Desarrollo de software para blockchain 2024

Clase 02 - Python

Temario

0 - Instalación y definiciones

1. Sintaxis básica

2. Estructuras de control

3. Funciones y módulos

4. Conceptos básicos de programación orientada a objetos (POO)

0- Instalación y definiciones

- Python es un lenguaje de programación de alto nivel, interpretado y de propósito general que se caracteriza por su sintaxis sencilla y legible, lo que facilita su aprendizaje y uso. Fue creado por Guido van Rossum y lanzado por primera vez en 1991. Python es muy versátil y se utiliza en una amplia variedad de aplicaciones, como desarrollo web, análisis de datos, inteligencia artificial, automatización de tareas y más. Su amplia comunidad y una gran cantidad de bibliotecas disponibles hacen que sea uno de los lenguajes de programación más populares y poderosos en la actualidad.

0- Instalación y definiciones

Visita la página oficial de Python: https://www.python.org/downloads/

Instalación en Windows:

1. Ejecuta el archivo descargado (.exe).

2. Antes de hacer clic en "Install Now", asegurarse de marcar la casilla que dice "Add Python to PATH". Esto es importante para poder ejecutar Python desde la línea de comandos.

3. Haz clic en "Install Now" y espera a que termine el proceso de instalación.

0- Instalación y definiciones : pip

¿Qué es pip?

pip es el administrador de paquetes de Python que te permite instalar y gestionar bibliotecas y paquetes adicionales que no están incluidos por defecto en Python.

Instalación de pip en Windows:

- Descarga el archivo get-pip.py desde https://bootstrap.pypa.io/get-pip.py.
- Abre la terminal (o el Símbolo del sistema) y navega a la carpeta donde se descargó el archivo.
- Ejecuta el siguiente comando:
    - python get-pip.py

1. Sintaxis básica

- Variables y tipos de datos (números, cadenas, listas, tuplas, diccionarios).
- Operadores (aritméticos, de comparación, lógicos).
- Comentarios y estilo de código.

1. Sintaxis básica

```
# Declaración de variables
string = "Ana"
entero = 25
flotante = 1.68
booleano = True
lista = ["Matemáticas", "Historia", "Ciencias"]
tupla = 1,2,"Hola!"
```

1. Sintaxis básica

```
# Operadores aritméticos
suma = 5 + 3  # Resultado: 8
resta = 10 - 4  # Resultado: 6

# Operadores de comparación
es_mayor = edad > 18  # Resultado: True
```

1. Sintaxis básica

```
# Operaciones con listas
# Crear una lista con diferentes tipos de elementos
mi_lista = [1, 2, 3, "Python", 5.5, True]
# Acceder al primer elemento (índice 0)
print(mi_lista[0])  # Salida: 1

# Acceder al último elemento
print(mi_lista[-1])  # Salida: True

# Cambiar el segundo elemento (índice 1)
mi_lista[1] = "nuevo valor"
print(mi_lista)  # Salida: [1, "nuevo valor", 3, "Python", 5.5, True]
```

1. Sintaxis básica

```
# Operaciones con listas
# Añadir un elemento al final de la lista
mi_lista.append("Nuevo elemento")
print(mi_lista)  # Salida: [1, "nuevo valor", 3, "Python", 5.5, True, "Nuevo elemento"]

# Insertar un elemento en una posición específica
mi_lista.insert(2, "Elemento insertado")
print(mi_lista)  # Salida: [1, "nuevo valor", "Elemento insertado", 3, "Python", 5.5, True, "Nuevo elemento"]
# Eliminar un elemento específico
mi_lista.remove("Python")
print(mi_lista)  # Salida: [1, "nuevo valor", "Elemento insertado", 3, 5.5, True, "Nuevo elemento"]
```

1. Sintaxis básica

```
# Eliminar el último elemento
mi_lista.pop()
print(mi_lista)  # Salida: [1, "nuevo valor", "Elemento insertado", 3, 5.5, True]

# Eliminar un elemento por su índice
del mi_lista[2]
print(mi_lista)  # Salida: [1, "nuevo valor", 3, 5.5, True]
```

1. Sintaxis básica

```
# Ordenar la lista de forma ascendente
numeros.sort()
print(numeros)  # Salida: [1, 2, 5, 7, 9]

# Ordenar la lista de forma descendente
numeros.sort(reverse=True)
print(numeros)  # Salida: [9, 7, 5, 2, 1]
```

