Anuncios

1. Contesten la ECA, tendremos sistema de puntos (a.k.a ECA *Points*)
2. Se han subido cambios de enunciado de la T01.
3. Recuerden hacer *push* frecuentes en su tarea
4. Programatón próximo 12 de septiembre (post 17:00)
5. Hoy tenemos actividad formativa.

Iterables y funciones de orden superior

Semana 03 - jueves 05 de septiembre 2019

¿Qué es la programación funcional?

Iterable e iterador

- **Iterable**

  Implementa `__iter__` o `__getitem__`*

- **Iterador**

  Implementa `__next__` e `__iter__`*

___

Generadores y funciones generadoras

- yield

Métodos

- `__len__`
- `__reversed__`
- `__getitem__`

`___`

Funciones útiles

Estilo pitónico

- `enumerate`
- `zip`
- `reversed`
- `sorted*`

$$N = \{n^2 : \text{n es un entero y } 1 \leq n \leq 10\}$$

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```python
[n**2 for n in range(1, 11)]
```

Comprensión de listas, diccionarios y sets

Estructuras por compresión

```
# lista por extensión
lista = ['1', '4', '55', '65', '4', '15', '90']

# lista por comprensión
int_lista = [int(c) for c in lista]
int_lista_dos_dígitos = [int(c) for c in lista if len(c) > 1]
```

Estructuras por compresión

```
from collections import namedtuple

Película = namedtuple("Pelicula", ["título", "director", "género"])
películas = [Película("Into the Woods", "Rob Marshall",
"Aventura"), ...]

# set por comprensión
directores_acción = {p.director for p in películas if p.género ==
'Acción'}

# diccionario por comprensión
dict_directores_acción = {p.director: p.título for p in películas
if p.género == 'Acción'}
```

Funciones *lambda*

- Funciones que siempre retornan sin declararlo
- Se pueden crear de forma anónima
- Permite ejecutar funciones sin definirlas

Map, filter y reduce

Map, filter, reduce en Python

```
strings = ['tExTo','ESCrito','mUy','mAl']
mapeo_strings = map(lambda x: x.lower(), strings)
print(list(mapeo_strings))

a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
b = [100, 101, 102]

mapeo_ints = map(lambda x, y: x + y, a, b)
print(list(mapeo_ints))
```

Map, filter, reduce en Python

```
def fibonacci(límite):
    a, b = 0, 1
    for _ in range(límite):
        yield b
        a, b = b, a + b

filtrado_impares = filter(lambda x: x % 2 != 0, fibonacci(10))
print(list(filtrado_impares))

filtrado_pares = filter(lambda x: x % 2 == 0, fibonacci(10))
print(list(filtrado_pares))
```

Map, filter, reduce en Python

```python
from functools import reduce

lista = [1, 2, 3, 4]

reduccion_1 = reduce(lambda x, y: x + y, lista)
reduccion_2 = reduce(lambda x, y: x*y, lista)

lista_con_listas = [[1, 2], [3, 4], [5, 6], [7, 8, 9]]
lista_aplanada = reduce(lambda x, y: x + y, lista_con_listas)

conjuntos = [{3, 5, 1}, {4, 3, 1}, {1, 2, 5}, {9, 5, 4, 1}]

union = reduce(lambda x, y: x | y, conjuntos)
print(reduce(lambda x, y: x if x > y else y, union))
```

¿Qué hace la siguiente sentencia?

```
yield from range(500)
```

Funciones son *first-class citizens*

- Son objetos
- Pueden ser pasadas como argumentos
- Pueden ser definidas dentro de otra función

¿Qué es un decorador?

Decorador simple

Decorador

Función a retornar

```
def haz_bold(func):
    def decorada():
        return f"<b>{func()}</b>"
    return decorada

def texto():
    return "Este es un texto."


print(texto())

texto_decorado = haz_bold(texto)

print(texto_decorado())
```

Decorador simple con *syntactic sugar*

Decorador

```
def haz_bold(func):
    def decorada():
        return f"<b>{func()}</b>"
    return decorada
```

Función a retornar

```
@haz_bold
def texto():
    return "Este es un texto."


print(texto())
```

Decorador para función con múltiples argumentos

Decorador

```
def haz_bold(func):
    def decorada(*args, **kwargs):
        return f"<b>{func(*args,**kwargs)}</b>"
    return decorada
```

Función a retornar

```
@haz_bold
def texto(a1, a2):
    return f"Este es un texto {a1} y {a2}."


print(texto("corto", "de ejemplo"))
```

Decorador con argumentos

Constructor del decorador

Decorador a retornar

Función a retornar

```
def agregar_tag(tag):
    def decorador(func):
        def decorada(*args, **kwargs):
            return f"<{tag}>{func(*args, **kwargs)}</{tag}>"
        return decorada
    return decorador
```

```
@agregar_tag("b")
def texto(a1, a2):
    return f"Este es un texto {a1} y {a2}."


print(texto("corto", "de ejemplo"))
```

Usando varios decoradores

```
@agregar_tag("i")
@agregar_tag("b")
def texto(a1, a2):
    return f"Este es un texto {a1} y {a2}."


print(texto("corto", "de ejemplo"))
```

¿Cómo influye el orden de los decoradores?

¿Qué tan diferente es decorar funciones generadoras?

