

# Examen 2 - CI3641

Joao Pinto 17-10490

## 1. Pregunta 1

Para esta pregunta se seleccionó **Python**. **Python** es un lenguaje de programación de scripting de alto nivel y multi-paradigma. **Python** usualmente es interpretado (*como en CPython, la implementacion clasica de Python*), sin embargo, existen implementaciones del lenguaje que utilizan compiladores *JIT* (*Como PyPy*), incluso existen implementaciones para ejecutar **Python** en el *JVM* del *Java Platform* (*Como Jython*).

La filosofia de **Python** enfatiza la *legibilidad del codigo*. El uso de la identacion, las construcciones del lenguaje y su capacidad multi-paradigma (*en particular su enfoque a la orientacion a objetos*) apuntan a ayudar a los programadores a escribir codigo claro y logico para proyectos de todas las escalas.

**Python** se ha mantenido en el tope de los lenguajes de programacion mas utilizados durante muchos años. La principal razon de ello es que el lenguaje es bastante sencillo de aprender. Ademias, **Python** cuenta con una de las colecciones mas ricas de librerias, posee librerias para infinidad de dominios, desde *Machine Learning* (con librerias como *TensorFlow*, *PyTorch* y *Keras*) hasta desarrollo Web (con librerias como *Django* y *Flask*). Estas librerias permiten realizar aplicaciones especificas de dominio bajo la simplicidad que provee **Python** como lenguaje, haciendo que el lenguaje se mantenga con una popularidad altisima.

### 1.1. Parte a)

#### 1.1.1. Tipos de datos

- **Tipos unitarios:**

Python posee dos tipos unitarios. El primero es `None`, este se utiliza para representar la ausencia de un valor, este es el valor que retornan los procedimientos del lenguaje. El segundo es `Ellipsis`, este puede utilizar a traves de los `...` (*tres puntos suspensivos*). `Ellipsis` es utilizado como tipo unitario alternativo a `None` cuando *la ausencia de un*

*valor* es un *valor valido*. **Ellipsis** tambien es utilizado en reemplazo al keyword **pass** para representar un bloque vacio.

## ■ Tipos numericos:

Python generaliza los tipos numericos sobre el tipo `numbers.Number`. Cada instancia de number puede ser un `numbers.Integral`, `numbers.Real` o `numbers.Complex`.

- `numbers.Integral` representa el conjunto de los numeros enteros. Dentro de este tipo tenemos `int` que representan a un numero entero en un rango ilimitado (limitado en espacio por la implementacion del lenguaje) y tenemos `bool` que representa los valores logicos `True` y `False`, estos valores se comportan como 1 y 0 respectivamente.
- `numbers.Real`, tambien llamado `float` representa el conjunto de los numeros reales a nivel de precision de la maquina.
- `numbers.Complex` representa el conjunto de los numeros complejos, para ello utiliza un par de numeros `float`, donde el primero representa la parte real y el segundo la parte compleja.

## ■ Secuencias:

Estos representan conjuntos finitos ordenados que son indexados por numeros no negativos. La funcion `len()` retorna el numero de elementos de la secuencia. Cuando se tiene una secuencia de longitud `n`, el conjunto de indices contiene los numeros `0, 1, ..., n - 1`. El elemento `i` de una secuencia `a` se selecciona a traves de `a[i]`.

Las secuencias tambien soportan el denominado *slicing*, esto es, `a[i:j]` selecciona los elementos de la secuencia `a` con indice `k` tal que `i <= k < j`, cuando se utiliza como expresion, el *slice* `a[i:j]` es una secuencia del mismo tipo que `a`, esto implica que en `a[i:j]` los indices comienzan a partir de 0.

Las secuencias tambien soportan *extended slicing* agregando un parametro de *step*, `a[i:j:k]` selecciona los elementos con index `x` donde `x = i + n * k`, `n >= 0` y `i <= x < j`.

Las secuencias ademas, son iterables, esto es, pueden utilizarse como rango de un ciclo `for`.

Las secuencias se distinguen a partir de su mutabilidad:

- **Secuencias inmutables:** Una secuencia inmutable, una vez definida, no puede cambiar el conjunto de elementos que contiene. Dentro de las secuencias inmutables tenemos:
  - **Strings:** El tipo `str` representa una secuencia de elementos que representan un valor *Unicode*. Python no tiene un tipo `char`, en su lugar, Python representa cada caracter como un `String` de longitud 1.

