```python
study = UC(name = "PDA", studyHs = 8)
mate = Mate() #nuevo mate

while study.notDone():
    study.meterle()
    mate.cebar()
    if mate.termoEmpyt():
        mate.refill()
```

UTEC
Universidad Tecnológica

Ingeniería Biomédica

# Programación Digital Avanzada

## Módulo 2 – 1ra parte

Mag. Bioing. Baldezzari Lucas

Ingeniería Biomédica
7mo Semestre

2022

V1.0

# Objetivos del Módulo 2

Revisión y repaso de,

- Variables y asignaciones.
- Estructuras de control.
- Tipos de datos.


# Tipos de datos, variables, asignaciones

**¿Qué es un Tipo de Dato?**

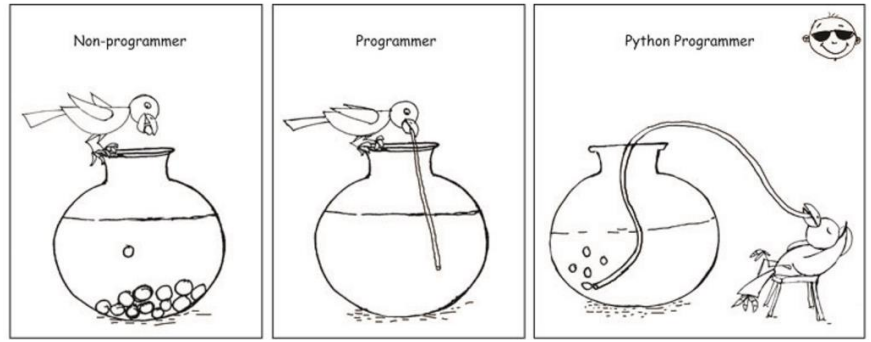
En Python significa la pertenencia a una categoría particular de dato. Cada tipo de dato tiene un nombre, por ejemplo *float* y al mismo tiempo posee métodos para operarlos.

Python incluye varios tipos de datos *built-in, como ser,* números enteros, reales y complejos. Diccionarios. Datos secuenciales como los strings, tuplas, listas y sets. Booleanos.

**¿Qué es una expresión?**

Una expresión es una combinación de valores, variables y operadores.

Un valor en si mismo es considerado una expresión.



# Tipos de datos, variables, asignaciones

**¿Qué es una variable?**

Es una posición de memoria que tiene un nombre y almacena un valor.

**Consideraciones**

No se tipifican

Los nombres de variables no pueden empezar con números u operadores (-, +, *, “”,entre otros)

No puden usarse palabras reservadas como nombres de variables.

Es una buena práctica declarar las variables con el método *camelCase* o con _.

Declarando un tipo de variable en Python

# Tipos de datos, variables, asignaciones

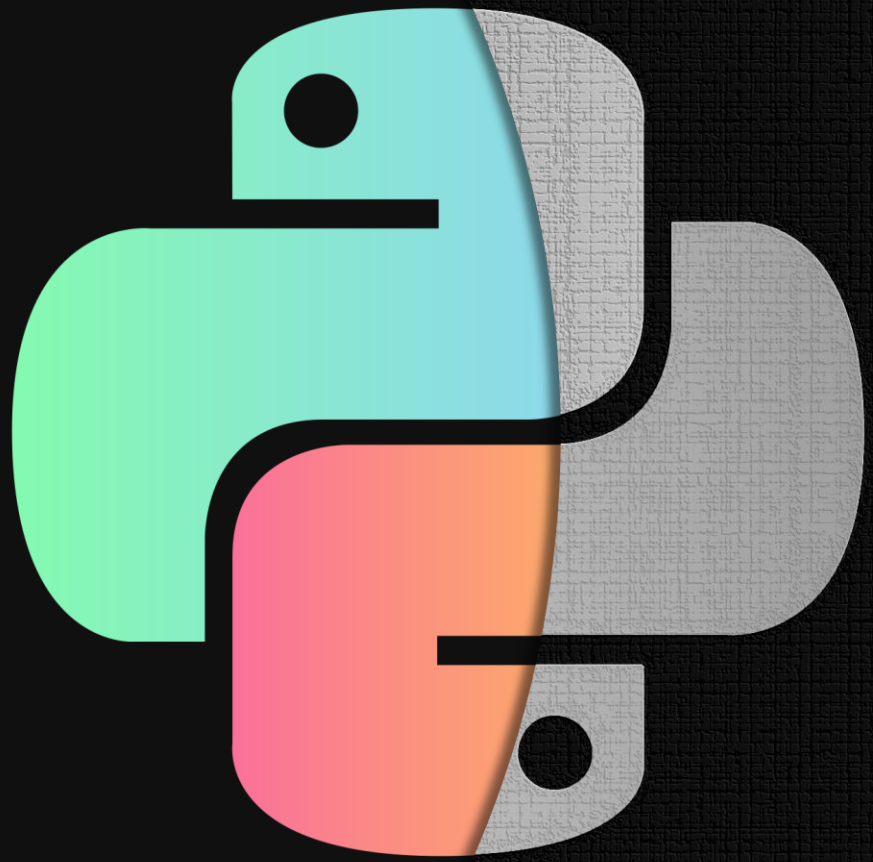
¿Cuál es el orden de operaciones en una asignación?

