1) Python es un lenguaje de programación de alto nivel, caracterizado por su fácil lectura y sintaxis clara. Es ampliamente utilizado para desarrollo web, análisis de datos, inteligencia artificial, y automatización de tareas.

   a) -En Python, los objetos se crean utilizando la palabra clave class. Por ejemplo, la siguiente declaración crea una clase llamada Persona:

```
class Persona:
    …
```

   - Para crear una instancia de una clase, basta con llamar a la clase seguido de una parentización junto a los parámetros que este necesite para su constructor:

```
persona = Persona(args)
```

   - Una instancia de una clase puede acceder a sus atributos y métodos utilizando el operador de acceso “.” Por ejemplo, la siguiente declaración imprime el nombre de la persona, siempre que este definido en sus atributos:

```
persona.nombre = "Juan Pérez"
print(persona.nombre)
```

   - En Python, los constructores se definen utilizando el método __init__(). Por ejemplo, la siguiente declaración define un constructor para la clase Persona:

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
```

   - Los métodos son funciones que se definen dentro de una clase. Para acceder a un método de una instancia de una clase, se utiliza el operador de acceso “.” seguido del nombre del método. Por ejemplo, la siguiente declaración imprime el nombre de la persona:

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def get_nombre(self):
        return self.nombre
```

```
persona = Persona("Juan Pérez")
print(persona.get_nombre())
```

b) El recolector de basura (GC [Garbage Collector]) de Python es un proceso que se encarga de liberar la memoria que ya no se utiliza y este funciona de manera automática, de la siguiente manera:

- El GC mantiene un registro de todos los objetos que existen en la memoria.
- El GC rastrea las referencias a cada objeto.
- Si un objeto no tiene ninguna referencia, se considera que está "inalcanzable".
- El GC libera la memoria de los objetos inalcanzables.

El GC utiliza un algoritmo cíclico generacional para clasificar los objetos en tres generaciones:

- Generación 0: Objetos recién creados.
- Generación 1: Objetos que han sobrevivido a una recolección de la generación 0.
- Generación 2: Objetos que han sobrevivido a dos recolectas de la generación 0.
-

El GC recolecta los objetos de la generación 0 con mayor frecuencia, ya que es más probable que sean inalcanzables. Los objetos de la generación 1 y 2 se recolectan con menos frecuencia.

El GC puede ser desactivado por el programador utilizando la función gc.disable().

c) Python utiliza asociación dinámica de métodos. Esto significa que el método que se invoca para una instancia de una clase se determina en tiempo de ejecución, en función del tipo de la instancia.

Existe una manera de alterar la elección por defecto del lenguaje de asociación de métodos, utilizando el decorador @staticmethod para crear un método de tipo estático:

```
@staticmethod
        def hw(self):
                print("Hola Mundo")

hw_estatico = types.MethodType(hw, None)
```

d) - En Python, los tipos básicos son int, float, str, list, dict, y tuple. Los tipos compuestos son clases y tipos enumerados.

   - Python no admite herencia múltiple. Esto significa que una clase sólo puede heredar de una clase base.

   - Python admite polimorfismo paramétrico utilizando clases abstractas. Una clase abstracta es una clase que no puede instanciarse. Las clases abstractas se utilizan para definir interfaces que las clases derivadas deben implementar.

   - Python admite el manejo de varianzas utilizando la anotación de tipos:

```
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre
    def hablar(self):
        print("hola")

class Perro(Animal):
    def ladrar(self):
        print("guau!")
```

2) - Haskell ofrece capacidades nativas para concurrencia, a través de la biblioteca estándar Control.Concurrent. Esta biblioteca proporciona una serie de primitivas para la creación, manejo y sincronización de procesos concurrentes.

a) Para crear una tarea concurrente en Haskell, se utiliza la primitiva forkIO. Esta primitiva toma como argumento una función IO, que se ejecutará en un nuevo proceso.

b) Para manejar las tareas concurrentes, se puede utilizar la primitiva join. Esta primitiva espera a que un proceso termine su ejecución y devuelve el valor resultante.

Ejemplo:

```
import Control.Concurrent
main = do
  t1 <- forkIO $ do
    print "Tarea 1"
  t2 <- forkIO $ do
    print "Tarea 2"
  join t1
  join t2
```

Haskell no proporciona soporte nativo para la memoria compartida. Sin embargo, se puede utilizar la biblioteca estándar Control.Concurrent.MVar para implementar la memoria compartida sincronizada.