- **Tuplas:** El tipo `tuple` representa una secuencia inmutable de objetos arbitrarios de Python. Tuplas de dos o mas elementos se forman a partir de una expresion separadas por comas, por ejemplo: `1, "Hello World", False, None, 5` es una tupla con 5 elementos. Una tupla de un elemento (*denominada singleton*) se puede construir agregando una coma despues de una expresion, por ejemplo `5,` es la tupla que solo contiene a 5. Una tupla vacia se puede construir usando `()` (*parentesis vacios*).
- **Bytes:** El tipo `bytes` es una secuencia inmutable de bytes de 8-bits representados como enteros en el rango  $0 \leq x < 256$ .
- **Secuencias mutables:** Una secuencia mutable puede cambiar una vez definida. La indexacion y el *slicing* de una secuencia mutable se pueden utilizar como *l-values*. De forma similar pueden usarse como argumentos del operador `del` (que elimina el rango especificado de la secuencia).

Existen dos tipos intrinsecos de secuencias mutables:

- **Listas:** El tipo `list` representa una secuencia mutable de objetos arbitrarios de Python. Las listas son formadas por expresiones separadas por comas dentro de corchetes `[]`. Por ejemplo, `["Hola Mundo", False, (None,), 5]` es una lista de 4 elementos. El tipo `list` posee una variedad de metodos para trabajar con la mutabilidad de la secuencia, de los cuales destaca `list.append` que permite agregar un elemento al final de la lista.
- **Byte arrays:** Equivalente mutable al tipo `bytes`.

Existen mas secuencias mutables dentro de las librerias nativas de Python, por ejemplo, el tipo `array` dentro del modulo `array` y el tipo `collection` dentro de modulo `collections`.

## ■ Tipos conjunto:

Representan conjuntos finitos no ordenados de elementos unicos inmutables. Al no ser ordenados, no se pueden indexar a traves de entero, sin embargo, son iterables. La funcion `len()` retorna el numero de elementos del conjunto.

Python posee dos tipos intrinsecos de conjuntos: El tipo `set` para conjuntos mutables y el tipo `frozenset` para conjuntos inmutables.

- **Tipos mapping:** Representan conjuntos finitos de objetos indexados por valores arbitrarios. La notacion `a[k]` selecciona el elemento indexado por el objeto `k`. La notacion `a[k]` puede utilizarse como *l-value* y como argumento del operador `del`. La funcion `len()` retorna el numero de elementos de un mapping.

Python posee un unico tipo mapping intrinseco:

- **Diccionarios:** Representan conjuntos finitos de objetos indexados por valores *casi arbitrarios*. Los valores que indexan se denominan *keys*, un *key* debe ser de un tipo comparable por *identidad*, esto quiere decir que solo se admiten elementos inmutables como *keys*.

Los diccionarios son mutables, se pueden crear usando la notacion `{ }` (*llaves*). Por ejemplo, `{True: 1, False: None, "DOS": "2"}` es un diccionario.

- **Tipos *Callable*** Tipos sobre los cuales el operador de llamada funcional puede ser aplicado.

Python tiene una amplia gama de tipos *callable*, describimos brevemente los mas relevantes:

- **Funciones definidas por el usuario:** Son funciones definidas usando la palabra reservada `def` o definidas usando la palabra reservada `lambda`.

```
1 def f(x, y):
2     return x + y
3
4 l = lambda x,y: x + y
5
6 print(f(1,1) == l(2,0)) # True
7
```

- **Metodos de instancia:** Funciones definidas por el usuario asociadas a la instancia de una clase. Usualmente, reciben un argumento implicito usualmente llamado `self` el cual es una referencia a la instancia a la cual pertenece.

```
1 class A:
2     ...
3     def m(self,x):
4         return x
5     ...
6
7 a = A()
8 t = a.m(True)
9
```

- **Funciones generadoras:** En otros lenguajes denominados *Iterators*, un *generator* es una funcion o metodo que use la palabra reservada `yield`. Hablaremos mas en detalle de este tipo cuando hablemos de rutinas en Python.
- **Co-rutinas:** Funciones definidas con las palabras reservadas `async def`. Hablaremos mas en detalle de este tipo cuando hablemos de rutinas en Python.
- **Clases:** Las clases de Python son *callable*. Cuando se llama una clase, los argumentos son pasados al metodo `__new__()`, esto generalmente llama `__init__()`, inicializando una nueva instancia de la clase.

Se dice que generalmente se llama a `__init__()` ya que es posible sobrescribir el comportamiento de `__new__()`, esto lo veremos mejor en la seccion de la pregunta dedicada al sistema de tipos.

```
1 class A:
2     def __init__(self):
3         ...
4
5 a = A()
6 print(type(a)) # <class '__main__.A'>
7
```

- **Instancias de clase:** Las instancias de clases pueden ser callable si dentro de su clase se tiene el metodo `__call__()`.

```

1 class A:
2     def __init__(self):
3         ...
4
5     def __call__(self):
6         print("I'm an A instance, I'm being called")
7
8 a = A()
9 a() # "I'm an A instance, I'm being called"
10

```

### ■ Tipos Clase e instancia de clase (*Objects*):

Las clases son la forma de construir nuevos tipos en Python. Una instancia de clase o *Object* representa, tal como su nombre lo indica, a una instancia de una clase en particular.