Python posee un orden de operación antes de realizar una asignación. En el caso de operaciones matemáticas, Python usa la convención matemática. El acrónimo PEMDAS es útil para recordar cómo opera Python.

1. Paréntesis
2. Exponenciales
3. Multiplicación y División
4. Adición y Substracción
5. Operadores con el mismo nivel de jerarquía se avalúan de izquierda a derecha (excepto las exponenciaciones)

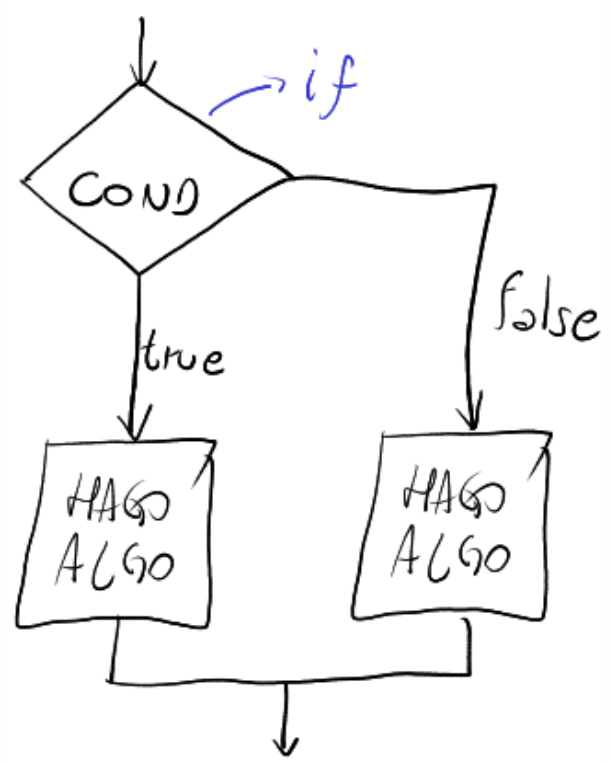
```
cuenta = (1+3)**(5-3) + 2*3**2 - (2*3-1) + (6+4/2)/8*2
```