1. Sintaxis básica

```
# Invertir el orden de los elementos de la lista
numeros.reverse()
print(numeros)  # Salida: [1, 2, 5, 7, 9]
# Obtener la longitud de la lista
longitud = len(mi_lista)
print(longitud)  # Salida: 5

# Verificar si un elemento está en la lista
existe = 3 in mi_lista
print(existe)  # Salida: True
```

2- Estructuras de control

```
edad = 20
if edad > 18:
    print("Es mayor de edad")
elif edad == 18:
    print("Tiene 18 años")
else:
    print("Es menor de edad")
```

2- Estructuras de control

```
materias = ["Matemáticas", "Historia", "Ciencias"]
for materia in materias:
    print("Estudiando:", materia)
```

```
materias = ["Matemáticas", "Historia", "Ciencias"]
for indice in range(0, len(materias)):
    print("Estudiando:", materias[indice])
```

2- Estructuras de control

```
contador = 1
while contador <= 5:
    print("Contador:", contador)
    contador += 1
```

3- Funciones y módulos

```
def saludar(nombre):
    return "Hola, "+nombre + "!"

print(saludar("Ana"))   # Resultado: Hola, Ana!
```

3- Funciones y módulos

```
import math

# Usando la función sqrt del módulo math
raiz_cuadrada = math.sqrt(16)
print(raiz_cuadrada)  # Resultado: 4.0

from mi_modulo import algo
algo()
```

4. Conceptos básicos de programación orientada a objetos (POO)

```
# Definición de la clase Persona
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
        self.__pv = "privado"
    def saludar(self):
        return f"Hola, me llamo {self.nombre} y tengo {self.edad} años."

    def __privada(self):
        return "Soy un método privado."

# Creación de un objeto de la clase Persona
persona1 = Persona("Ana", 25)
print(persona1.saludar())  # Resultado: Hola, me llamo Ana y tengo 25 años.
print(persona1.__privada())  # Resultado: AttributeError: 'Persona' object has no attribute
'__privada'
print(persona1.__pv)  # Resultado: AttributeError: 'Persona' object has no attribute '__pv'
```

Desarrollo de software para blockchain 2024

Clase 02 - Ethereum

Conceptos

- Ethereum
- Algoritmos de hash que utiliza
- Curva elíptica que utiliza
- Algoritmo de consenso: PoW - PoS
- Smart Contract

Ethereum

Ethereum(2013)

Blockchain inicialmente propuesta en el 2013: whitepaper donde su primera implementación fue en 2015.

Características técnicas en su implementación:

- **Hash Keccak-256 (pre SHA-3)**

- **Criptografía de clave pública(asimétrica):** Idem Bitcoin (secp256k1)

Ethereum: Generación de claves(idem Bitcoin salvo …)

1- Generación de seed phrase (idem Bitcoin BIP39).

2- Transformación de la semilla en una semilla binaria utilizando la función de derivación de clave PBKDF2 con HMAC-SHA512.

3- Usando la semilla binaria del paso anterior, se aplica el proceso BIP-32 para obtener la clave privada maestra y la cadena de derivación maestra. Aunque Ethereum y Bitcoin ambos utilizan BIP-32, *Ethereum no suele usar la jerarquía compleja de claves que permite BIP-32*. En su lugar, muchas wallets de Ethereum *simplemente derivan claves individuales directamente de la clave privada maestra*.

4- Uso de secp256k1 para obtener la clave pública: Este paso es idéntico tanto en Bitcoin como en Ethereum.

5- Generación de la dirección Ethereum: se deriva de la clave pública. *Se toma la clave pública, se aplica Keccak-256 y luego se toman los últimos 20 bytes del hash resultante*. Esta dirección se muestra comúnmente en formato hexadecimal y es la que se utiliza para recibir fondos y para identificar cuentas en la red Ethereum.