Haskell proporciona soporte nativo para el pasaje de mensajes a través de canales. Los canales se pueden crear utilizando la primitiva newChan.

c) Los mecanismos de sincronización más comunes en Haskell son los siguientes:

- MVar: Variables mutables sincronizadas. Las MVar son variables mutables que se pueden utilizar para proteger el acceso a datos compartidos. Para acceder a una MVar, se debe primero adquirir el bloqueo de la MVar. Una vez que se tiene el bloqueo, se puede modificar el valor de la MVar. Cuando se termina de modificar el valor de la MVar, se debe liberar el bloqueo.

- Semaphore: Los semáforos se utilizan para controlar el acceso a recursos compartidos. Un semáforo tiene un valor entero que representa el número de recursos disponibles. Cuando un proceso necesita acceder a un recurso, debe esperar hasta que el valor del semáforo sea mayor que cero. Cuando el proceso termina de utilizar el recurso, debe liberar el recurso decrementando el valor del semáforo.

- Mutex: Los mutexes son mecanismos de exclusión mutua. Un mutex solo puede ser adquirido por un proceso a la vez. Cuando un proceso necesita adquirir un mutex, debe esperar hasta que el mutex esté disponible. Cuando el proceso termina de utilizar el mutex, debe liberarlo.

- Variable de condición: Las variables de condición se utilizan para sincronizar procesos que esperan a que se cumpla una condición. Un proceso puede esperar a que se cumpla una condición bloqueándose en una variable de condición. Cuando la condición se cumple, otro proceso puede despertar al proceso que está esperando.

3) (a) Asociación Estática de Métodos

| X = 1 | Y = 2 | Z = 6 |
|---|---|---|

```
class Abra {
   int a = X, b = Y

   fun cus(int x): int {
      a = b + x
      return pide(a)
   }

   fun pide(int y): int {
      return a - y * b
   }
}


class Cadabra extends Abra {
   Abra zo = new PataDeCabra()

   fun pide(int y): int {
      return zo.cus(a + b) - y
   }
}


class PataDeCabra extends Cadabra {
   int b = Y + Z, c = Z

   fun cus(int x): int {
      a = x - 3
      c = a + b * c
      return pide(a * b + x)
   }

   fun pide(int y): int {
      return c - y * a
}
```

Asociacion Estática de Métodos

```
Abra ho = new Cadabra()
Abra po = new PataDeCabra()
Cadabra cir = new PataDeCabra()

print(ho.cus(X + 1) + po.cus(Y + 1) + cir.cus(Z + 1))
```

| Objeto | Metodos | Variables |
|---|---|---|
| ho (Cadabra) | pide = Abra::pide() | zo (PataDeCabra()) |
| | cus = Abra::cus() | a = 1 |
| | | b = 2 |
| ho.zo (PataDeCabra) | cus = Abra::cus() | a = 1 |
| | pide = Abra::pide() | b = 2 |
| | | c = 6 |
| po (PataDeCabra) | cus = Abra::cus() | a = 1 |
| | pide = Abra::pide() | b = 2 |
| | | c = 6 |
| cir (PataDeCabra) | cus = Abra::cus() | a = 1 |
| | pide = Cadabra::pide() | b = 2 |
| | | c = 6 |

**PILA**

| | | | |
|---|---|---|---|
| 19 | | | |
| 18 | | | |
| 17 | | | |
| 16 | | | |
| 15 | | | |
| 14 | | | |
| 13 | | | |
| 12 | | | |
| 11 | | | |
| 10 | | | |
| 9 | | | |
| 8 | | | |
| 7 | | | |
| 6 | | | |
| 5 | | | |
| 4 | out = 4 - 4 * 2 | out = 5 - 5 * 2 | out = 9 - 9 * 2 |
| 3 | y = a | y = 5 | y = 9 |
| 2 | pide(a) | pide(a) | pide(a) |
| 1 | a = 4 | a = 5 | a = 9 |
| 0 | x = 2 | x = 3 | x = 7 |
| | ho.cus(X + 1) = -4 | po.cus(Y + 1) = -5 | cir.cus(Z + 1) = -9 |

**PRINT = -18**

(a) Asociación Dinámica de Métodos

| X = 1 | Y = 2 | Z = 6 |
|---|---|---|

```
class Abra {
   int a = X, b = Y

   fun cus(int x): int {
      a = b + x
      return pide(a)
   }

   fun pide(int y): int {
      return a - y * b
   }
}


class Cadabra extends Abra {
   Abra zo = new PataDeCabra()

   fun pide(int y): int {
      return zo.cus(a + b) - y
   }
}


class PataDeCabra extends Cadabra {
   int b = Y + Z, c = Z

   fun cus(int x): int {
      a = x - 3
      c = a + b * c
      return pide(a * b + x)
   }

   fun pide(int y): int {
      return c - y * a
}
```

| Asociacion Dinámica de Métodos |
|---|

```
Abra ho = new Cadabra()
Abra po = new PataDeCabra()
Cadabra cir = new PataDeCabra()

print(ho.cus(X + 1) + po.cus(Y + 1) + cir.cus(Z + 1))
```