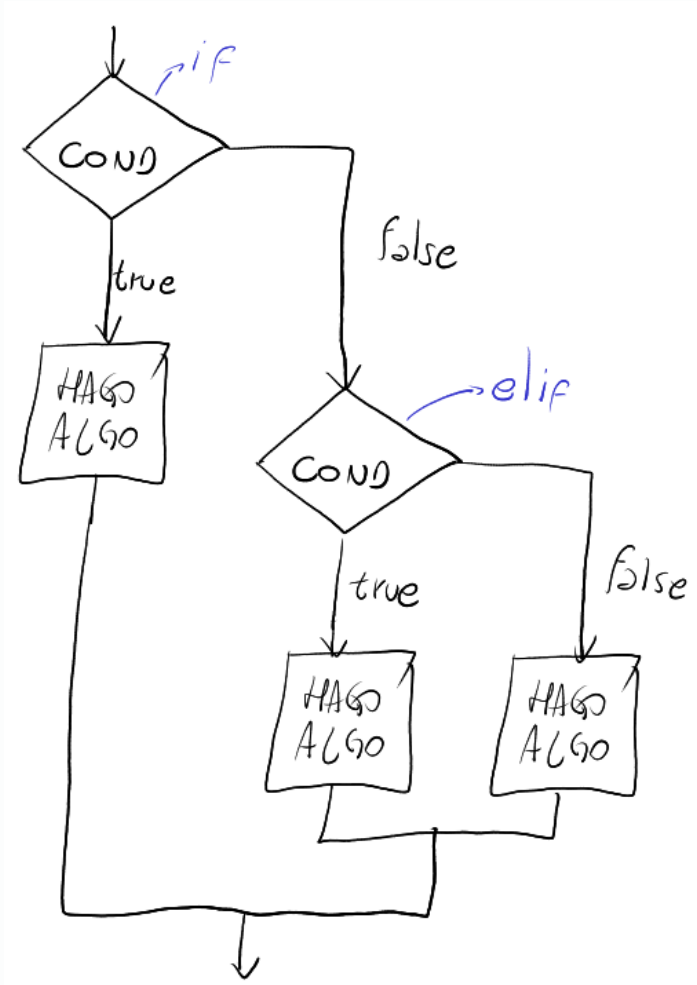
# Estructuras de control



# Sentencias *if, elif, else*

Estructura de control…

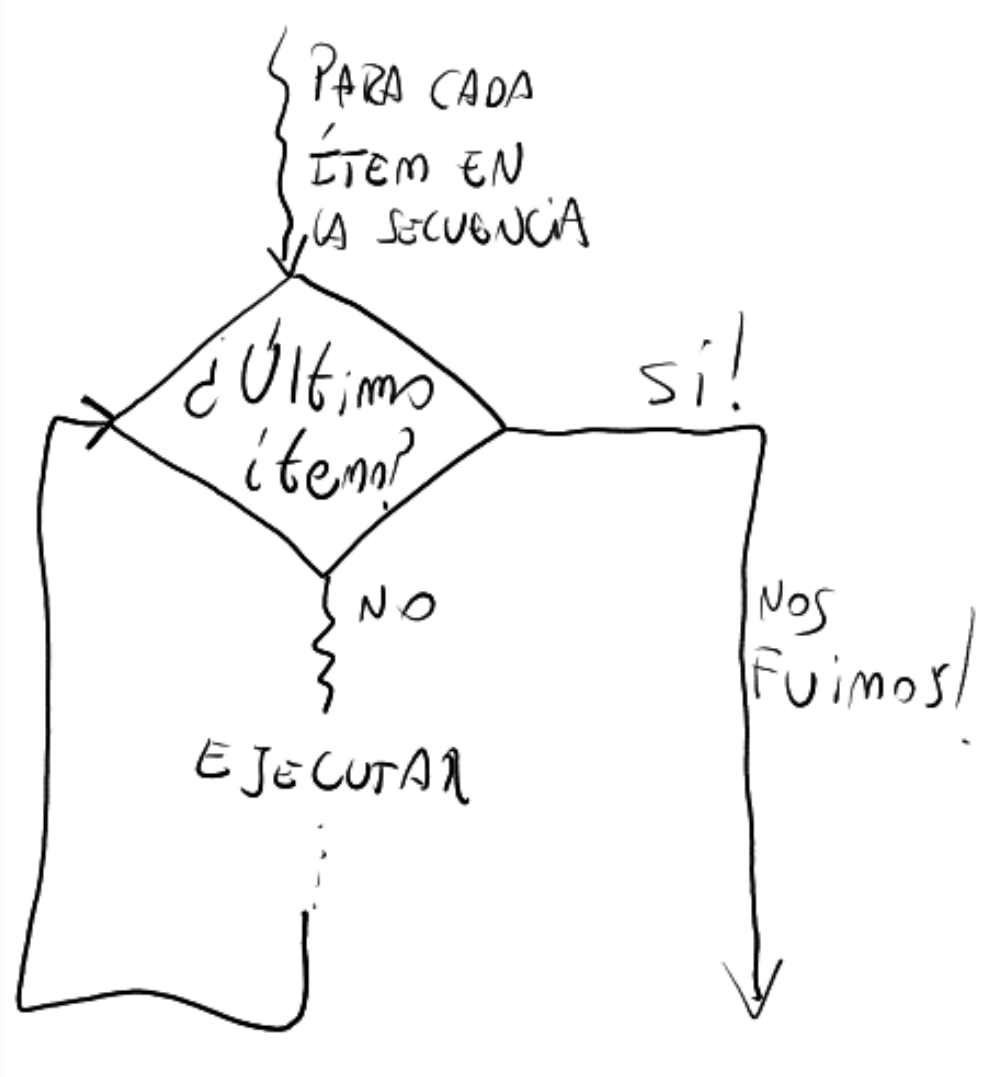



```
1 += 1 #incrementamos un trial
unter = 0 #reiniciamos timer

elf.trial

alControl(self):
Función para llevar a cabo un control gener

if self.systemControl[0] == b"1" and not self

    if not self.useExternalTimer:
        self.timer()

    if self.timerInteFlag: #timerInteFlag se
        self.trialControl()
        self.timerInteFlag = 0 #reiniciamos f

elif self.systemControl[0] == b"1" and self.tr

    if not self.useExternalTimer:
        self.timer()

    if self.timerInteFlag: #timerInteFlag se
        self.trialControl()
        self.timerInteFlag = 0 #reiniciamos f

e:
self.endSesion()

    self.sessionStatus


ime.time()#/1000
```

