1
4
```

9
16
25
36
49
64
81

Se recomienda usar **yield** cuando se desea iterar sobre una secuencia, pero no se quiere almacenar toda la secuencia en memoria.

17.1 Ejemplo 1.

Crear una función generadora que genere los cuadrados del 1 al ∞ .

```
# Función generadora que genera el cuadrado de un número
def cuadradoSiguiente():
    i = 1;
    while True:
        yield i*i
        i += 1 # La siguiente ejecución se
              # reactiva en este punto

for numero in cuadradoSiguiente():
    if numero > 100:
        break
    print(numero)
```

1
4
9
16
25
36
49
64
81
100

17.2 Ejemplo 1.

Crear un generador de los números de Fibonacci.

```
# Función generadora
def fib(limite):
    a, b = 0, 1

    while a < limite:
        yield a
        a, b = b, a + b # La siguiente iteración se reactiva en este punto
```

```
N = 100

# Generador
x = fib(N)

while True:
    try:
        print(next(x), end=' ') # Usamos la función next() para iterar
    except StopIteration:      # Manejamos la excepción
        break
```

0 1 1 2 3 5 8 13 21 34 55 89


```
# Usando la función generadora directamente en un ciclo for
for i in fib(N):
    print(i, end=' ')
```

0 1 1 2 3 5 8 13 21 34 55 89

9 Funciones y docstring.

Objetivo. ...

Funciones de Python: ...

[MACTI-Algebra_Lineal_01](#) by [Luis M. de la Cruz](#) is licensed under [Attribution-ShareAlike 4.0 International](#) 

10 Definición de funciones

Las funciones son la primera forma de estructurar un programa. Esto nos lleva al paradigma de programación estructurada, junto con las construcciones de control de flujo. Las funciones nos permiten agrupar y reutilizar líneas de código.

La sintaxis es:

```
def nombre_de_la_función(parm1, parm2, ...):  
    bloque_de_código  
    return resultado
```

Una vez definida la función, es posible ejecutarla (hacer una llamada a la función) como sigue:

```
nombre_de_la_función(arg1, arg2, ...)
```

También es posible hacer lo siguiente:

En ambos casos, la función regresa un resultado debido a que existe la declaración **return** dentro de la función. Este resultado puede ser referenciado por una variable haciendo lo siguiente:

```
variable = nombre_de_la_función(arg1, arg2, ...)
```

La **variable** puede ser utilizada posteriormente para otros cálculos.

Observa que: * Cuando se define la función, se definen los **parámetros** que recibirá, en este caso **param1**, **param2**, ... * Cuando se ejecuta la función, se pasan los valores los **arg1**, **arg2**, ... , los cuales son los **argumentos** de la ejecución y serán sustituidos en los parámetros de la función.

Veamos un ejemplo simple:

```
# Función que calcula el cuadrado de su argumento.  
def squared(f):  
    return f ** 2
```

```
# Se ejecuta la función con el argumento 2
squared(2)
```

4

```
# Se ejecuta la función con el argumento 3
# el resultado se almacena en f2
f2 = squared(3)
print(f2)
```

9

Veamos ahora un ejemplo más interesante

```
# La siguiente función calcula la secuencia de Fibonacci
def fib(n): # La función se llama fib y tiene el parámetro n
    a, b = 0, 1
    while a < n:
        print(a, end=',')
        a, b = b, a+b
```

Observa que esta función no regresa ningún valor, solo imprime en pantalla un valor conforme lo calcula.

```
fib(50) # ejecutamos la función fib con el argumento 10
```

0,1,1,2,3,5,8,13,21,34,

Le podemos poner otro nombre a la función

```
Fibonacci = fib
```

```
Fibonacci(200)
```

0,1,1,2,3,5,8,13,21,34,55,89,144,

```
print(type(fib))
print(type(Fibonacci))
```

```
<class 'function'>
<class 'function'>
```

```
print(id(fib))
print(id(Fibonacci))
```

140670158000416

140670158000416

Observamos que se puede ejecutar la función `fib()` a través de `Fibonacci()` y que ambos nombres hacen referencia a la misma función.

11 Ámbitos

Las funciones (y otros operadores también), crean su propio ámbito, de tal manera que las etiquetas declaradas dentro de funciones son locales.

```
a = 20 # Objeto global etiquetado con a

def f():
    a = 21 # Objeto local etiquetado con a
    return a
```

```
# ¿Que valor tiene 'a' fuera de la función?
print(a)
```

20

```
# ¿Qué valor tiene la 'a' dentro de la función?
print(f())
```

20

Para usar el objeto global dentro de la función debemos usar `global`

```
a = 20

def f():
    global a
    return a
```

```
# ¿Que valor tiene 'a' fuera de la función?
print(a)
```

20

```
# La 'a' dentro de la función hace referencia a la 'a' global
print(f())
```

20

12 Retorno de una función

Como se mencionó antes, la declaración `return`, dentro de una función, regresa un objeto que en principio contiene el resultado de las operaciones realizadas por la función.

Veamos un ejemplo.

