

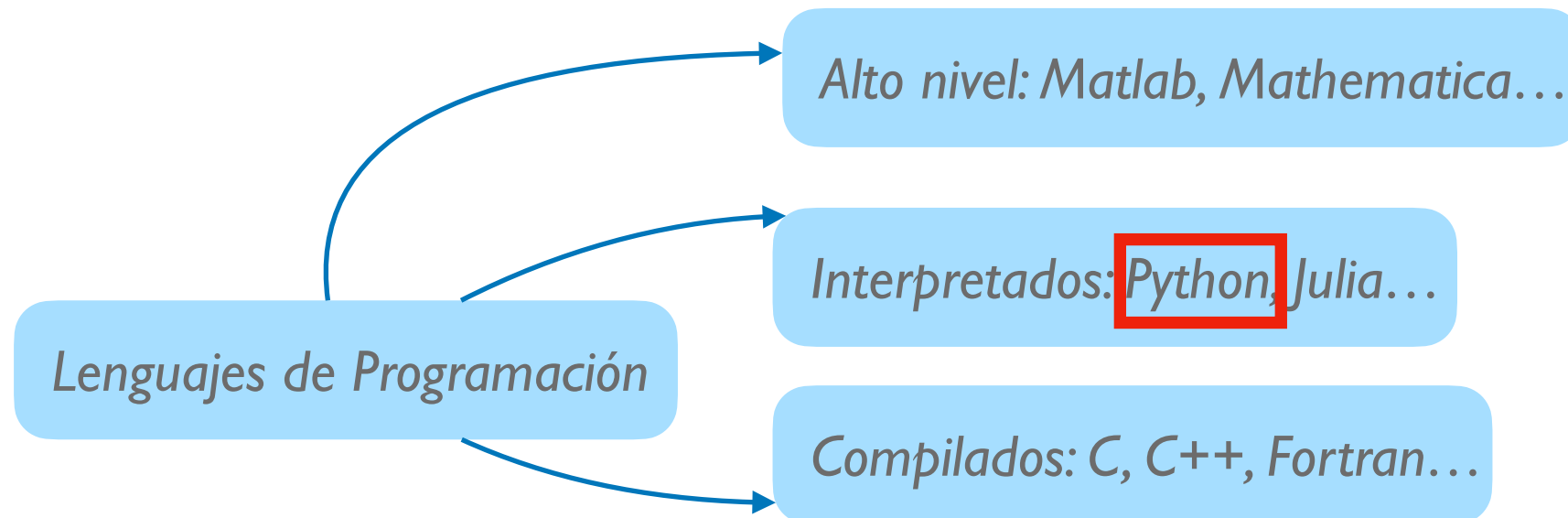
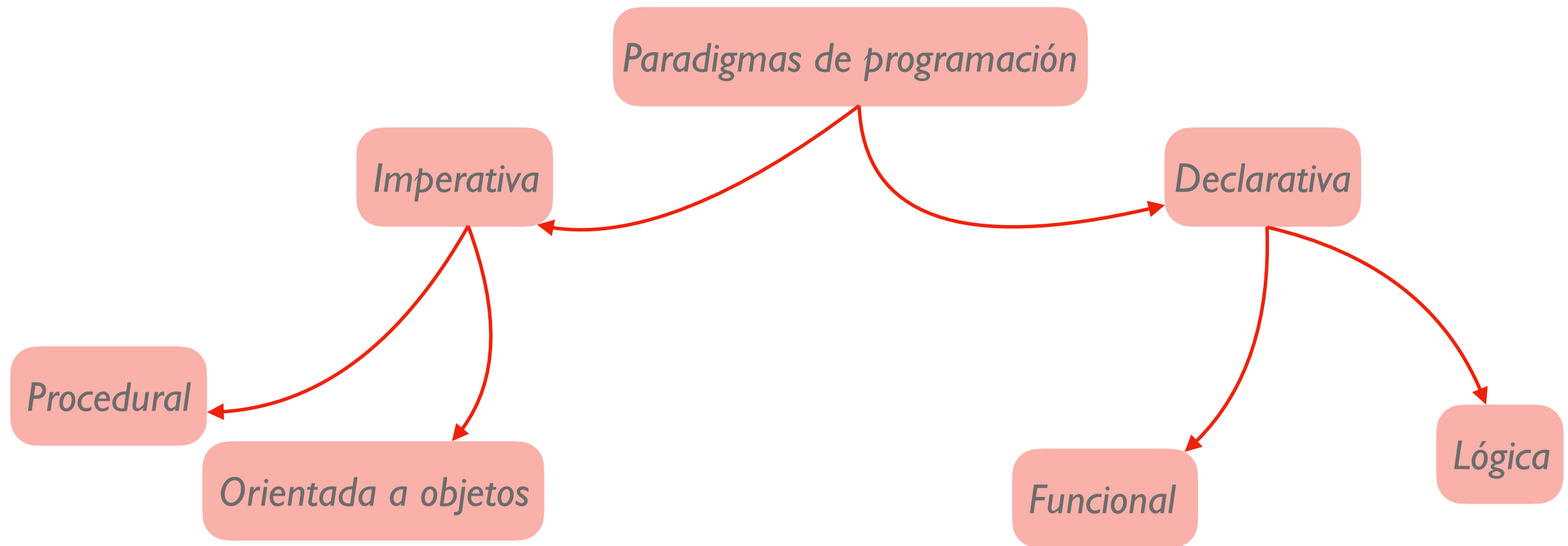


