CURSOS INTERSEMESTRALES

PROTECO

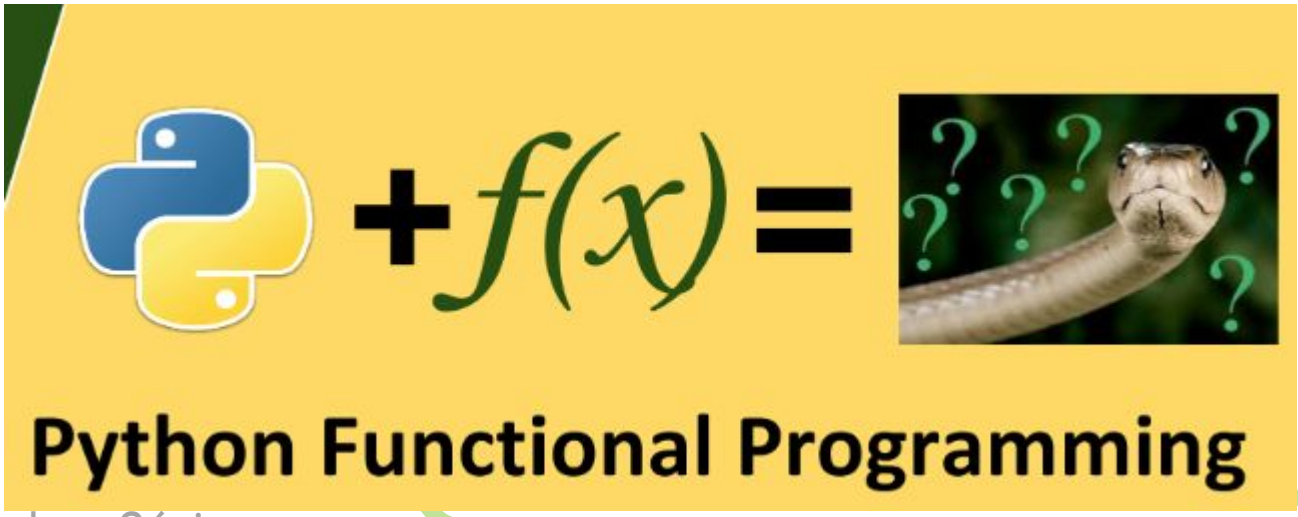
Python Básico

Programación Funcional

Junio de 2016

Paradigma funcional

- El paradigma de programación funcional es uno de los fundamentales entre los llamados de programación declarativa. La programación funcional 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.



Características

- Se dice que una función (f x y z) tiene un efecto colateral si los valores de x, y, y/o z cambian en el entorno de llamada durante la aplicación de la función a sus argumentos, no se hacen asignaciones a las variables globales, no tiene efectos colaterales.

Características

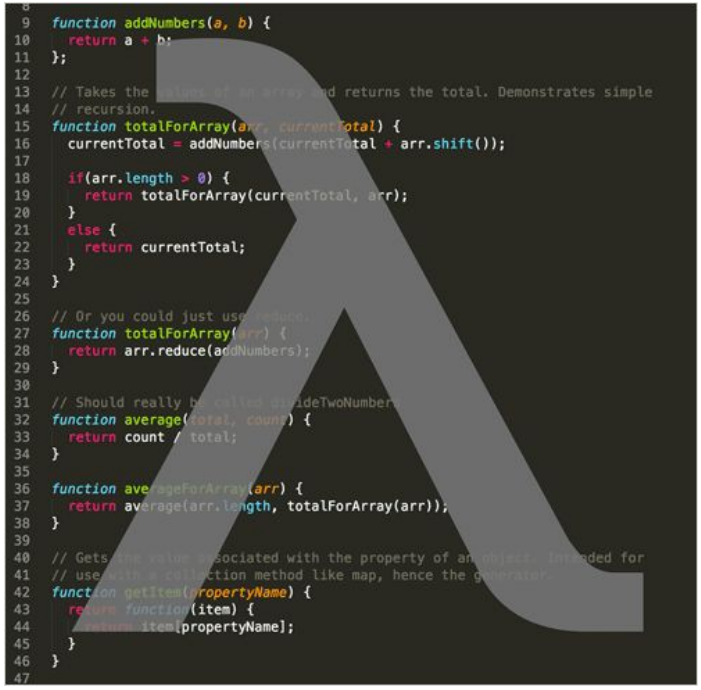
- Ausencia de efectos colaterales, todas las variables son inmutables
- El valor de una expresión sólo depende de los valores de sus subexpresiones, si las tiene.

```
resultado = func1(a) + func2(a)
```