| Objeto | Metodos | Variables |
|---|---|---|
| ho (Cadabra) | pide = Cadabra::pide() | zo (Abra()) |
| | cus = Abra::cus() | a = 1 |
| | | b = 2 |
| ho.zo (PataDeCabra) | cus = Abra::cus() | a = 1 |
| | pide = Abra::pide() | b = 2 |
| po (PataDeCabra) | cus = PataDeCabra::cus() | a = 1 |
| | pide = PataDeCabra::pide() | b = 8 |
| | | c = 6 |
| cir (PataDeCabra) | cus = PataDeCabra::cus() | a = 1 |
| | pide = PataDeCabra::pide() | b = 8 |
| | | c = 6 |

**PILA**

| | | | |
|---|---|---|---|
| 19 | | | |
| 18 | | | |
| 17 | | | |
| 16 | | | |
| 15 | | | |
| 14 | | | |
| 13 | | | |
| 12 | | | |
| 11 | | | |
| 10 | | | |
| 9 | out = -8 -4 | | |
| 8 | y = 8 | | |
| 7 | pide(a) | | |
| 6 | a = 8 | | |
| 5 | x = 6 | out = 48 | out = -104 |
| 4 | ho.zo.cus(a + b) | y = 3 | y = 39 |
| 3 | y = 4 | pide(a * b + x) | pide(a * b + x) |
| 2 | pide(a) | c = 48 | c = 52 |
| 1 | a = 4 | a = 0 | a = 4 |
| 0 | x = 2 | x = 3 | x = 7 |
| | ho.cus(X + 1) = -12 | po.cus(Y + 1) = 48 | cir.cus(Z + 1) = -104 |

**PRINT = -68**

5) a)
(i) orden normal

- misteriosa "abc" (gen 1)
- foldr what (const []) "abc" (gen 1)
- what "a" $ foldr what (const []) "bc" (gen 1)
- what "a" $ foldr what (const []) "bc" (1 : gen 2)
- ("a", 1) : $ foldr what (const []) "bc" (gen 2)
- ("a", 1) : what "b" $ foldr what (const []) "c" (gen 2)
- ("a", 1) : what "b" $ foldr what (const []) "c" (2 : gen 3)
- ("a", 1) : ("b", 2) : $ foldr what (const []) "c" (gen 3)
- ("a", 1) : ("b", 2) : what "c" $ foldr what (const []) "" (gen 3)
- ("a", 1) : ("b", 2) : what "c" $ foldr what (const []) "" (3 : gen 4)
- ("a", 1) : ("b", 2) : ("c", 3) $ foldr what (const []) "" (gen 4)
- ("a", 1) : ("b", 2) : ("c", 3) : (const []) (gen 4)
- ("a", 1) : ("b", 2) : ("c", 3) : []
- ("a", 1) : ("b", 2) : ("c", 3) : []
- ("a", 1) : ("b", 2) : [("c", 3)]
- ("a", 1) : [("b", 2), ("c", 3)]
- [("a", 1), ("b", 2), ("c", 3)]

## Resultado Final:

[("a", 1), ("b", 2), ("c", 3)]

(ii) orden aplicativo

- misteriosa "abc" (gen 1)
- misteriosa "abc" (1 : gen 2)
- misteriosa "abc" (1 : 2 : gen 3)
- misteriosa "abc" (1 : 2 : 3 : gen 4)
- misteriosa "abc" (1 : 2 : 3 : 4 : gen 5)
- misteriosa "abc" (1 : 2 : 3 : 4 : 5 : gen 6)

## Recursión infinita de Gen, se detiene la corrida.

b)

```
data Arbol a = Hoja | Rama a (Arbol a) (Arbol a)
foldA :: (a -> b -> b -> b) -> b -> Arbol a -> b
foldA f b Hoja = b
foldA f b (Rama value left right) = Rama (f value) (foldA f b left) (foldA f b right)
```

c) (i)Orden de evaluación Normal:

- sospechosa arbolito (genA 1)
- foldA whatTF (const Hoja) arbolito (genA 1)

- foldA whatTF (const Hoja) (Rama 'a'
  (Rama 'b'
  Hoja
  (Rama 'c' Hoja Hoja)
  )
  Hoja
  ) (genA 1)
- whatTF 'a' (foldA whatTF (const Hoja) (Rama 'b'
  Hoja
  (Rama 'c'
  Hoja
  Hoja
  )
  )
  )
  (foldA whatTF (const Hoja) Hoja)
  (genA 1)