```
def texto(a1, a2):
    for x in range(5):
       yield f"{x} - Este es un texto {a1} y {a2}"
```

El decorador a aplicar también debe entregar un generador

Decorador para función generadora

```
def agregar_tag(tag):
    def decorador(gen):
        def decorada(*args, **kwargs):
            for x in gen(*args, **kwargs):
                yield f"<{tag}>{x}</{tag}>"
        return decorada
    return decorador


@agregar_tag("b")
def texto(a1, a2):
    for x in range(5):
        yield f"{x} - Este es un texto {a1} y {a2}"
```

¿Para qué decoradores?

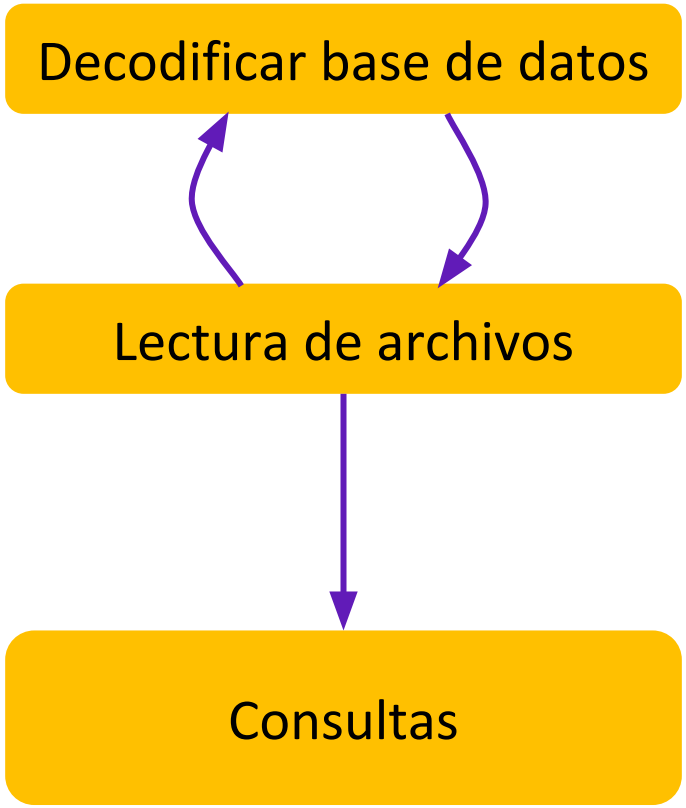
- *Wrapping* de funciones
- *Logging / analytics*
- Verificaciones en *runtime*

Actividad

1. En el *syllabus*, vayan a la carpeta “Actividades” y descarguen el enunciado de la actividad 3 (AC03) https://github.com/IIC2233/syllabus
2. Recuerden hacer *commit* y *push* cada cierto tiempo.
3. Para esta AC, es conveniente ir leyendo y programando a medida que encuentren cosas que hacer.
4. Recuerden que esta actividades formativa y tienen hasta las **16:30**.

Cierre

Diagrama de flujo de AC03




```
def decodificar(string)
@desenciptar(decodificar, tipo_archivo)
```

```
def usuarios_por_antiguedad(...)
def ratings_usuarios(...)
def canciones_artista(...)
def rating_promedio(...)
```

Iterables y funciones de orden superior

¿Se podría haber hecho la AC sin decoradores y sin iterables?

Sin decoradores, no hubiésemos podido hacer un cambio importante con una pequeña porción de código, hubiéramos cambiado cada método.

Sin iterables, hubiésemos cargado en memoria todos los archivos durante la ejecución de nuestro código, lo que no sería factible con muchos datos.

Decoradores

Permiten la capacidad de agregar comportamiento extra a **varias** funciones.

```
def desencriptar(funcion_decod, tipo_archivo):
    tipos = {“canciones”: Cancion, “artistas”: Artista, ...}
    tupla = tipos[tipo_archivo]
    def decorador(funcion):
        ...



        ...
    return decorador
```

Decoradores

Permiten la capacidad de agregar comportamiento extra a **varias** funciones.

```
def desencriptar(funcion_decod, tipo_archivo):
    tipos = {“canciones”: Cancion, “artistas”: Artista, ...}
    tupla = tipos[tipo_archivo]
    def decorador(funcion):
        def wrapper(*args, **kwargs):
            ...


            ...
        return wrapper
    return decorador
```

Decoradores

Permiten la capacidad de agregar comportamiento extra a **varias** funciones.

```
def desencriptar(funcion_decod, tipo_archivo):
    tipos = {“canciones”: Cancion, “artistas”: Artista, ...}
    tupla = tipos[tipo_archivo]
    def decorador(funcion):
        def wrapper(*args, **kwargs):
            generador_original = funcion(*args, **kwargs)
            for resultado in generador_original:
                desencriptado = map(funcion_decod, resultado)
                yield tupla(*desencriptado)
        return wrapper
    return decorador
```

Iterables

Permiten recuperar valores de forma dinámica, sin cargarlos en memoria

```
def rating_promedio(nombre_cancion, canciones, ratings):
    id_cancion = list(
        map(
            lambda cancion_1: cancion_1.id,
            filter(
                lambda cancion: cancion.nombre == nombre_cancion,
                canciones
                )
            )
        )
    if id_cancion:
        id_cancion = id_cancion[0] # Debería encontrar solo una
    else:
        return 0 # Si no, entregamos 0
    ...
```

Iterables

Permiten recuperar valores de forma dinámica, sin cargarlos en memoria

```
def rating_promedio(nombre_cancion, canciones, ratings):
    ...
    ratings_cancion = list(
        map(
            lambda prom: int(prom[2]),
            filter(
                lambda rating: rating[1] == id_cancion,
                ratings
                )
            )
        )

    if len(ratings_cancion) == 0:
        return 0
    return sum(ratings_cancion)/len(ratings_cancion)
```