UNIVERSIDAD DE COSTA RICA

PROGRAMACIÓN INTERPRETADA: PYTHON

Prof. Marlon Brenes y Prof. Federico Muñoz
Escuela de Física, Universidad de Costa Rica

Programación Intermedia: Guía de Curso



Lenguajes interpretados

- Un programa llamado *interpretador* toma las instrucciones y las traduce a lenguaje de máquina
- No existe etapa de compilación
- A diferencia de un compilador, el cual toma todas las instrucciones y genera un ejecutable en lenguaje de máquina, un interpretador toma el código línea por línea y las ejecuta
- Razones para usar lenguajes interpretados:
 - ↳ • Facilidad en la curva de aprendizaje
 - El número de bibliotecas es mucho más grande, en general
 - Excelente soporte para aplicaciones científicas
 - Ciclos de desarrollo más rápidos

Python: opciones de diseño

- Python es un lenguaje multi-paradigma:
 - └─→ • Programación estructurada, OOP y funcional son posibles
- Ningún paradigma es forzado por el lenguaje (al contrario de, e.g., Java)
- Sintaxis usualmente limpia, fácil de entender pero puede llegar a ser difícil de optimizar
- Altamente extendible. La funcionalidad base es relativamente pequeña, mientras que las bibliotecas personalizadas son extensas

Hola mundo

```
1 #!/usr/bin/env python
2
3 print('Hello, world!')
```

- Ejecución:

```
(base) mbrenes@nia-login03:Semana_04$ ll
total 1
-rw-r----- 1 mbrenes dvira 48 Apr  4 00:20 hello.py
(base) mbrenes@nia-login03:Semana_04$ chmod +x hello.py
(base) mbrenes@nia-login03:Semana_04$ ./hello.py
Hello, world!
(base) mbrenes@nia-login03:Semana_04$
```

Esto se hace solo una vez!



Type system (sistema de interpretación)

```
1 # Strong Typing
2
3 'foo' + 5 # Esto es un error al runtime
```

```
1 # Dynamic Typing
2
3 a = 'foo'
4 b = 2 * a # b = 'foofoo'
5 a = 5
6 b = 2 * a # b = 10
```

```
1 # 'Duck' Typing
2
3 def foo(a, b):
4     return a + b
5
6 # Las funciones toman cualquier tipo
7 # como argumento. Si el tipo no calza,
8 # se devuelve un error al runtime
```

Sintaxis

- El espacio en blanco es significativo! (TAB)

- 
- Los espacios denotan distintos scopes

```
1 if(a > b){  
2     foo();  
3     bar();  
4 }  
5  
6 baz();
```

C/C++

```
1 if a > b:  
2     foo()  
3     bar()  
4  
5 baz()
```

Python

Flujos de control

```
1 for i in list_i:  
2     baz(i)
```

```
1 while a > b:  
2     foo()  
3     bar()
```

```
1 if a > b:  
2     foo()  
3 elif b != c:  
4     bar()  
5 else:  
6     baz()
```

```
1 pass  
2  
3 break  
4  
5 continue
```