```
g = 9.81
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t, v0):
    g = 3.1416 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return (y, v) # regresa la posición [m] y la velocidad [m/s] en un objeto de
```

```
t = 2.0 # [s]
v0 = 20 # [m/s]
verticalThrow(t, v0)
```

(33.7168, 13.7168)

```
resultado = verticalThrow(t, v0)
```

```
print(resultado)
```

(33.7168, 13.7168)

13 Argumentos por omisión

Los parámetros de una función pueden tener valores (argumentos) por omisión, es decir, si no se da un valor para uno de los parámetros, entonces se toma el valor definido por omisión. Esto crea una función que se puede llamar con menos argumentos de los que está definida inicialmente.

Por ejemplo:

```
# Función que calcula la posición y velocidad en el tiro vertical de un objeto.
def verticalThrow(t, v0 = 20): # El valor 20 es un argumento por omisión
    g = 9.81 # [m / s**2]
    y = v0 * t - 0.5 * g * t**2
    v = v0 - g * t
    return (y, v)
```

```
pos, vel = verticalThrow(2.0) # El valor 2.0 corresponde al primer parámetro de 1
                             # En este caso v0 será igual a 20.
print(pos, vel)
```

20.38 0.3799999999999999

```
pos, vel = verticalThrow(2.0, 30) # En este caso v0 = 30
print(pos, vel)
```

40.379999999999995 10.379999999999999

Una función puede tener más de un argumento por omisión. Todos los parámetros que tienen argumentos por omisión deben estar al final de la lista en la declaración de la función.

Por ejemplo:

```
def f(a,b,c,d=10,e=20):
    return a+b+c+d+e
```

```
print(f(1,2,3)) # Se los dos argumentos por omisión 10 y 20
```

36

```
print(f(1,2,3,4)) # Se usa el último argumento por omisión 20
```

30

```
print(f(1,2,3,4,5)) # Se dan todos los argumentos.
```

15

14 Argumentos posicionales y keyword

Un **argumento** es el valor que se le pasa a una función cuando se llama. Hay dos tipos de argumentos:

Positional argument:

1. Un argumento que no es precedido por un identificador: `verticalThrow(3, 50)`
2. Un argumento que es pasado en una tupla precedido por `*`: `verticalThrow(*(3, 50))`.

En este caso, el `*` indica a Python que la tupla `(3, 50)` debe desempacarse cuando se reciba en la función, de tal manera que `3` será el primer argumento y `5` el segundo.

La llamada a la función del punto 2 es equivalente a la del punto 1.

Keyword argument:

3. Un argumento precedido por un identificador. `verticalThrow(t=3, v0=50)`
4. Un argumento que se pasa en un diccionario precedido por `**`: `verticalThrow(**{'t': 3, 'v0': 50})`.

En este caso, el `**` indica a Python que el diccionario `{'t': 3, 'v0': 50}` debe desempacarse cuando se reciba en la función. Observa que el diccionario contiene dos pares clave-valor: `'t': 3` y `'v0': 50`.

La llamada a la función del punto 4 es equivalente a la del punto 3.

Veamos los ejemplos en código:

Primer recordemos que la firma de la función es `def verticalThrow(t, v0 = 20):` es decir, el primer parámetro es `t` y el segundo `v0`.

```
# Los argumentos se sustituyen en los parámetros en el orden de acuerdo a su posición
verticalThrow(3,50)
```

```
(105.85499999999999, 20.57)
```

```
# Lo anterior NO es equivalente a:
verticalThrow(50,3)
```

```
(-12112.5, -487.5)
```

```
# Se pueden pasar los argumentos empacados en una tupla
verticalThrow(*(3,50))
```

```
(105.85499999999999, 20.57)
```

```
# Se puede usar el nombre del parámetro para determinar
# como se sustituyen los argumentos:
verticalThrow(t=3,v0=50)
```

```
(105.85499999999999, 20.57)
```

```
# Lo anterior SI es equivalente a:
verticalThrow(v0=50, t=3)
```

```
(105.85499999999999, 20.57)
```

```
# Se pueden pasar los argumentos empacados en un diccionario
verticalThrow(**{'t':3,'v0':50})
```

```
(105.85499999999999, 20.57)
```

```
# También se acepta esta forma:
verticalThrow(**dict(t = 3, v0 = 50))
```

```
(105.85499999999999, 20.57)
```

15 Número variable de parámetros

Dada la funcionalidad descrita en la sección anterior, es posible que una función reciba un número variable de argumentos.