# Sentencia *for*

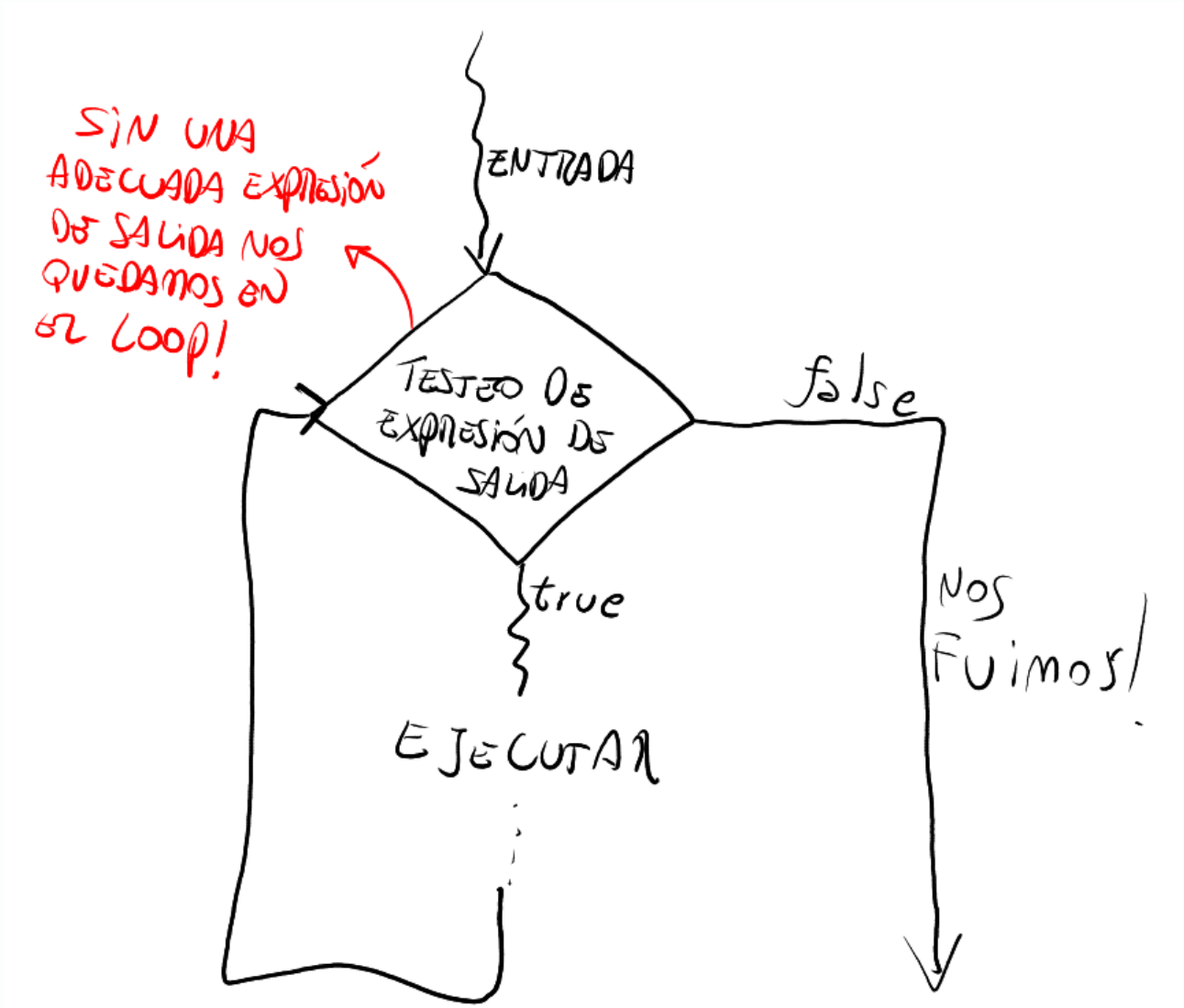
Estructura de control…



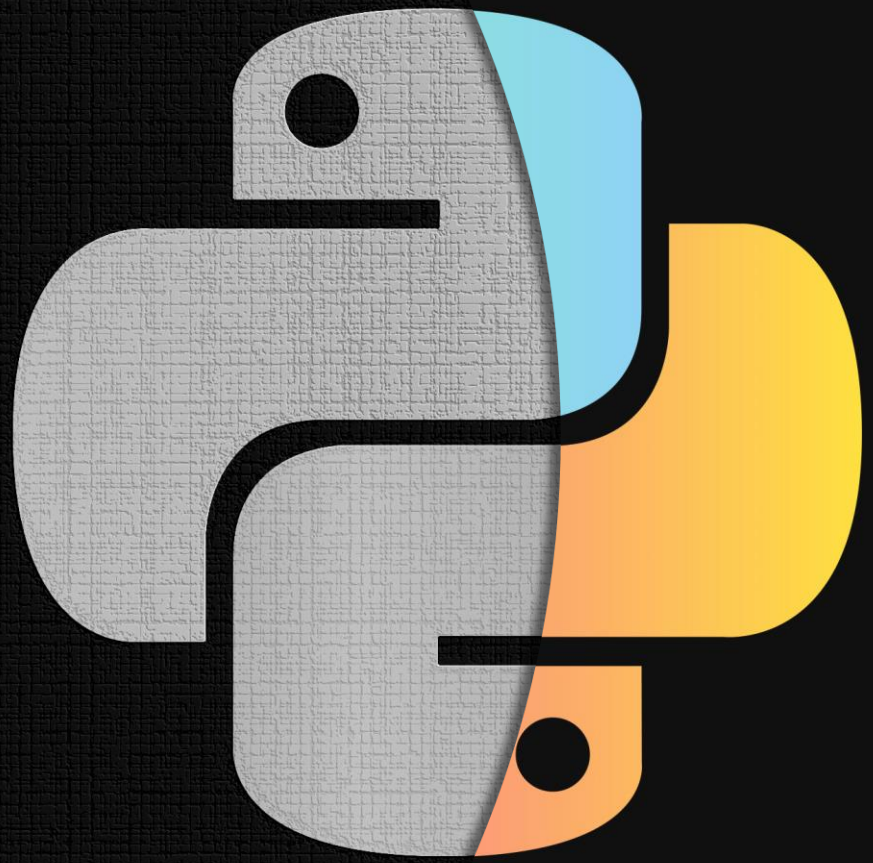

```
lts = np.zeros((len(hiperParams["g
rResults = list()

ed = np.random.randint(100)

for i, kernel in enumerate(hiperParams["k

    if kernel != "linear":
        for j, gamma in enumerate(hiperPa

            for k, C in enumerate(hiperPa
                #Instanciamos el modelo p

                svm = SVMTrainingModule(t
                        nnumberChannels =

                modelo = svm.createSVM(ke

                sampleFrec, signalPSD  =
                                        c

                metricas = svm.trainAndVa
                accu = metricas["modelo_t
                rbfResults[j,k] = accu

                clasificadoresSVM[kernel]
    else:
        for k, C in enumerate(hiperParams

            svm = SVMTrainingModule(train
                    nnumberChannels = 1,

            modelo = svm.createSVM(kernel

            sampleFrec, signalPSD  = svm.
                                        c
```