Hablaremos mas de estos tipos de datos cuando se explique la construccion de nuevos tipos en Python.

### ■ Otros:

Otros tipos que posee python son:

- Tipo `module`. Utilizado para representar modulos.
- Objetos I/O. Utilizado para operaciones de lectura escritura, utilizados, por ejemplo, para leer/escribir en archivos.
- Tipos Internal. Utilizados dentro de la ejecucion dinamica del programa. Aca tenemos a los tipos como `frame`, `code` (*bitcode objects*) y `traceback` usado para el manejo de excepciones.
- Metodos estaticos y metodos de clase. Estos permiten asociar metodos estaticos a un tipo `class`.

### 1.1.2. Sistema de tipos

Los *Objects* son la abstraccion de la informacion para Python. Toda la data en Python es representada a traves de *Objects*, incluso el propio codigo. Cada objeto posee una *identidad*, un *tipo* y un valor (se utiliza a traves de `x`).

La identidad de un objeto nunca cambia una vez se crea el objeto, este valor es unico para cada objeto. El operador `is` compara la identidad de dos objetos. La identidad de un objeto `x` se puede obtener llamando `id(x)`, que retorna la identidad de un objeto en forma de un entero (en CPython `id(x)` retorna la direccion de memoria donde se encuentra `x`).

El *tipo* de un objeto determina las operaciones que el objeto soporta (por ejemplo, "tiene una longitud?") y también define los posibles valores de los objetos de ese tipo. La funcion

`type()` devuelve el tipo de un objeto (que es un objeto en si mismo). Al igual que su identidad, el tipo de un objeto también es inmutable.

El valor de algunos objetos puede cambiar. Los objetos cuyo valor puede cambiar se dicen mutables; los objetos cuyo valor es inalterable una vez creados se llaman inmutables. (El valor de un objeto contenedor inmutable que contiene una referencia a un objeto mutable puede cambiar cuando se modifica el valor de este último; sin embargo, el contenedor sigue considerándose inmutable, porque la colección de objetos que contiene no puede modificarse. Por lo tanto, la inmutabilidad no es estrictamente lo mismo que tener un valor inmutable, es mas sutil). La mutabilidad de un objeto viene determinada por su tipo; por ejemplo, los `numbers`, `str` y los `tuple` son inmutables, mientras que los `dict` y las `list` son mutables.

Los objetos no se destruyen explícitamente, Python es un lenguaje con recolector de basura. La implementación de la recolección de basura depende de la implementación de Python. CPython utiliza recolección de basura por *Reference Counting* con capacidad de detección de *referencias circulares*. El recolector de basura de CPython es accesible a través del módulo `gc`.

Las variables de Python no tiene ninguna restricción de tipo, de forma general podemos decir que las variables de Python hacen referencia a un `object`.

```
1 x = 1 # type(x) == int
2 y = "Hola" # type(y) == string
3
4 z = x
5
6 # asignacion valida, pues x e y son objetos.
7 x = y
8 y = z
```

Una consecuencia de los tipos sean objetos, es que la **equivalencia de tipos** esta dada por el operador `is`:

```
1 int is int # True
2 float is int # False
```

La **compatibilidad de tipos** esta dada por el *Duck typing*. Las restricciones de compatibilidad tipo no son verificadas de forma estática, en su lugar, las operaciones sobre objetos *pueden fallar* significando que el objeto operando no es de tipo compatible. Ejemplos de clases equivalentes a través de *Duck typing*:

- `int` con `float` y `complex`
- `float` con `complex`
- `bytearray` con `bytes`

Python es estricto cuando el *Duck typing* determina que dos tipos son incompatibles bajo una operación, en lugar de intentar producir un resultado que podría ser incoherente (*shoots to my boy JavaScript*), simplemente lanza un `TypeError`. En este sentido, Python es un lenguaje fuertemente tipado, sin embargo, como las variables de Python no poseen

restricciones de tipo, Python se encuentra en un espacio intermedio entre ser un lenguaje fuertemente tipado y ser un lenguaje debilmente tipado.

A tiempo dinamico, Python no posee **inferencia de tipos**, el tipo de una expresion esta dada por el valor resultante (*esto debido al Duck typing*).