Ethereum: Generación de claves

```python
# pip install eth-utils eth-keys mnemonic
from eth_keys import keys
from mnemonic import Mnemonic
from eth_utils import keccak
# 1. Generación de la frase semilla (seed phrase)
mnemo = Mnemonic("english")
seed_phrase = mnemo.generate(strength=256)
print("Seed Phrase:", seed_phrase)
# 2. Transformación de la seed phrase en una seed
seed = Mnemonic.to_seed(seed_phrase)
# Función para derivar la clave privada a partir de la seed y un índice
def derive_private_key(seed, index):
    # Usamos keccak256 como función hash
    return keccak(seed + index.to_bytes(4, byteorder='big'))

# Derivamos dos claves privadas con índices diferentes
private_key1 = derive_private_key(seed, 0)
private_key2 = derive_private_key(seed, 1)

# Convertimos las claves privadas en claves públicas y direcciones
private_key_obj1 = keys.PrivateKey(private_key1)
public_key_obj1 = private_key_obj1.public_key
address1 = public_key_obj1.to_address()

private_key_obj2 = keys.PrivateKey(private_key2)
public_key_obj2 = private_key_obj2.public_key
address2 = public_key_obj2.to_address()

print("\nPrivate Key 1:", private_key_obj1.to_hex())
print("Public Key 1:", public_key_obj1.to_hex())
print("Address 1:", address1)

print("\nPrivate Key 2:", private_key_obj2.to_hex())
print("Public Key 2:", public_key_obj2.to_hex())
print("Address 2:", address2)
```

Algoritmo de consenso: PoW to PoS

Inicialmente desde su implementación Ethereum como algoritmo de consenso utilizaba PoW, como en Bitcoin, el 15/9/22 pasó del algoritmo de consenso PoW a PoS, evento fue conocido como *The Merge*.

- **Validadores en lugar de mineros:** En lugar de mineros que compiten por resolver problemas matemáticos difíciles como en PoW, PoS utiliza validadores. Los validadores son participantes de la red que bloquean (o "apuestan") una cierta cantidad de ETH como garantía.
- **Proposición de bloques:** De manera aleatoria, según varios factores, incluida la cantidad de ETH apostada, un validador es seleccionado para proponer un bloque.
- **Validaciones:** Otros validadores luego atestiguan o validan el bloque propuesto. Estas validaciones sirven como votos para la validación del bloque
- **Finalidad:** Una vez que se recibe un cierto número de validaciones para un bloque, ese bloque se finaliza, lo que significa que es parte permanente de la cadena y no puede ser revertido.
- **Recompensas y penalizaciones:** Los validadores son recompensados por proponer y atestiguar correctamente bloques. Sin embargo, pueden ser penalizados (perder parte de su garantía) si actúan de manera maliciosa o si no están disponibles para realizar las tareas cuando se les solicita.

Smart contract: qué es?

programa autoejecutable que se despliega y se ejecuta en la blockchain. Estos programas permiten la ejecución automática de lógica de negocio cuando se cumplen ciertas condiciones predefinidas, sin requerir la intervención de intermediarios.

Smart contract: características

- **Inmutabilidad:** Una vez que un smart contract se despliega en la blockchain, su código no puede ser modificado. Esto garantiza que el contrato se ejecute siempre según su programación original.

- **Transparencia:** El código fuente de la mayoría de los smart contracts en Ethereum es visible públicamente, lo que permite a cualquiera verificar su funcionamiento.

- **Seguridad:** Los smart contracts en Ethereum se ejecutan en un entorno sandbox (la Ethereum Virtual Machine o EVM), lo que significa que no pueden afectar directamente a otras aplicaciones o partes del sistema fuera de su propio contexto.

- **Decentralización:** Al ser ejecutados en la blockchain de Ethereum, los smart contracts se benefician de las propiedades descentralizadas de la red, incluida la resistencia a la censura y la manipulación.

Smart contract: características cont...

- **Interconexión**: Los smart contracts pueden interactuar entre sí, permitiendo la creación de aplicaciones descentralizadas complejas compuestas por múltiples contratos.

- **Gas y Tarifas:** La ejecución de funciones dentro de un smart contract requiere gas, que es una medida de la cantidad de trabajo computacional requerido. Los usuarios pagan tarifas en Ether (la criptomoneda nativa de Ethereum) para compensar a los validadores por ejecutar y validar las transacciones y contratos.

- **Lenguajes de Programación:** Los smart contracts en Ethereum suelen ser escritos en lenguajes como Solidity o Vyper, que luego se compilan a *bytecode* para ser ejecutados en la EVM.

