Introducción a la programación

Práctica 7: Funciones sobre listas (tipos complejos)

Contenido de la clase de hoy:

1. Anotaciones de tipado de colecciones
2. Ejercicios de la Guía 7: 1.1, 1.3, 2.1, 2.2

Tipado de tipos básicos

- x: int = 1
- x: float = 1.0
- x: bool = True
- x: str = "test"

Tipado de colecciones

- Depende la versión de Python las anotaciones cambian.
- Para ver la versión usamos en la terminal:
  `python3 --version`
- En Python 3.8 o inferior debemos importar un módulo para tipar.
- En Python 3.9 no hay que importar ningún módulo.

Python 3.8 o inferior

En esta versión de Python los tipos se escriben con la primera letra en mayúscula y debemos importar el nombre del tipo que queremos usar:

```
from typing import List, Set, Dict, Tuple
x: List[int] = [1]
x: Set[int] = {6, 7}
x: Dict[str, float] = {"field": 2.0}
x: Tuple[int, str, float] = (3, "yes", 7.5)
x: Tuple[int, ...] = (1, 2, 3)
```

Python 3.9 o superior

En esta versión de Python los tipos se escriben con la primera letra en minúscula y NO debemos importar el nombre del tipo que queremos usar:

```
x: list[int] = [1]
# Para diccionarios debemos declarar el tipo
# de dato de la clave y los valores
x: dict[str, float] = {"clave": 2.0}

# Para las tuplas debemos decir el tipo
# de dato de todos los valores
x: tuple[int, str, float] = (3, "Si", 7.5)
```

Resumen:

Recordar usar las anotaciones de tipado en todas las variables. Por ejemplo: `def función(numero: int) -> bool:`
Las anotaciones de tipado de las colecciones, listas, diccionarios o tuplas, depende de la versión de Python que usamos. Por ejemplo:

- En Python 3.8 son en mayúscula (ej: List, Dict, Tuple) y hay que importar el tipo: `from typing import List, Dict, Tuple`.
- En Python 3.9 son en minúscula (ej: list, dict, tuple) y no hace falta importar nada.

Sugerimos revisar esta página que resumen las formas de uso:
`https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html`

Ejercicio 1.1: Pertenece

`problema pertenece` (in s: $seq\langle\mathbb{Z}\rangle$, in e: $\mathbb{Z}$) : Bool $\{$
  `requiere:` $\{ True \}$
  `asegura:` $\{ (res = true) \leftrightarrow$ e $\in$ s$\}$
$\}$

Implementar al menos de 3 formas distintas éste problema.

¿La implementación en Python es un `procedimiento` o `función`?
¿Qué diferencia tienen?

Ejercicio 1.1: Pertenece

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[], 1`	`False`
Único elemento y pertenece	`[1], 1`	`True`
Único elemento y no pertenece	`[1], 2`	`False`
Más de un elemento y pertenece	`[1, 2, 3, 4], 3`	`True`
Más de un elemento y no pertenece	`[1, 2, 3, 4], 10`	`False`

Ejercicio 1.1: Pertenece

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[], 1`	`False`
Único elemento y pertenece	`[1], 1`	`True`
Único elemento y no pertenece	`[1], 2`	`False`
Más de un elemento y pertenece	`[1, 2, 3, 4], 3`	`True`
Más de un elemento y no pertenece	`[1, 2, 3, 4], 10`	`False`

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?

Ejercicio 1.1: Pertenece

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[], 1`	`False`
Único elemento y pertenece	`[1], 1`	`True`
Único elemento y no pertenece	`[1], 2`	`False`
Más de un elemento y pertenece	`[1, 2, 3, 4], 3`	`True`
Más de un elemento y no pertenece	`[1, 2, 3, 4], 10`	`False`

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?
Tener en cuenta que el *requiere* es *True*.