*Podríamos ejecutar func1 y func2 en paralelo porque sabemos que “a” no se va a modificar

Características

- Ausencia de efectos colaterales, todas las variables son inmutables
- El valor de una expresión sólo depende de los valores de sus subexpresiones, si las tiene.

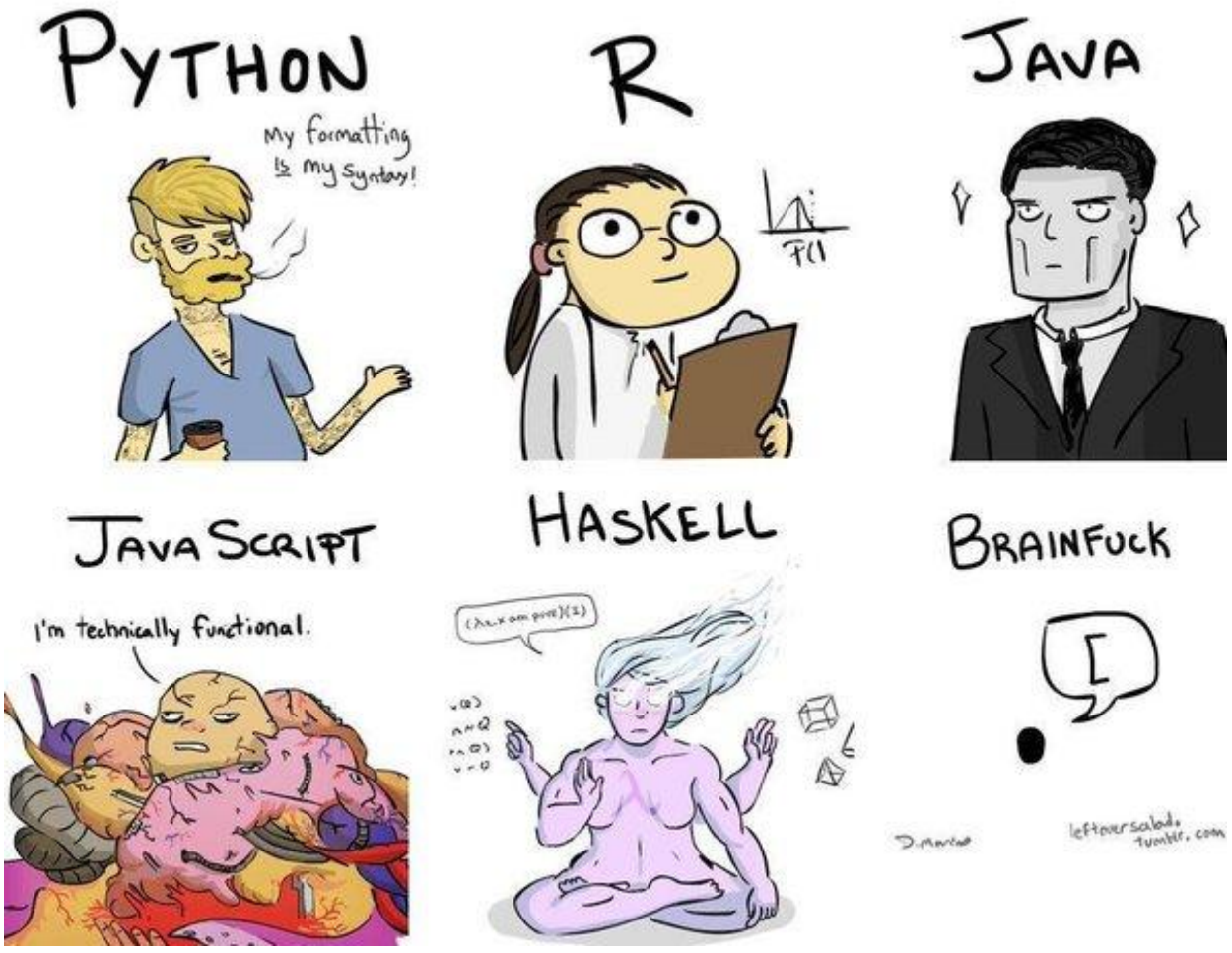


```
function addNumbers(a, b) {
  return a + b;
};

// Takes the values of an array and returns the total. Demonstrates simple
// recursion.
function totalForArray(arr, currentTotal) {
  currentTotal = addNumbers(currentTotal + arr.shift());

  if(arr.length > 0) {
    return totalForArray(currentTotal, arr);
  }
  else {
    return currentTotal;
  }
}

// Or you could just use reduce.
function totalForArray(arr) {
  return arr.reduce(addNumbers);
}

// Should really be called divideTwoNumbers
function average(total, count) {
  return count / total;
}

function averageForArray(arr) {
  return average(arr.length, totalForArray(arr));
}

// Gets the value associated with the property of an object. Intended for
// use with a collection method like map, hence the generator.
function getItem(propertyName) {
  return function(item) {
    return item[propertyName];
  }
}
```