# Sentencia *while*

Estructura de control…




```
urn self.sessionStatus

n():

initialTime = time.time()#/1000
time.sleep(2)

"""
#creamos un objeto ArduinoCommunicatio
#entre arduino y nuestra PC en el COM3
#n trials.
#Pasado estos n trials se finaliza la
#En el caso de querer ejecutar Trials
#debe hacerse trials = None (default)
"""
ard = ArduinoCommunication('COM10', tr
                           timing = 10
time.sleep(1)
ard.iniSesion()

#Simulamos que enviamos un comando de
ard.systemControl[2] = b'2'

while ard.generalControl() == b"1":
    pass

ard.endSesion()
ard.close() #cerramos comunicación ser

opTime = time.time()#/1000

(f"Tiempo transcurrido en segundo

== "__main__":
```

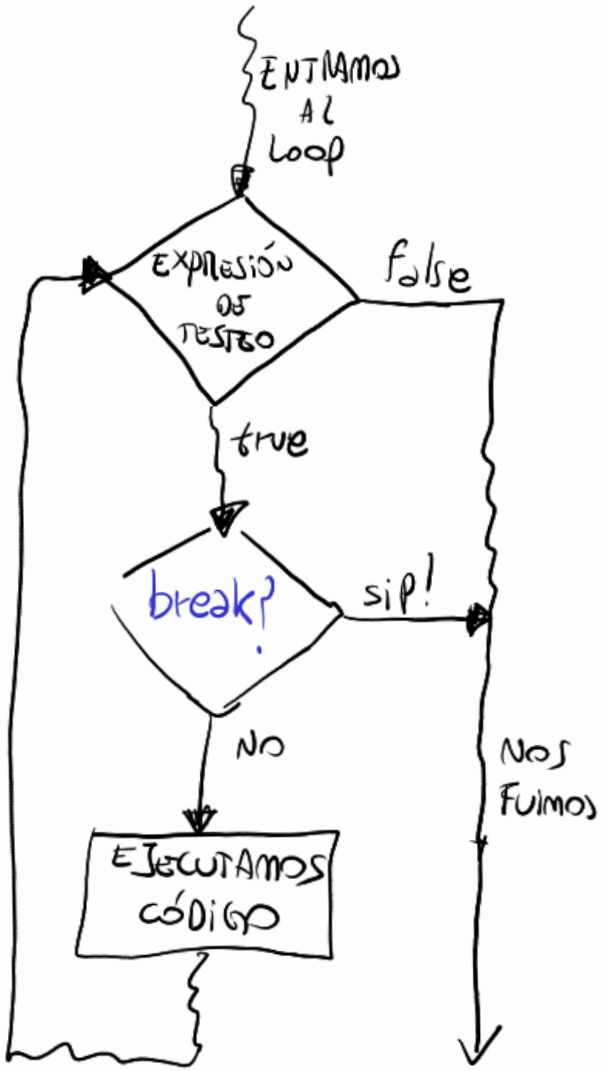


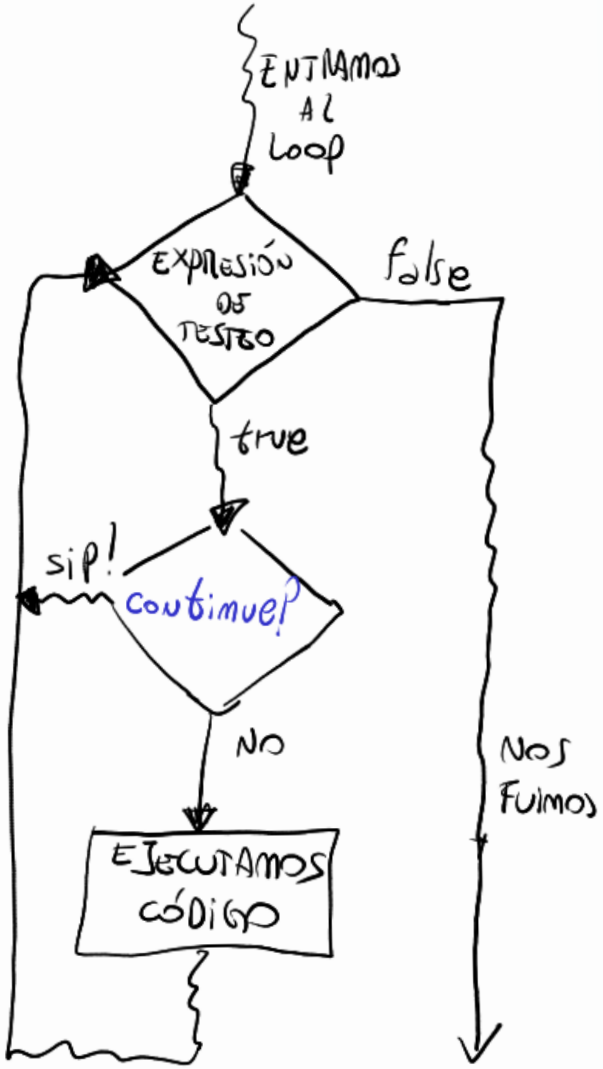
# Break y Continue

# Sentencias *break* y *continue*

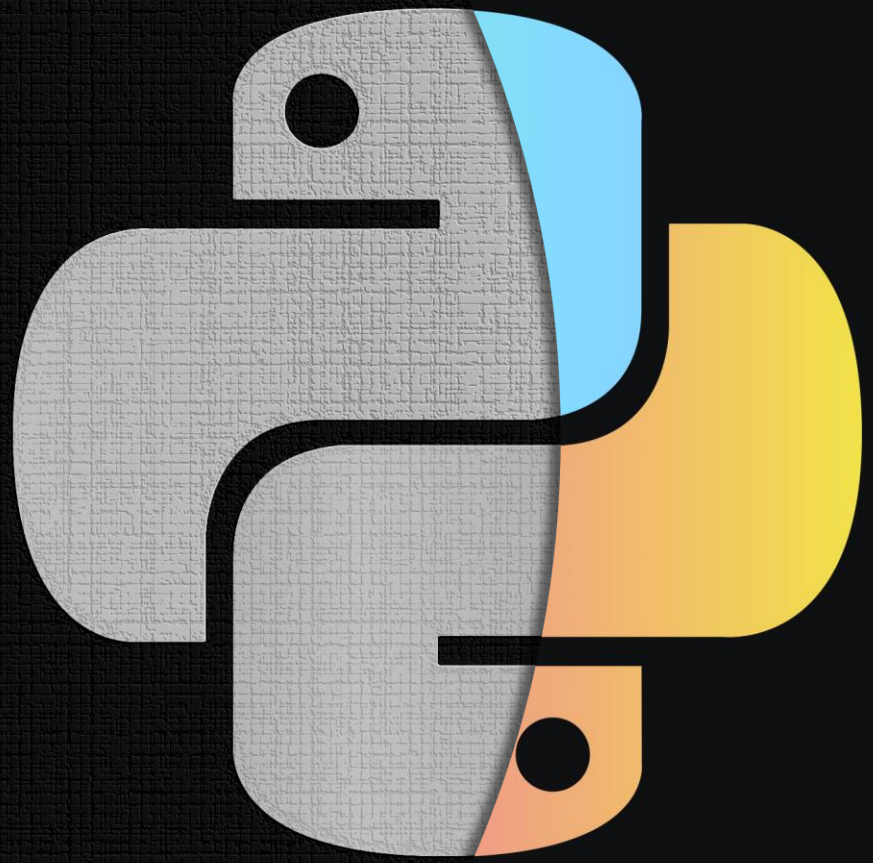
En algunos casos si se cumple cierta condición quisiéramos salir de un loop sin evaluar la expresión de control del mismo o bien no ejecutar cierta parte del código.

¿CUAL ES LA DIFERENCIA ENTRE SI?




```
            self.trial
            self.count

        return self.tr

    def generalControl
        """Función par

        if self.system

            if not sel
                self.t

            if self.ti
                self.t
                self.t

        elif self.syst

            if not sel
                self.t

            if self.ti
                self.t
                self.t

        else:
            self.endSe

        return self.se

ef main():

    initialTime = time
    time.sleep(2)

    """
```



# Iteradores

# Iteradores *enumerate()* y *zip()*

En algunas ocasiones en un *for* es útil iterar sobre el elemento $i\_ésimo$ y también queremos el índice que está siendo iterado, para esto usamos *enumarate()*.

*enumerate(iterable, start=0)*

A veces quisiéramos iterar varios objetos iterables en paralelo, esto lo podemos hacer con *zip()*. Esta función devuelve una lista de tuplas.