En las ultimas version de Python, el lenguaje soporta *Type annotations*. Estas permiten dar *type hints* a las variables, por ejemplo:

```
1 def greeting(name: str) -> str:
2     return 'Hello ' + name
```

Con los *Type annotations*, las implementaciones de Python **podrian** implementar validaciones de restricciones de tipo de forma estatica, asi como tambien **inferencia de tipo** sobre expresiones. Se dice **podrian** porque en la mayoria de implementaciones actuales de Python, los *Type annotations* son ignorados, se utilizan solo a modo de documentacion.

La meta a mediano-largo plazo es agregar a la especificacion del lenguaje un **sistema de tipo gradual** 🐼 que permita validar de forma estatica el tipo de las expresiones utilizando *Type annotations* e inferencia de tipos. Manteniendo el Duck typing parcialmente de forma dinamica.

### 1.1.3. Creacion de nuevos tipos

Python permite la creacion de nuevos tipos utilizando *classes*. Utilizando la plabbra reservada `class` podemos definir nuevos tipos.

```
1 class Person:
2     def __init__(self, name):
3         self.name
4
5     def say_hi(self):
6         print(f'Hi! my name is {self.name}')
```

Una clase define el comportamiento de sus instancias. La clase especifica los atributos y los metodos de sus instancias. Los atributos se pueden ver como **las características** de las instancias y los metodos se pueden ver como **aquello que puede hacer cada instancia**. El concepto de clase viene directamente de la programacion orientada a objetos, lo cual tiene sentido pues la unidad fundamental de informacion en Python son los objetos.

El metodo reservado `__init__` sirve para construir una instancia de la clase. Este metodo es por defecto llamado cuando se utiliza el nombre de la clase como **callable**. `__init__` es unico entre los nombres de metodos de la clase, esto es, solo existe un constructor, no existe sobrecarga de constructores pues python no soporta sobrecarga. Aca un ejemplo de una clase y de como construir instancias de esa clase:

```
1 class Person:
2     def __init__(self, name):
3         self.name
4
```

```

5     def say_hi(self):
6         print(f'Hi! my name is {self.name}')
7
8 joao = Person('Joao Pinto')
9 joao.say_hi() # Hi! my name is Joao Pinto

```

La clase puede especificar atributos en el cuerpo de la clase. Tambien puede especificar atributos dentro de cualquier metodo. Sin embargo, si se intenta acceder a un atributo que no haya sido especificado se lanza un `AttributeError`:

```

1 class C:
2     other_attr = "Hola"
3
4     def __init__(self):
5         ...
6
7     def set_attr(self):
8         self.attr = 'Im set'
9
10
11 a = C()
12 b = C()
13
14 a.set_attr()
15
16 print(a.other_attr) # Hola
17 print(a.attr)      # Im set
18
19 print(b.other_attr) # Hola
20 print(b.attr)      # AttributeError: 'C' object has no attribute 'attr'

```

Los metodos que se especifican dentro de una clase siempre reciben el argumento `self`. Este argumento es una referencia al objeto de la instancia, y puede usarse para acceder a los atributos y nombres de la misma. La razon por la cual se debe recibir el argumento `self` es porque los metodos de instancia son una azucar sintactica:

```

1 class C:
2     other_attr = "Hola"
3
4     def __init__(self):
5         ...
6
7     def set_attr(self):
8         self.attr = 'Im set'
9
10
11 a = C()
12 b = C()
13
14 a.set_attr()
15 # Sugar for: C.set_attr(a)

```

Python permite extender el comportamiento de instancias clases antes diferentes operaciones a traves del *Duck typing*. Python define una lista de nombres de metodo reservados que permiten definir el comportamiento de las instancias, por ejemplo: si defines el metodo



`__call__` las instancias de esa clase seran objetos callable, si defines `__add__` las instancias podran *sumarse* a traves del operador `+`:

```
1 class Box:
2     def __init__(self, objects):
3         self.objects = objects
4
5     def __call__(self, index):
6         return self.objects[index]
7
8     def __add__(self, other):
9         return Box(self.objects + other.objects)
10
11
12 toys = Box(['Insector', 'Dibujador garabatero', 'Bestia trepa rocas'])
13 games = Box(['FFVII', 'Spyro'])
14
15 toys_and_games = toys + games
16 print(toys_and_games(slice(4)))
17
18 # ['Insector', 'Dibujador garabatero', 'Bestia trepa rocas', 'FFVII']
```

Existen multitud me nombres reservados, definiendo los nombres reservados correctos podemos hacer que las instancias de clase soporte variedad de operaciones. Incluso, haciendo las definiciones correctas, podriamos hacer que nuestro fuera equivalente con otros tipos como los `int` o `list`.