```
# *args: número variable de Positional arguments empacados en una tupla
# *kwargs: número variable de Keyword arguments empacados en un diccionario
def parametrosVariables(*args, **kwargs):
    print('args es una tupla : ', args)
    print('kwargs es un diccionario: ', kwargs)
    print(set(kwargs))
```

```
parametrosVariables('one', 'two', 'three', 'four', a = 4, x=1, y=2, z=3, w=[1,2,2])
```

```
args es una tupla : ('one', 'two', 'three', 'four')
kwargs es un diccionario: {'a': 4, 'x': 1, 'y': 2, 'z': 3, 'w': [1, 2, 2]}
{'a', 'y', 'w', 'z', 'x'}
```

```
parametrosVariables(1,2,3, w=8, y='cadena')
```

```
args es una tupla : (1, 2, 3)
kwargs es un diccionario: {'w': 8, 'y': 'cadena'}
{'w', 'y'}
```

```
# Los argumentos que vienen en un diccionario se desempacan
# y se pueden usar dentro de la función:
def funcion_kargs(**argumentos):
    for key, val in argumentos.items():
        print(f" {key} : {val}")
```

```
funcion_kargs(nombre = 'Luis', apellido='de la Cruz', edad=15, peso=80.5 )
```

```
nombre : Luis
apellido : de la Cruz
edad : 15
peso : 80.5
```

```
funcion_kargs(nombre = 'Luis', apellido='de la Cruz', estudios='primaria', edad=1
```

```
nombre : Luis  
apellido : de la Cruz  
estudios : primaria  
edad : 15  
peso : 80.5  
num_cuenta : 12334457
```

```
# Se puede definir un diccionario  
mi_dicc = {'nombre':'Luis', 'apellido':'de la Cruz', 'edad':15, 'peso':80.5}
```

```
# Se usa el diccionario para llamar a la función  
funcion_kargs(**mi_dicc)
```

```
nombre : Luis  
apellido : de la Cruz  
edad : 15  
peso : 80.5
```

16 Funciones como parámetros de otras funciones.

Las funciones pueden recibir como argumentos objetos muy complejos, incluso otras funciones. Veamos un ejemplo simple:

```
# Un función simple  
def g():  
    print("Iniciando la función 'g()'")  
  
# Una función que reibirá otra función:  
def func(f):  
    print("Iniciando la función 'func()'")  
    print("Ejecución de la función 'f()', nombre real '" + f.__name__ + "()'")  
    f() # Se ejecuta la función que se recibió en el parámetro f
```

```
func(g)
```

```
Iniciando la función 'func()'  
Ejecución de la función 'f()', nombre real 'g()'  
Iniciando la función 'g()'
```

16.0.1 Ejemplo 1. Integración numérica.

En este ejemplo el objetivo es crear un función que recibirá como argumentos la función matemática a integrar, los límites de integración y el número de puntos para realizar la integración. Regresará como resultado un número que es la aproximación de la integral.

```
import math

def integra(func,a,b,N):
    # Se utiliza el método de Simpson para la integración.
    # El parámetro 'func' es la función a integrar
    print(f"Integral de la función {func.__name__}() en el intervalo ({a},{b}) us
    h = (b - a) / N
    resultado = 0
    x = [a + h*i for i in range(N+1)]
    for xi in x:
        resultado += func(xi) * h
    return resultado
```

```
# Integral de la función sin() de la biblioteca math.
print(integra(math.sin, 0, math.pi, 100))

# Integral de la función cos() de la biblioteca math.
print(integra(math.cos, -0.5 * math.pi, 0.5 * math.pi, 50))
```

Integral de la función sin() en el intervalo (0,3.141592653589793) usando 100 puntos
1.9998355038874436

Integral de la función cos() en el intervalo (-1.5707963267948966,1.5707963267948966)
usando 50 puntos
1.9993419830762613

17 Funciones que regresan otra función.

Como vimos antes, una función puede regresar un objeto de cualquier tipo, incluyendo una función. Veamos un ejemplo:

```
# La funcionPadre() regresará como resultado una de dos funciones
# definidas dentro de ella.
def funcionPadre(n):

    # Se define la función 1
    def funcionHijo1():
        return "funcionHijo1(): n = {}".format(n)

    # Se define la función 2
    def funcionHijo2():
        return "funcionHijo2(): n = {}".format(n)
```

```
# Se determina la función que se va a regresar
if n == 10:
    return funcionHijo1
else:
    return funcionHijo2
```

```
# La funcionPadre() regresa una función
funcionPadre(36)
```

```
<function __main__.funcionPadre.<locals>.funcionHijo2(>
```

```
# Asignamos el resultado de la funcionPadre() a un nombre
f1 = funcionPadre(10)
f2 = funcionPadre(20)
```