Ventajas

- Una de las mayores ventajas con la programación funcional es que el orden de ejecución de las funciones libre de efectos secundarios para permitir la simultaneidad (procesamiento en paralelo) de una manera muy transparente.
- Y debido a que las funciones en lenguajes funcionales comportan de manera muy similar a las funciones matemáticas, es muy fácil de traducir, aquellas a los lenguajes funcionales. En algunos casos, esto puede hacer que el código sea más legible.

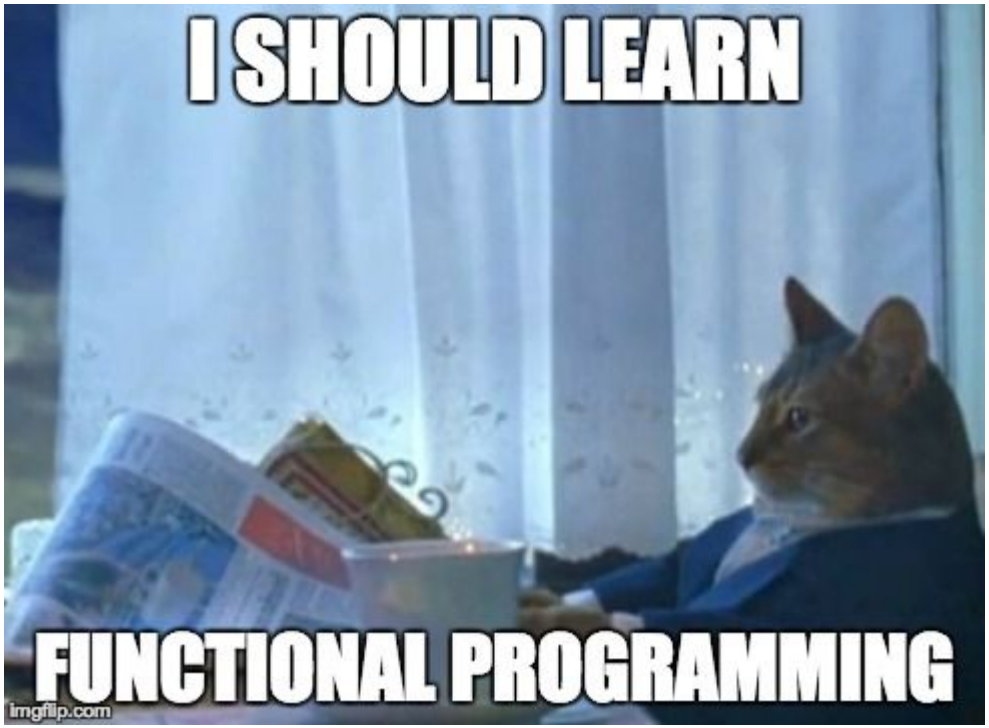
Diferencia con otros lenguajes



Ventajas?

