# Lenguaje de Programación Python

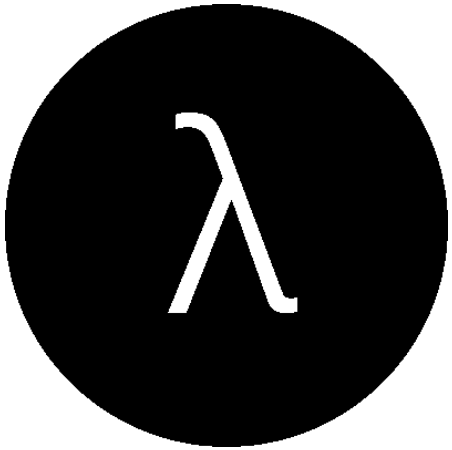
## Soporte Funcional: Llamables

Dr. Mario Marcelo Berón
Argentina Programa
Universidad Nacional de San Luis

## Llamables

- Funciones regulares creadas con *def* con nombre asignado en tiempo de definición.
- Funciones anónimas creadas con *lambda*.
- Instancias de clases definidas con *__call__*.
- Clausuras retornadas por fábricas de funciones.
- Métodos estáticos de instancias definidos utilizando el decorador @staticmethod o __dict__.
- Funciones generadoras.

# Programación Funcional- Funciones Regulares



```
>>>def hola_1(nombre):
...     print("Hola:", nombre)

>>>hola1("David")
Hola David
```

```
>>>hola_2= lambda nombre: print("Hola", nombre)
>>>hola("David")
 hola David
```

## Comentario - Importante

Ambas funciones tienen un atributo *__qualname__* que indica el paso al nombre del objeto destino.

```
>>>hola_1.__qualname__
 'hola_1'
>>>hola_2.__qualname__
 '<lambda>'
```

# Programación Funcional- Lambdas


**Comentario - Importante**

Un programador que desea focalizar sobre el estilo funcional de programación puede intencionalmente decidir escribir muchas funciones puras para permitir el razonamiento matemático y formal.

# Programación Funcional- Clausuras

## Clausura

Operaciones con datos. Se enfatiza inmutabilidad y funciones puras.

## Ejemplo

```
def hacerSumador(n):
 def sumador(m):
  return m + n
 return sumador
sumar5_f = hacerSumador(5) # "funcional"
>>> sumar5_f(10) #15
```

## Ejemplo

```
#Seguramente no es el comportamiento que se espera.
>>> sumadores = []
>>> for n in range(5):
...     sumadores.append(lambda m: m+n)
>>> [sumador(10) for sumador in sumadores]
[14, 14, 14, 14, 14]
>>> n = 10
>>> [sumador(10) for sumador in sumadores]
[20, 20, 20, 20, 20]
```

## Ejemplo

```
>>> sumadores = []
>>> for n in range(5):
....     sumadores.append(lambda m, n=n: m+n)
>>> [sumadores(10) for sumador in sumadores]
[10, 11, 12, 13, 14]
>>> n = 10
>>> [sumador(10) for sumador in sumadores]
[10, 11, 12, 13, 14]
>>> sumar_4 = sumadores[4]
>>> sumar_4(10, 100)
110
```


**Comentario - Importante**

Es importante notar que se usa el truco de usar argumentos de palabra clave para cambiar el valor lo cual reduce la confusión. El valor predominante para la variable nombrada debe pasarse explícitamente en la llamada misma, ligarse en algún lugar remoto en el flujo del programa.


**Comentario - Importante**

Todos los métodos de clases son llamables. Para muchos llamar a un método de una instancia va en contra del estilo de programación funcional. Usualmente se usan métodos porque se desea referenciar a datos mutables que están empaquetados en los atributos de una instancia, y por eso en cada llamado de un método produce un resultado diferente que varía independientemente de los argumentos pasados.

```
class Auto(object):
 def __init__(self):
  self._velocidad = 100



 @property
 def velocidad(self):
  print("La velocidad es:"
        self._velocidad)
  return self._velocidad

 @velocidad.setter
 def velocidad(self, valor):
  print("Inicializar a:", valor)
  self._velocidad = valor
```

**Comentario - Importante**

Los accesores creados con el decorador *@property* o de otro modo, técnicamente son llamables, aunque los accesores son llamables con un uso limitado (desde una perspectiva de la programación funcional). Ellos no tienen artumentos como los *getters* y no retornan un valor como los *setters*.

```
>>> class IntCharlatan(int):
      def __lshift__(self, otro):
        print("Desplazar", self, "por", otro)
        return int.__lshift__(self, otro)
>>> t = IntCharlatan(8)
>>> t << 3
```

**Comentario - Importante**

Todo operador en Python es básicamente un método. Aunque si bien produce un *Lenguaje Específico del Dominio* más legible, el cual define significados especiales para los operadores no incorpora mejoras para las capacidades de los llamadas a funciones.