```
print(f1()) # Resultado de la funcionf1(), generada con la funcionPadre()
print(f2()) # Resultado de la funcionf2(), generada con la funcionPadre()
```

```
funcionHijo2(): n = 20
funcionHijo1(): n = 10
```

17.0.1 Ejemplo 2. Polinomios de segundo grado.

Implementar una fábrica de polinomios de segundo grado:

$$p(x) = ax^2 + bx + c$$

```
# Esta función recibe los coeficientes del polinomio
# y regresa una función que calcula el polinomio de
# segundo grado.
def polinomio(a, b, c):

    def polSegundoGrado(x):
        return a * x**2 + b * x + c

    return polSegundoGrado
```

```
# Dos polinomios de segundo grado
p1 = polinomio(2, 3, -1) # 2x^2 + 3x - 1
p2 = polinomio(-1, 2, 1) # -x^2 + 2x + 1

# Evaluación de los polinomios en el intervalo
# (-2,2) con pasos de 1
```



```
for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d}')
```

x = -2	p1(x) = 1	p2(x) = -7
x = -1	p1(x) = -2	p2(x) = -2
x = 0	p1(x) = -1	p2(x) = 1
x = 1	p1(x) = 4	p2(x) = 2

17.0.2 Ejemplo 2. Polinomios de cualquier grado.

Implementar una fábrica de polinomios de cualquier grado:

$$\sum_{k=0}^n a_k x^k = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

```
# Esta función recibe un conjunto de argumentos variable
# para construir un polinomio de cualquier grado.
# Regresa la función que implementa el polinomio.
def polinomioFactory(*coeficientes):
```

```
    def polinomio(x):
        res = 0
        for i, coef in enumerate(coeficientes):
            res += coef * x ** i
        return res
```

```
    return polinomio
```

```
# Se generan 4 polinomios de diferente grado
p1 = polinomioFactory(5)           # a_0 = 5
p2 = polinomioFactory(2, 4)        # 4 x + 2
p3 = polinomioFactory(-1, 2, 1)    # x^2 + 2x - 1
p4 = polinomioFactory(0, 3, -1, 1) # x^3 - x^2 + 3x + 0
```

```
# Evaluación de los polinomios en el intervalo
# (-2,2) con pasos de 1
for x in range(-2, 2, 1):
    print(f'x = {x:3d} \t p1(x) = {p1(x):3d} \t p2(x) = {p2(x):3d} \t p3(x) = {p3(x):3d} \t p4(x) = {p4(x):3d}')
```

x = -2	p1(x) = 5	p2(x) = -6	p3(x) = -1	p4(x) = -18
x = -1	p1(x) = 5	p2(x) = -2	p3(x) = -2	p4(x) = -5
x = 0	p1(x) = 5	p2(x) = 2	p3(x) = -1	p4(x) = 0
x = 1	p1(x) = 5	p2(x) = 6	p3(x) = 2	p4(x) = 3

18 Documentación con *docstring*

Python ofrece dos tipos básicos de comentarios para documentar el código:

1. Lineal.

Este tipo de comentarios se llevan a cabo utilizando el símbolo especial `#`. El intérprete de Python sabrá que todo lo que sigue delante de este símbolo es un comentario y por lo tanto no se toma en cuenta en la ejecución:

```
a = 10 # Este es un comentario
```

2. Docstrings

En programación, un *docstring* es una cadena de caracteres embebidas en el código fuente, similares a un comentario, para documentar un segmento de código específico. A diferencia de los comentarios tradicionales, las docstrings no se quitan del código cuando es analizado, sino que son retenidas a través de la ejecución del programa. Esto permite al programador inspeccionar esos comentarios en tiempo de ejecución, por ejemplo como un sistema de ayuda interactivo o como metadatos. En Python se utilizan las triples comillas para definir un *docstring*.

```
def funcion(x):  
    '''  
    Esta es una descripción de la función ...  
    '''  
  
def foo(y):  
    '''  
    También de esta manera se puede definir una docstring  
    '''
```

```
def suma(a,b):  
    '''  
    Esta función calcula la suma de los parámetros a y b.  
    Regresa el resultado de la suma  
    '''  
    return a + b
```

```
suma
```

```
<function __main__.suma(a, b)>
```

```
# En numpy se usa la siguiente definición de docstrings  
def suma(a,b):  
    '''  
    Calcula la suma de los dos parámetros a y b.  
  
    Args:  
        a: int Numero a sumar
```

```
        b: int Numero a sumar
Return:
        c: int Suma del numero a y b
...
c = a + b
return c
```

suma

<function __main__.suma(a, b)>

Existen diferentes estilos de documentación tipo *docstring* vease por ejemplo: [Numpy](#), [Matplotlib](#).

Para más información véase [PEP 257 – Docstring Conventions](#) y [PEP 8 – Style Guide for Python Code](#).