Ejemplo de test

```
import unittest
from guia7 import *

class Test_pertenece(unittest.TestCase):
  def test_1(self):
    s = []
    self.assertFalse(pertenece(s, 1))
```

1.3 Suma Total

`problema suma_total` (in s: $seq\langle\mathbb{Z}\rangle$) : $\mathbb{Z}$ {
  `requiere:` { $True$ }
  `asegura:` { *res* es la suma de todos los elementos de *s*}
}

Nota: no utilizar la función `sum()` nativa

Pista: En este ejercicio estaremos usando una variable que acumula el resultado y luego lo devuelve.

1.3 Suma Total

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	0
Un elemento	`[1]`	1
Más de un elemento	`[1, 2, 3, 4]`	10

1.3 Suma Total

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	0
Un elemento	`[1]`	1
Más de un elemento	`[1, 2, 3, 4]`	10

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?

1.3 Suma Total

Elaboremos casos de test para la solución, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	0
Un elemento	`[1]`	1
Más de un elemento	`[1, 2, 3, 4]`	10

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?
Tener en cuenta que el *requiere* es *True*.

Ejemplo de test

```
...
class Test_suma_total(unittest.TestCase):
    def test_1(self):
        s = [1, 2, 3, 4]
        res_obtenido = suma_total(s)
        res_esperado = 10
        self.assertEquals(res_obtenido, res_esperado)
```

Ejercicio 2.1

`problema CerosEnPosicionesPares` (inout s: $seq\langle\mathbb{Z}\rangle$) {

`requiere:` { $True$ }

`modifica:` {s}

`asegura:` { ($|s| = |s@pre|$) y (para todo i entero, con $0 <= i < |s|$, si i es impar entonces $s[i] = s@pre[i]$ y, si i es par, entonces $s[i] = 0$)}

}

Ejercicio 2.2

`problema CerosEnPosicionesPares2` (in s:$seq\langle\mathbb{Z}\rangle$) : $seq\langle\mathbb{Z}\rangle$ {

`requiere:` { $True$ }

`asegura:` { ($|s| = |res|$) y (para todo i entero, con $0 <= i < |res|$, si i es impar entonces $res[i] = s[i]$ y, si i es par, entonces $res[i] = 0$)}

}

Ejercicios 2.1 y 2.2

Elaboremos casos de test para las dos soluciones, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	`[]`
Un elemento	`[1]`	`[0]`
Dos elementos	`[1,2]`	`[0, 2]`
Más de un elemento, longitud par	`[1, 2, 3, 4]`	`[0,2,0,4]`
Más de un elemento, longitud impar	`[1, 2, 3, 4, 5]`	`[0,2,0,4,0]`

Ejercicios 2.1 y 2.2

Elaboremos casos de test para las dos soluciones, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	`[]`
Un elemento	`[1]`	`[0]`
Dos elementos	`[1,2]`	`[0, 2]`
Más de un elemento, longitud par	`[1, 2, 3, 4]`	`[0,2,0,4]`
Más de un elemento, longitud impar	`[1, 2, 3, 4, 5]`	`[0,2,0,4,0]`

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?

Ejercicios 2.1 y 2.2

Elaboremos casos de test para las dos soluciones, por ejemplo:

Caso	**Entradas**	**Esperado**
Lista vacía	`[]`	`[]`
Un elemento	`[1]`	`[0]`
Dos elementos	`[1,2]`	`[0, 2]`
Más de un elemento, longitud par	`[1, 2, 3, 4]`	`[0,2,0,4]`
Más de un elemento, longitud impar	`[1, 2, 3, 4, 5]`	`[0,2,0,4,0]`

Preguntas: ¿Son suficientes los casos? ¿Qué otros agregarían a partir de las entradas posibles?
Tener en cuenta que el *requiere* es *True*.

Ejemplo de test

```
...
class Test_ceros(unittest.TestCase):
  # Test de version inout
  def test_1(self):
    s = [1, 2]
    ceros_en_posiciones_pares(s)
    resultado_esperado = [0, 2]
    self.assertEqual(s, resultado_esperado)

  # Test de version in
  def test_2(self):
    s = [1, 2]
    res = ceros_en_posiciones_pares2(s)
    resultado_esperado = [0, 2]
    self.assertEqual(res, resultado_esperado)
```