# Programación Funcional- Métodos Estáticos

```
import math
class TriánguloRectángulo(object):
  """Clase usada solamente como espacio de nombres
    para funciones relacionadas"""
  @staticmethod
  def hipotenusa(a, b):
    return math.sqrt(a**2 + b**2)
...
```

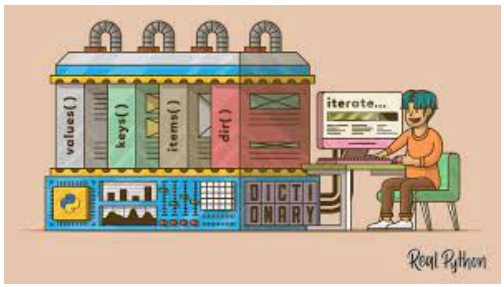
## Comentario - Importante

Uno de los usos de las clases y sus métodos que está más estrechamente alineado con el estilo de programación funcional es usarlos simplemente como espacio de nombres para mantener una variedad de funciones relacionadas.

# Programación Funcional-Iteradores

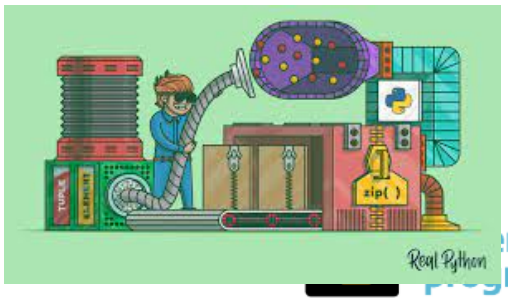
***Iterador***: objeto que representa una corriente de datos. Este objeto retorna un dato a la vez.

- Debe soportar el método ___next()___ que siempre retorna el próximo elemento en la corriente de datos.

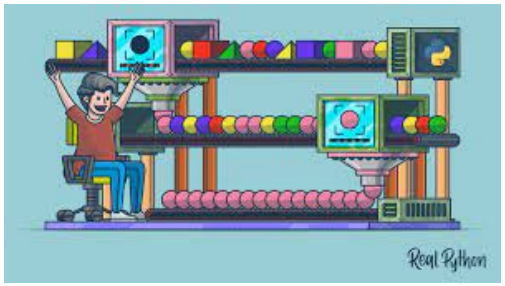

- Si no existen más elementos en la corriente de datos el método ___next()___ retorna una excepción.

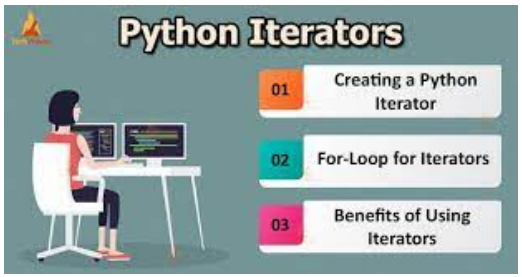
- La función *iter()* toma un objeto arbitrario e intenta retornar un iterador.



# Programación Funcional-Iteradores

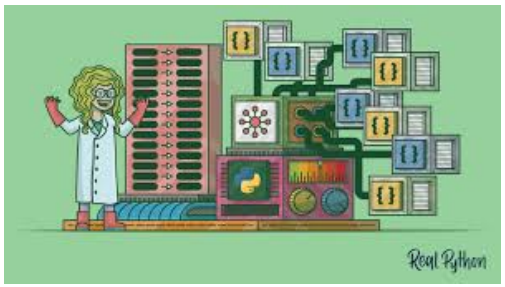
- Si el objeto no soporta iteración entonces la función retorna una excepción *TypeError*.




- Varios tipos de datos de Python soportan iteraciones. Ejemplo: Listas y Diccionarios.

- Un objeto se denomina iterable si se permite que se obtenga un iterador.



# Programación Funcional-Iteradores

- Pueden ser materializados como listas o tuplas usando los constructores list() o tuple().

## Ejemplo

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

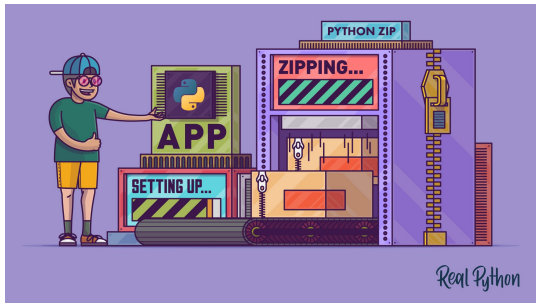
## Ejemplo

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

- Soporta el desempaquetado de secuencias.

# Programación Funcional-Iteradores

- Python espera objetos iterables en diferentes contextos, el más importante es la sentencia *for X in Y* *Y* debe ser iterable.





- El iterador solo puede ir hacia adelante no hay una forma de ir hacia atrás, inicializar el iterador o hacer una copia del iterador