Funciones

```
1 def func(a, b, c):  
2     a = 3 * b  
3     return a + b - c
```

- Las funciones pueden ser pasadas como argumentos de funciones y las funciones pueden ser valores de retorno

└─ • En C/C++ esto requiere de bibliotecas externas (e.g., boost)

```
1 def times_n(n):  
2     def helper(x):  
3         return n * x  
4     return helper  
5  
6 times_6 = times_n(6)  
7 a = times_6(7) # a = 42
```

Manejo de excepciones

- Similar a la estructura `try/catch` en C++

└─→ • La sintaxis en este caso es `try/except`

```
1 # Usualmente es mejor utilizar:
2 try:
3     a = read_my_data()
4 except:
5     print("Correupted data")
6
7 # que utilizar:
8 if consistent_data:
9     a = read_my_data()
10 else:
11     print("Corrupted data")
```

Expresiones

- Hay que tener cuidado con la división numérica

└─→ • El resultado de la operación depende del tipo interpretado

```
1 # En Python 3:
2 a = 5 / 3 # a = 1.666666666667
3
4 a = 5 // 3 # 1
5
6 # Los booleanos y operaciones booleanas
7 # se escriben explícitamente
8 and or not
9 True False
```

Hileras de caracteres

- Los delimitadores de las hileras de caracteres son el operador ' y ", dependiendo de la necesidad

- 
- Secciones de código se pueden comentar encapsulando la sección con `"""` o `'''`

```
1 a = "Me dijo 'Hola, Marlon', ayer por la tarde"
2
3 # Comentarios de una sola línea con '#'
4 """
5 Más de una línea
6 de código
7 puede ser comentada
8 de esta manera
9 """
```

Formatos de impresión

- Existen dos formas de realizar distintos formatos de variables y caracteres

- • La primera es mediante el operador %

```
1 print("Hoy comí %d %s para el desayuno" % (2, "manzanas"))
2
```

- • La segunda es mediante el acceso al método .format()

```
1 print("Hoy comí {} {} para el desayuno".format(2, "manzanas"))
2
```

- • La segunda forma es más flexible

```
1 texto = "Hoy comí {num} {food}. Si, realmente comí {num}."
2 respuesta = texto.format(num = 12, food = "manzanas")
3
4 print(respuesta)
```

Arreglos y colecciones de datos

- En Python, no existe el concepto de pasar por valor/referencia/puntero

└─→ • En lugar de esto, algunas estructuras son mutables y otras inmutables

```
1 #!/usr/bin/env python
2
3 # Esto es una lista y es *mutable*
4 # Es decir, si se pasa como argumento de una
5 # función, la función puede modificar su contenido
6 a = [3, 1, 'foo', 12.0]
7 def func(l):
8     l[0] = 5
9     return l
10 print(func(a)) # [5, 1, 'foo', 12.0]
11
12 # Esto es un tuple y es *inmutable*
13 b = (3, 1, 'foo', 12.0)
14 print(func(b)) # Runtime error. b es inmutable
```

- mutable_vs_inmutable.py

Arreglos y colecciones de datos

- Los diccionarios son mutables

```
1 #!/usr/bin/env python
2
3 # Los diccionarios son mutables
4 d = {'name': 'Marlon', 'age': 21}
5
6 print(d['name']) # Marlon
7 print(d['age']) # 21
8
9 def func(dd):
10     dd['age'] = 20
11     return dd
12
13 print(func(d))
```

- dict.py

Las listas se pueden recorrer de forma sencilla

```
1 #!/usr/bin/env python
2
3 pts = [
4     (1, 3),
5     (5, 6),
6     (9, 10)
7 ]
8
9 for i in pts:
10     print(i)
11
12 for x, y in pts:
13     print(x, 'and', y)
```

- lists.py

Paquetes externos

- Python dispone de una cantidad increíble de paquetes externos para realizar distintas operaciones
 - └─→ • numpy, scipy, matplotlib, Pandas, TensorFlow, PyTorch...
- Los paquetes se instalan mediante un *package manager*

Ejercicio de laboratorio

- └─→ • Implementar `pi.cpp` en Python