## Evaluamos genA 1 - x

- whatTF 'a' (foldA whatTF (const Hoja) (Rama 'b'
  Hoja
  (Rama 'c' Hoja Hoja)
  )
  )
  (foldA whatTF (const Hoja) Hoja)
  (Rama 1
  (genA (1 + 1))
  (genA (1 * 2))
  )

## Evaluamos whatTF 'a' f g

- Rama ('a', 1)
  ((foldA whatTF (const Hoja) (Rama 'b'
  Hoja
  (Rama 'c' Hoja Hoja)
  )
  )
  (genA (1 + 1))
  )
  ((foldA whatTF (const Hoja) Hoja)
  (genA (1 * 2))
  )

## Evaluamos el primer foldA

- Rama ('a', 1)

(whatTF 'b' (foldA whatTF (const Hoja) Hoja) (foldA whatTF (const Hoja) (Rama 'c' Hoja Hoja)) (genA (1 + 1)))
((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos (genA (1+1))

- (Rama y
  Rama ('a', 1)
  (whatTF 'b' (foldA whatTF (const Hoja) Hoja) (foldA whatTF (const Hoja) (Rama 'c' Hoja Hoja)) (Rama 2
  (genA (2 + 1))
  (genA (2 * 2))
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos whatTF 'b'

Rama ('a', 1)
(Rama ('b', 2)
((foldA whatTF (const Hoja) Hoja) (genA (2 + 1)))
(foldA whatTF (const Hoja) (Rama 'c' Hoja Hoja) (genA (2 * 2)))
)
((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos el primer foldA

- Rama ('a', 1)
  (Rama ('b', 2)
  ((const Hoja) (genA (2 + 1)))
  (foldA whatTF (const Hoja) (Rama 'c' Hoja Hoja) (genA (2 * 2)))
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos (const Hoja) (genA (2 + 1))

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (foldA whatTF (const Hoja) (Rama 'c' Hoja Hoja) (genA (2 * 2)))
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos el primer foldA

- Rama ('a', 1)
  (Rama ('b', 2)

Hoja
(whatTF 'c' (foldA whatTF (const Hoja) Hoja)(foldA whatTF (const Hoja) Hoja) (genA (2 * 2)))
)
((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos (genA (2 * 2))

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (whatTF 'c' (foldA whatTF (const Hoja) Hoja) (foldA whatTF (const Hoja) Hoja) (Rama 4
  (genA(4+1))
  (genA(4*2))
  )
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos whatTF 'c'

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  ((foldA whatTF (const Hoja) Hoja) (genA(4+1)))
  ((foldA whatTF (const Hoja) Hoja) (genA(4*2)))
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos el primer foldA

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  ((const Hoja) (genA(4+1)))
  ((foldA whatTF (const Hoja) Hoja) (genA(4*2)))
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos (const Hoja) (genA(4+1))

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  Hoja
  ((foldA whatTF (const Hoja) Hoja) (genA(4*2)))
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos el primer foldA

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  Hoja
  ((const Hoja) (genA(4*2))
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos (const Hoja) (genA(4*2))

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  Hoja
  Hoja
  )
  )
  ((foldA whatTF (const Hoja) Hoja) (genA (1 * 2)))

## Evaluamos foldA de más arriba

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)

Hoja
Hoja
)
)
((const Hoja) (genA (1 * 2)))

## Evaluamos (const Hoja) (genA (1 * 2))

- Rama ('a', 1)
  (Rama ('b', 2)
  Hoja
  (
  Rama ('c', 4)
  Hoja
  Hoja
  )
  )
  Hoja

## Resultado:

Rama ('a', 1) (Rama ('b', 2) Hoja (Rama ('c', 4) Hoja Hoja)) Hoja

(ii) Orden de evaluación Aplicativo:

- sospechosa arbolito (genA 1)
- sospechosa arbolito Rama 1
  (genA (2))
  (genA (2))
- sospechosa arbolito Rama 1
  (Rama 2 (genA (3)) (genA (4)))
  (genA (2))
- sospechosa arbolito Rama 1
  (Rama 2 (genA (3)) (genA (4)))
  (Rama 2 (genA (3)) (genA (4)))

- sospechosa arbolito Rama 1
  (Rama 2
  (genA (3))
  (genA (4))
  )
  (Rama 2
  (genA (3))
  (genA (4))
  )

- sospechosa arbolito Rama 1

```
(Rama 2
    (Rama 3 (genA (4)) (genA (6)))
    (genA (4))
)
(Rama 2
    (genA (3))
    (genA (4))
)
```

- sospechosa arbolito Rama 1

```
(Rama 2
    (Rama 3 (genA (4)) (genA (6)))
    (Rama 3 (genA (4)) (genA (6)))
)
(Rama 2
    (genA (3))
    (genA (4))
)
```

**Evaluación recursiva infinita por genA, se detiene la corrida.**