Python tambien soporta herencia. Python permite incluso herencia multiple, la prioridad para resolver colisiones de nombres viene dada de izquierda a derecha:

```
1 class A:
2     def who(self):
3         print('A')
4
5     def method_A(self):
6         print('method_A')
7
8
9 class B:
10     def who(self):
11         print('B')
12
13     def method_B(self):
14         print('method_B')
15
16
17 class C(A, B):
18     def method_C(self):
19         print('method_C')
20
21
22 print(issubclass(C, A)) # True
23 print(issubclass(C, B)) # True
24 print(C is A) # False
25 print(C is B) # False
```

```

26
27 k = C()
28 k.who()    # A
29 k.method_A() # method_A
30 k.method_B() # method_B
31 k.method_C() # method_C

```

En Python, no existen modificadores de acceso, es decir, todo es *public*. Sin embargo, los programadores han adoptado una convencion que permite hacer a miembros de la clase *mas dificiles de acceder* haciendo uso de una convencion del interprete. Cuando un miembro de clase es definido con el prefijo `__`, el interprete le asigna el nombre la clase como prefijo. La intencion del interprete con esto es evitar colisiones utilizar herencia.

```

1 class A:
2     def __init__(self):
3         self.__foo = "A"
4
5
6 class B(A):
7     def __init__(self):
8         super().__init__()
9         self.__foo = "B"
10
11
12 b = B()
13 print(b._B__foo) # B
14 print(b._A__foo) # A
15 print(b.__foo)  # AttributeError

```

Esto permite simular atributos y metodos privados. Se dice simular pues sigue existiendo forma de accesarlos. La idea es hacer que los programadores no utilicen miembros que no deberian utilizar desde fuera de la instancia. Es una suerte de recomendacion: *No deberias usar estos miembro, si decides usarlos, es tu reesponsabilidad.*

La versiones mas recientes de Python ofrecen *decorators* para aumentar la funcionalidad de los miembros de una clase. el decorador `property` permite definir atributo a traves de un getter, muy util para definir propiedades de solo lectura:

```

1 class A:
2     def __init__(self):
3         self.__foo = "A"
4
5     @property
6     def foo(self):
7         return self.__foo
8
9
10 a = A()
11 print(a.foo) # A

```

Los decoradores `classmethod` y `staticmethod` permiten crear metodos de clase y metodos estaticos. La diferencia esencial entre ambas, es que los metodos decorados con `classmethod` reciben una referencia a la clase, algo parecido a `self` pero haciendo referencia a la clase en lugar de la instancia. Los `classmethod` son muy utiles para definir constructores

alternativos (una manera de simular la sobrecarga de constructores). Los `staticmethod` son útiles para organizar funciones bajo el espacio de nombres de una clase. Por ejemplo, veamos un segmento de la clase `Arbol` utilizada en la parte de B de la pregunta 1:

```
1 class Arbol:
2     # Representacion interna de Arbol.
3     # Esto para restringir las formas de construir un Arbol a solo Rama y
4     # Hoja.
5     class __Representation:
6         def __init__(self, value, left=None, right=None):
7             self.__value = value
8             self.__left = left
9             self.__right = right
10
11         # Definimos getters sobre los atributos del arbol.
12
13         @property
14         def value(self):
15             return self.__value
16
17         @property
18         def left(self):
19             return self.__left
20
21         @property
22         def right(self):
23             return self.__right
24
25         # No definimos setters para asegurar que la estructura sea
26         # inmutable desde fuera de la representacion.
27
28         # Constructor Hoja a
29         @classmethod
30         def Hoja(cls, value):
31             return cls.__Representation(value)
32
33         # Constructor Rama a (Arbol a) (Arbol a)
34         @classmethod
35         def Rama(cls, value, left, right):
36             return cls.__Representation(value, left, right)
37
38         # Funcion esDeBusqueda
39         @staticmethod
40         def is_BST(root):
41             ...
```

Esto permite luego utilizar la clase `Arbol` de la siguiente manera:

```

1
2 arbol = Arbol.Rama(
3     6,
4     Arbol.Rama(
5         4,
6         Arbol.Rama(
7             2,
8             Arbol.Hoja(1),
9             Arbol.Hoja(3)
10        ),
11        Arbol.Hoja(5)
12    ),
13    Arbol.Rama(
14        9,
15        Arbol.Rama(
16            8,
17            Arbol.Hoja(7),
18            Arbol.Hoja(9)
19        ),
20        Arbol.Rama(
21            10,
22            Arbol.Hoja(9),
23            Arbol.Hoja(11)
24        )
25    ),
26 )
27
28 print(Arbol.is_BST(arbol))