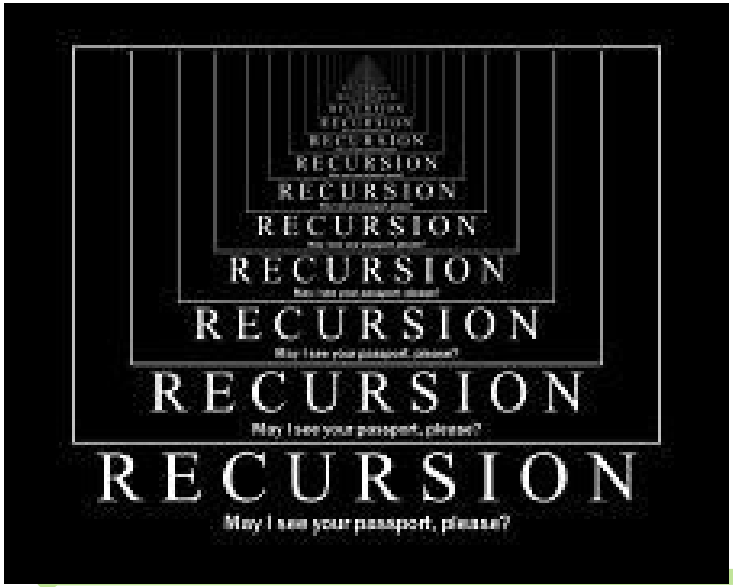
- Quizás no se vea ahora el impacto de esta forma de programar, pero ayuda a ciertas cosas:
- Una forma de encontrar nuevas abstracciones (funciones superiores) que a su vez...
  permiten reutilizar código
- Separar la lógica por niveles y así tener funciones "incompletas" en cuanto a que les falta una parte de su comportamiento que se completará con parámetros. Es decir que estamos parametrizando las funciones.

Así que....



Buenas Prácticas

- Para asegurarse de que estamos usando las bases de este paradigma podemos seguir estas buenas prácticas:
- Todas las funciones al menos deben recibir un argumento
- Todas las funciones deben regresar un dato u otra función
- ¡No hay loops!



Lambdas

- Funciones de una sola línea también funciones anónimas
- Su sintaxis es la siguiente:
- función=(lambda args : operación)
- En la operación no puede haber más que funciones, no podemos usar for o while.
- Ejemplo:

```
cubo=( lambda x : x**3 )
```

Funciones de alto orden: map, filter y reduce

- Estas funciones nos permiten pasar una función completamente sobre todos los elementos de una lista
- Regresan una lista nueva sin afectar la que usemos
- Su sintaxis es la siguiente:

  mfrz(función, lista)

- función puede ser una lambda

  *mfrz=map,filter,reduce o zip

Listas por Comprensión

- Forma potente de crear listas
- Aplica una función a cada elemento de la lista
- Su sintaxis es la siguiente:
- [**expresión** for **nombre** in **lista**]
- La **expresión** debe ser capaz de operar sobre toda la **lista**
- Si **lista** (puede ser un generador) contiene objetos indexables entonces **nombre** debe de tener un patrón que "empate" con sus elementos.

Generadores

- Forma potente de crear iteradores
- Como una función pero usan **yield** en vez de **return**
- Cada vez que se llama a **__next__()** el generador continua donde se quedó
- Permanece un estado de las variables en la memoria, y la última línea que se ejecutó.

Decoradores

- Funciones que toman como parámetro una función y regresan una función
- Crea un “envoltura” alrededor de la función a decorar.
- Se declaran como funciones **@decorador** y se coloca esta etiqueta sobre la funcion a decorar.
- Cambian el comportamiento de un función o clase sin modificarla.

Ligas de interés

- Documentación más extensa
- https://docs.python.org/3/howto/functional.html
- ¿Qué define a un lenguaje funcional?
- https://clojurefun.wordpress.com/2012/08/27/what-defines-a-functional-programming-language/
- ¿Porqué python no es muy bueno para PF?
- http://stackoverflow.com/questions/1017621/why-isnt-python-very-good-for-functional-programming

def fibo(n)

- Tarea :

Generador de la Serie de Fibonacci