*zip(*iterables, strict=False)*

Prestar atención al operador * dentro de la función zip().

```
        self.trial
        self.count

    return self.tr

def generalControl
    """Función par

    if self.system

        if not sel
            self.t

        if self.ti
            self.t
            self.t

    elif self.syst

        if not sel
            self.t

        if self.ti
            self.t
            self.t

    else:
        self.endSe

    return self.se

ef main():

    initialTime = time
    time.sleep(2)

    """
```

# Tipos de datos

# Tipos numéricos

- Enteros (*int*): Números positivos y negativos no decimales. Python gestiona el tamaño de memoria del número de manera automática.

- Flotantes (*float*): Se utilizan para representar números reales. No tienen precisión infinita. Cuidado con usar variables flotantes dentro de estructuras de control.

- Complejos (*complex*): Python permite realizar operaciones básicas con números complejos.

# Sets

Los sets en Python permiten almacenar cualquier tipo de datos y poseen las siguientes características,

- Sus elementos son únicos.
- Son desordenados, eso significa que sus elementos no mantienen el orden una vez creado el set.
- Los elementos que lo conforman deben ser inmutables (no podríamos poner una lista como elemento de un set).

Los set se crean mediante el objeto *set()* o bien pasando los datos entre {}.

```
>> nombres = {'ragnar', 'floki', 'lagertha', 'bjorn’}
>> emptySet = set()
```

Los sets no pueden ser indexados

Los sets son iterables.

# Tuplas

Las *tuplas* son datos secuenciales inmutables. Se definen entre paréntesis o con *tuple()*.

```
>> tup1 = tuple((1,2,3))
>> tup2 = (4,5,6)
>> tuplasAnidadas = (tup1,tup2)
```

Podemos acceder al elemento *i-ésimo* de una tupla mediante indexación.

Las tuplas son iterables y soportan slicing.

# Diccionarios

Los diccionarios son colecciones que almacenan datos con un par *llave/valor*.

Podemos crear diccionarios de la siguiente manera,

```
>> dict1 = dict([("key1",1), ("key2",2), ("key3",3)])
>> dict2 = {"key1":4, "key2":5, key3":6}
```

Son indexados mediante un *key*.

Son mutables y dinámicos.

# Listas

Las *listas* son un tipo de datos que permiten almacenar datos de cualquier tipo. Son mutables y dinámicas. Los elementos de una lista son secuenciales y ordenados.

Las listas se crean con el objeto *list* o bien utilizando brackets [].

```
>> empty_list = []
>> listaDeCosas= [1, “hola”, {‘carrera’: ‘biomédica’}, empty_list, 3.14]
>> anotherEmptyList = list()
```

Podemos acceder al *i-ésimo* ítem de la lista por indexación utilizando [i].

Las listas en Python son objetos y por lo tanto poseen métodos para interactuar con ellas.

Las listas son iterables y soportan slicing.

# Comprensión de listas

La comprensión de listas o “List Comprehension” o “Listcomps” es una expresión que produce una lista a partir de aplicar una expresión/método/función a cada elemento de una colección.

Su sintaxis es,

[*expresión* for *ítem* in *colección*]

1. Son más rápidas que recorrer una lista con for y aplicar la expresión deseada.
2. Permite leer el código de manera más sencilla.
3. Esta funcionalidad es realmente potente porque podemos usar expresiones lógicas (if, or, and) para crear filtros.

# Slicing

Una característica común a las listas, tuplas y string es la posibilidad de realizar *slicing.*

El slicing se hace de la siguiente manera

>> secuencia [*índiceInicio* : *índiceFinal* : *saltos*]

Lo anterior puede ser usado para formar una nueva secuencia desde *índiceInicio* hasta *índiceFinal* con cierta cantidad de *saltos.*

Un ejemplo puede ser

```
>> nums = [0,1,2,3,4,5,6,7,8,9]
>> print(nums[2:6:2]) # imprime [2,4]
>> [2,4]
```

```python
study = UC(name = "PDA", studyHs = 8)
mate = Mate() #nuevo mate

while study.notDone():
    study.meterle()
    mate.cebar()
    if mate.termoEmpyt():
        mate.refill()
```

UTEC
Universidad Tecnológica

Ingeniería Biomédica

# Programación Digital Avanzada

## Módulo 2 – 1ra parte

Mag. Bioing. Baldezzari Lucas

Ingeniería Biomédica
7mo Semestre

2022

V1.0