```

### 1.1.3.1. Rutinas

- **Funciones:** El tipo de rutina mas simple en Python. Una funcion se define usando la palabra reservada **def**, indicando el nombre de la funcion, la lista de argumentos y el cuerpo de la funcion. En la lista de argumentos puedes tener de cero a tantos argumentos como se deseen, tambien se puede asignar valores por defectos para los argumentos. Los argumentos que tengan valor por defecto deben estar al final de la lista de argumentos. El cuerpo de la funcion puede opcionalmente contener una instruccion de **return** que especifica el valor de retorno de la funcion. Si una funcion no tiene una instruccion de **return**, la funcion siempre retorna el tipo unitario **None**. Ejemplos de funciones:

```

1 # Funcion vacia y sin argumentos
2 # Ambas retornan None y no ejecutan nada
3 def f():
4     ...
5
6 def g():
7     pass
8
9 # Funcion con uno o varios arguments

```

```

10 def s1(x)
11     return x
12
13 def s2(x, y, z)
14     return x + y + z
15
16 # Procedimiento (funcion que retorna None)
17 def p(x, y):
18     print(x, y)
19
20 # Funcion con valores por defecto
21 def h(a, b, c=False, d=True)
22     return a and b and c and d
23
24

```

- **Funciones lambda:** Tambien llamadas funciones anonimas. Estas permiten definir funciones de forma anonima utilizando la notacion de *Lambda calculo*. Estas funciones equivalentes en tipo con las funciones convencionales. Las funciones lambda retornan la expresion especificada despues de los dos puntos. Solo se puede tener una expresion dentro de una funcion lambda. Ejemplo de funciones lambda:

```

1 suma = lambda x,y: x + y
2
3 suma_curry = lambda x: lambda y: x + y
4

```

Estas funciones son ampliamente usadas en el aspecto funcional de Python, en expresiones que involucran `map`, `reduce`, `zip`. En general se utilizan mucho en la manipulacion y uso de funciones de orden superior.

- **Decorators:** Los *decorators* son funciones de orden superior que reciben una funcion y algunos argumentos. Los *decorators* permiten *decorar* o *aumentar* el comportamiento de una funcion. Se definen como funciones convencionales o como funciones lambda. Los decorators son tipo equivalente con las funciones convencionales. El uso de decorators es un azucar sintactica a traves del prefijo `@`:

```

1 import math
2
3 # Decorador
4 def validator(condition):
5     return lambda f: lambda x: f(x) if condition(x) else None
6
7 # Azucar sintactica usando @
8 @validator(lambda x: x != 0)
9 def inverse(x):
10     return 1 / x
11
12 # los decoradores se pueden componer
13 @validator(lambda x: x != 0)
14 @validator(lambda x: x > 0)
15 def inverse_sqrt(x):

```

```
16     return 1 / math.sqrt(x)
17
```

Los *decorators* se pueden implementar con cualquier *callable*, sin embargo me parecio interesenante mencionarlos brevemente al hablar de funciones.

- **Funciones generadoras:** En otros lenguajes denominados *Iterators*, un *generator* es una funcion o metodo que use la palabra reservada `yield`. Una función de este tipo, cuando se llama, siempre devuelve un objeto de tipo *iterator* que puede utilizarse para ejecutar el cuerpo de la funcion: llamar al metodo `iterator.__next__()` del iterador hará que la función se ejecute hasta que proporcione un valor utilizando `yield`. La proxima vez que se llame el metodo `iterator.__next__()`, la ejecucion funcion se resume al punto inmediatamente despues de el ultimo `yield` que genero un valor. Si la ejecucion de la funcion llega a un `return` o al final de su bloque antes de llegar a un `yield`, la proxima vez que se llame `iterator.__next__()`, la ejecucion comenzara desde el inicio.

Los *generators* son considerados **co-rutinas**, pues su estado se mantiene independientemente de su llamador.

Naturalmente, los *generator* se utilizan para generar secuencias de valores, estas secuencias pueden ser infinitas, por ejemplo, el siguiente *generator* genera la secuencia de los numeros fibbonacci:

```
1 def fib_generator():
2     l = 0
3     c = 1
4     while True:
5         yield l
6         n = l + c
7         l = c
8         c = n
9
10 for x in fib_generator():
11     print(x)
12
13 # 0, 1, 1, 2, 3, 5, 8, 13, ...
14
```

- **Co-rutinas:** Funciones definidas con las palabras reservadas `async def`. Estas funciones trabajan en conjuncion con la libreria nativa `asyncio` para proveer de un manejo de co-rutinas via Async IO, muy similar al funcionamiento de las promesas de *JavaScript*.