Conceptos: nonce, gas, gas limit, wei, gas price

nonce: representa el número de transacciones enviadas desde la dirección de esa cuenta. Este valor es utilizado para ordenar las transacciones enviadas desde una cuenta y para prevenir el doble gasto o la repetición de transacciones. Cada vez que una cuenta envía una transacción, su nonce aumenta en uno. Al requerir que las transacciones tengan nonces en orden consecutivo, la red garantiza que cada transacción se procese en orden y solo una vez.

gas: sirve para medir la cantidad de trabajo computacional necesario para ejecutar operaciones, como hacer transacciones o ejecutar contratos inteligentes.

gas limit: especifica la cantidad máxima de gas que va a utilizar una txn. Si una txn utiliza más gas del que se especificó en el límite, la txn se detiene, pero el gas gastado hasta ese momento no te es reembolsado. A su vez cada bloque tiene un gas limit

wei: es la unidad más pequeña de ether en Ethereum. 1 Ether = $1*10^{18}$ wei o 1 wei = $1*10^{-18}$ Ether

gas price: representa la cantidad de ether (ETH) que un remitente está dispuesto a pagar por cada unidad de gas al realizar una transacción o ejecutar una operación en un contrato inteligente. Se suele medir en gwei.

ej: https://etherscan.io/tx/0xb13e3b53fd09b01bea46796ef4bf530e7b281ec58293b6d94a1430be3c2f25d6

Tipos de cuentas en EVM

En el contexto de la Ethereum Virtual Machine (EVM) y la blockchain de Ethereum, existen principalmente dos tipos de cuentas:

1- Cuentas de Propietario Externas (Externally Owned Accounts, EOAs):

- Son controladas por claves privadas y no tienen código asociado a ellas.
- Las EOAs son básicamente las "wallets" de los usuarios. Desde estas cuentas, los propietarios pueden enviar transacciones que transfieran Ether, interactuar con contratos, o incluso crear nuevos contratos.
- Las transacciones originadas desde EOAs pueden incluir mensajes hacia otros EOAs o contratos.
- Estas cuentas tienen un saldo en Ether y pueden recibir o enviar Ether.

Tipos de cuentas en EVM

2- Cuentas de Contrato (Contract Accounts):

- Son controladas por su código de contrato y pueden ser activadas solo por una EOA o por otros contratos a través de un mensaje.
- Cada cuenta de contrato tiene un código asociado, que se ejecuta cada vez que se recibe un mensaje en esa cuenta.
- Estas cuentas también tienen un saldo en Ether.
- Tienen un almacenamiento persistente, lo que permite almacenar y recuperar datos entre llamadas o transacciones.
- La creación de una cuenta de contrato ocurre cuando un contrato se despliega en la red.

Tipos de cuentas en EVM: como se genera una address de contrato?

En Ethereum, la dirección de un contrato se genera deterministicamente a partir de la dirección de la cuenta que crea el contrato (generalmente una Externally Owned Account o EOA) y del nonce de esa cuenta. El proceso es el siguiente:

1. Crear una transacción: Se una transacción especial donde la dirección de destino (to) está vacío, y los datos (data field) contienen el código del contrato en bytecode.

2. Generación de la dirección del contrato: La dirección de un contrato se genera mediante el hash de la dirección del creador del contrato (la EOA) y su nonce. Específicamente, se utiliza la función keccak256 (el algoritmo de hash usado en Ethereum, similar a SHA-3) para obtener el hash de la concatenación de la dirección del creador y su nonce. Luego, se toman los últimos 20 bytes del hash resultante, y eso se convierte en la dirección del contrato.

3. Desplegar el contrato: Una vez que la transacción es confirmada y añadida a un bloque, el contrato es oficialmente desplegado y la dirección del contrato se puede usar para interactuar con él.

Tipos de cuentas en EVM: como se genera una address de contrato?

Es importante destacar que debido a que este proceso es determinístico, si se conoce la dirección del creador y su nonce, siempre se podrá predecir la dirección del contrato antes de que este sea desplegado.

```
#pip install web3
from web3 import Web3
def generate_contract_address(account_address: str, nonce: int) -> str:
    """
    Genera la dirección de un contrato Ethereum a partir de la dirección del creador y un nonce.
    """
    rlp_encoded = Web3.toBytes(hexstr=account_address) + Web3.toBytes(nonce)
    contract_address = Web3.keccak(rlp_encoded)[-20:].hex()
    return contract_address
# Dirección de la cuenta creadora (ejemplo)
account_address = '0x742d35Cc6634C0532925a3b844Bc454e4438f44e'
nonce = 1
contract_address = generate_contract_address(account_address, nonce)
print(f"Contract Address: {contract_address}")
```