Dentro del cuerpo de una co-rutina A se puede utilizar la palabra reservada `await`, esta palabra cede el control de la ejecucion a otra co-rutina B, cuando la co-rutina invocada (B) realiza un `return`, llega al final de su bloque o realiza `yield`, el control

de la ejecucion vuelve a la co-rutina A.

```
1 async def f(x):  
2     y = await z(x)  
3     return y  
4  
5 async def g(x):  
6     yield x  
7
```

Note que del parrafo anterior se infiere que se pueden definir co-rutinas que sean *generators*, de hecho, las co-rutinas retornan un valor de tipo **coroutine** que casualmente es una especializacion de **iterator**, el tipo que retornan los *generators*.

Python provee un control de flujo especial para *generators* definidos como **async def**. Este es el **async for**, que permite iterar sobre la secuencia generada por un *generator* **async def**. Este control de puede usar fuera de funciones **async**, pues trata a la co-rutina como si fuera netamente un *generator*:

```
1 async def get_docs():  
2     page = await fetch_page()  
3     while page:  
4         for doc in page:  
5             yield doc  
6         page = await fetch_page()  
7  
8 async for doc in get_docs():  
9     pass # work on doc  
10
```

Las limitaciones de **await**, es que solo se puede usar dentro del cuerpo de una co-rutina y que solo puede operar sobre objetos del tipo **coroutine**.

**asyncio** provee herramientas para ejecutar funciones **async def**, la principal es **asyncio.run** que ejecuta una co-rutina utilizando un *Event loop*. Usando **asyncio.run**, **await** agrega la co-rutina a invocar a la cola de *Eventos* y cede el control de la ejecucion al **Event loop**. Posteriormente el **Event loop** se encarga de dar control a la co-rutina que este en el tope de la cola de *Eventos*. Cuando el **Event loop** otorga control a una co-rutina y la co-rutina finaliza su ejecucion, el **Event loop** agrega a la cola de *Eventos* al invocador de la co-rutina finalizada (junto con los valores que esperaba de la co-rutina invocada), de forma que se cumpla la propiedad descrita en el parrafo anterior.

```
1 import asyncio  
2  
3 async def count():  
4     print("One")  
5     await asyncio.sleep(1)  
6     print("Two")  
7  
8 async def main():
```

```

9     await asyncio.gather(count(), count(), count())
10
11 asyncio.run(main())
12
13 # One
14 # One
15 # One
16 # Two
17 # Two
18 # Two
19

```

### 1.1.3.2. Pasaje de parametros

El pasaje de parametros en Python es por *call-by-sharing*. Los parametros son pasados como referencia indirectas a los objetos que representan, esto permite la lectura de los argumentos con total libertad, sin embargo, al asignar un valor a un argumento, al argumento se le modifica el valor de referencia indirecta de forma que apunte al objeto que se le esta asignando, de esta forma, no se modifica el argumento original. El pasaje de parametros de Python en ocasiones es denominado *por asignacion*. Aca tenemos un ejemplo del pasaje de parametro de Python:

```

1 def main():
2     n = 9001
3     print(f"Initial address of n: {id(n)}")
4     increment(n)
5     print(f"    Final address of n: {id(n)}")
6
7 def increment(x):
8     print(f"Initial address of x: {id(x)}")
9     x += 1
10    print(f"    Final address of x: {id(x)}")
11
12 main()
13 # Initial address of n: 140562586057840
14 # Initial address of x: 140562586057840
15 #    Final address of x: 140562586057968
16 #    Final address of n: 140562586057840

```

### 1.1.4. Manejo de excepciones

Python provee la clase `Exception`. Cuando durante la ejecucion de ocurre una excepcion, Python *lanza* una excepcion:

```

1 def divide_by_zero():
2     return 1 / 0
3
4 divide_by_zero()
5 # Traceback (most recent call last):
6 #   File "<stdin>", line 1, in <module>
7 #   File "<stdin>", line 2, in divide_by_zero

```



```
8 # ZeroDivisionError: division by zero
```

En este caso, Python lanza `ZeroDivisionError` que es sub-clase de `Exception`.

Como programadores también podemos *lanzar* excepciones usando la palabra reservada `raise`:

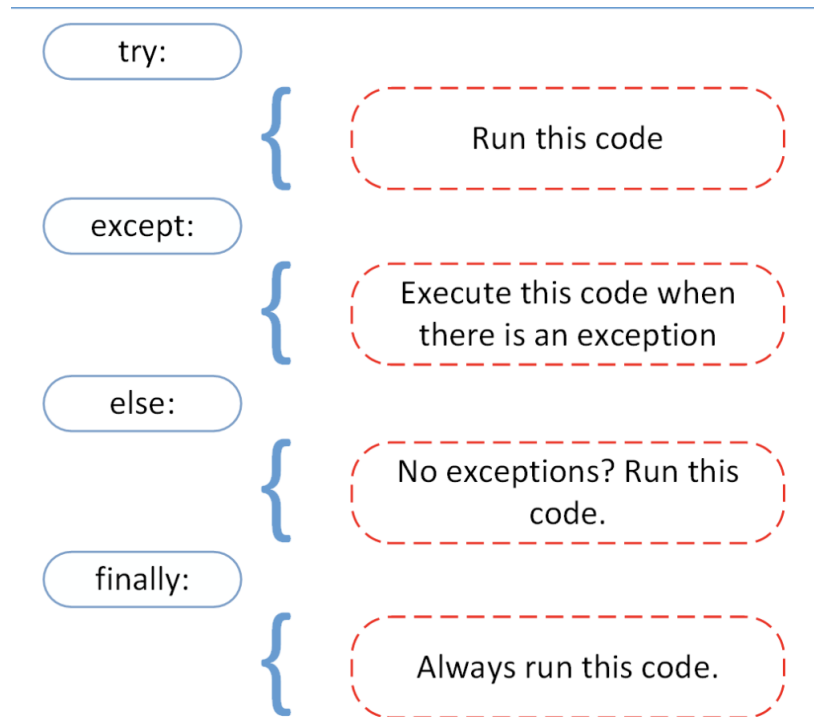
```
1 x = 10
2 if x > 5:
3     raise Exception('x should not exceed 5. The value of x was: {}'.format
4         (x))
5 # Traceback (most recent call last):
6 #   File "<input>", line 4, in <module>
7 # Exception: x should not exceed 5. The value of x was: 10
```

El operador `raise` recibe una instancia de la clase `Exception`. Muchas veces conviene crear sub-tipos para nuestras excepciones específicas.

También podemos *capturar* excepciones usando el bloque `try`, `except`, `else`, `finally`:


```
1 try:
2     linux_interaction()
3 except AssertionError as error:
4     print(error)
5 else:
6     try:
7         with open('file.log') as file:
8             read_data = file.read()
9     except FileNotFoundError as fnf_error:
10        print(fnf_error)
11 finally:
12    print('Cleaning up, irrespective of any exceptions.')
```

El flujo de este bloque sigue la siguiente semántica:




## 1.2. Parte b


### 1.2.1. Tipo arbol:

En el archivo *arbol.py*  se encuentra dicha la implementacion del tipo solicitado.

### 1.2.2. Tipo Church:

En el archivo *church.py*  se encuentra dicha la implementacion del tipo solicitado. Para implementar este tipo, nos basamos en la definicion de los numerales de Church dada por el lambda calculo. Adicional a lo especificado en el enunciado, agregamos el constructor `from_int` que construye un numeral de Church a partir de su equivalente `int`.

## 2. Pregunta 2

En la carpeta *pregunta-2*  se encuentra un archivo *.pdf* para cada parte de esta pregunta.

## 3. Pregunta 3

Respuesta no entregada.

## 4. Pregunta 4

La respuesta a esta pregunta se encuentra en una presentacion en formato `.pdf`. En el archivo ***pregunta-4.pdf*** se encuentra dicha presentacion.

## 5. Pregunta 5

En esta pregunta, se logro implementar el parser de tipos y el algoritmo de unificacion. En el archivo ***type.py*** se encuentra la implementacion de parse usando ***Lark*** y la implementacion del algoritmo de unificacion. En ese mismo archivos estan los casos de prueba que utilice para comprobar los resultados.

Intente implementar el parser de expresiones con Lark, pero la asociatividad generaba expresiones ambiguas, el resultado era no determinista. Despues de horas de jugar con Lark sin lograr respetar la asociatividad de la aplicacion funcional, decidi probar otros parsers, probe con ***Arpeggio*** pero no logre hacer que el parser retornara la representacion correcta. En el archivo ***expression.py*** se encuentra la representacion que se le dio a las expresiones y el parser de Lark (ambiguo, pero esta, si se usan parentesis redundantes, funciona).

Tambien se trabajo en la clase `TypeSystem`. En el archivo ***system.py*** se encuentra el desarrollo alcanzado. No continue con el desarrollo de esta parte porque me di cuenta de que el parser de expresiones era ambiguo.

En el archivo ***parser-tries.py*** se encuentra el ultimo los intento de parser usando Arpeggio. Lo dejo por si sirve de algo.

Voy a continuar trabajando en la rama `after\_parcial`. De forma que si usted desea ver mi implementacion a momento de la entrega pueda hacerlo entrando en los links que deje en este documento (directos a la rama main).

Lo invito a revisar el ***README.md*** de la version `after\_parcial` para que pueda apreciar el resultado final de este problema. Entenderia completamente si la correccion fuera sobre lo entregado al momento de enviar este documento. Gracias.