Script

Implementación

¿Qué nos falta?

Ya que implementamos Tx

Para poder crear transacciones necesitamos poder:

- Crear/serializar scripts de Script!!!

Para poder validar transacciones necesitamos poder:

- Correr los scripts de Script!!!

La clase que viene lo implementaremos

Scripts

Comandos

Dos tipos:

1. OP_CODES
2. Bytes (data)

Script

Op codes

OP_DUP	Duplicates the top item on the stack
OP_HASH160	Hashes twice: first using SHA-256 and then RIPEMD-160
OP_EQUALVERIFY	Returns true if the inputs are equal. Returns false and marks the transaction as invalid if they are unequal
OP_CHECKSIG	Checks that the input signature is a valid signature using the input public key for the hash of the current transaction

Implementación de Op codes

OP_DUP, OP_VERIFY

```python
def op_verify(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    if decode_num(element) == 0:
        return False
    return True


def op_dup(stack):
    if len(stack) < 1:
        return False
    stack.append(stack[-1])
    return True
```

Siempre manipulan el stack!!!

Implementación de Op codes

OP_DUP, OP_VERIFY

```
def op_verify(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    if decode_num(element) == 0:
        return False
    return True


def op_dup(stack):
    if len(stack) < 1:
        return False
    stack.append(stack[-1])
    return True
```

Implementación de Op codes

OP_DUP, OP_VERIFY

```python
def op_verify(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    if decode_num(element) == 0:
        return False
    return True


def op_dup(stack):
    if len(stack) < 1:
        return False
    stack.append(stack[-1])
    return True
```

Números son un poco raros en Script!

Implementación de Op codes

Números en Script

```python
def encode_num(num):
    if num == 0:
        return b''
    abs_num = abs(num)
    negative = num < 0
    result = bytearray()
    while abs_num:
        result.append(abs_num & 0xff)
        abs_num >>= 8
    # if the top bit is set,
    # for negative numbers we ensure that the top bit is
    # for positive numbers we ensure that the top bit is
    if result[-1] & 0x80:
        if negative:
            result.append(0x80)
        else:
            result.append(0)
    elif negative:
        result[-1] |= 0x80
    return bytes(result)
```

```python
def decode_num(element):
    if element == b'':
        return 0
    # reverse for big endian
    big_endian = element[::-1]
    # top bit being 1 means it's negative
    if big_endian[0] & 0x80:
        negative = True
        result = big_endian[0] & 0x7f
    else:
        negative = False
        result = big_endian[0]
    for c in big_endian[1:]:
        result <<= 8
        result += c
    if negative:
        return -result
    else:
        return result
```

Implementación de Op codes

OP_EQUAL

```
def op_equal(stack):
    if len(stack) < 2:
        return False
    element1 = stack.pop()
    element2 = stack.pop()
    if element1 == element2:
        stack.append(encode_num(1))
    else:
        stack.append(encode_num(0))
    return True


def op_equalverify(stack):
    return op_equal(stack) and op_verify(stack)
```

Implementación de Op codes

OP_HASH

```
def op_hash160(stack):
    # check that there's at least 1 element on the stack
    if len(stack) < 1:
        return False
    # pop off the top element from the stack
    element = stack.pop()
    # push a hash160 of the popped off element to the stack
    h160 = hash160(element)
    stack.append(h160)
    return True


def op_hash256(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    stack.append(hash256(element))
    return True
```

Implementación de Op codes

OP_CHECKSIG

```python
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pubkey
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER signature
    # take off the last byte of the signature as that's the hash_type
    # More at https://en.bitcoin.it/wiki/OP_CHECKSIG
    der_signature = stack.pop()[:-1]
    # parse the serialized pubkey and signature into objects
    try:
        point = S256Point.parse(sec_pubkey)
        sig = Signature.parse(der_signature)
    except (ValueError, SyntaxError):
        #print('Parse fail', point)
        return False
    # verify the signature using S256Point.verify()
    # push an encoded 1 or 0 depending on whether the signature verified
    if point.verify(z, sig):
        stack.append(encode_num(1))
    else:
        stack.append(encode_num(0))
    return True
```

Implementación de Op codes

OP_CHECKSIG

```python
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pa
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER signature
    # take off the last byte of the signature as that's th
    # More at https://en.bitcoin.it/wiki/OP_CHECKSIG
    der_signature = stack.pop()[:-1]
    # parse the serialized pubkey and signature into objects
    try:
        point = S256Point.parse(sec_pubkey)
        sig = Signature.parse(der_signature)
    except (ValueError, SyntaxError):
        #print('Parse fail', point)
        return False
    # verify the signature using S256Point.verify()
    # push an encoded 1 or 0 depending on whether the signature verified
    if point.verify(z, sig):
        stack.append(encode_num(1))
    else:
        stack.append(encode_num(0))
    return True
```

Input: stack, y hash para firmar (como int)

Implementación de Op codes

OP_CHECKSIG

```
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pubkey
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER sig
    # take off the last byte of the signature as
    # More at https://en.bitcoin.it/wiki/OP_C
    der_signature = stack.pop()[:-1]
    # parse the serialized pubkey and signature i
    try:
        point = S256Point.parse(sec_pubkey)
        sig = Signature.parse(der_signature)
    except (ValueError, SyntaxError):
        #print('Parse fail', point)
        return False
    # verify the signature using S256Point.verify()
    # push an encoded 1 or 0 depending on whether the signature verified
    if point.verify(z, sig):
        stack.append(encode_num(1))
    else:
        stack.append(encode_num(0))
    return True
```

Cuando uno hace la firma en Bitcoin se concatena el hashType (SIGHASH_ALL) al final!!! Pero 1 byte!!!

Implementación de Op codes

OP_CHECKSIG

```python
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pubkey
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER signature
    # take off the last byte of the signature as that's the hash_type
    # More at https://en.bitcoin.it/wiki/OP_CHECKSIG
    der_signature = stack.pop()[:-1]
    # parse the serialized pubkey and signature into objects
    try:
        point = S256Point.parse(sec_pubkey)
        sig = Signature.parse(der_signature)
    except
        #p
        re
    # veri
    # push
    if poi
        stack.append(encode_num(1))
    else:
        stack.append(encode_num(0))
    return True
```

```python
def op_checksigverify(stack, z):
    return op_checksig(stack, z) and op_verify(stack)
```

Scripts

Comandos

Dos tipos:

1. OP_CODES
2. Bytes (data)

¿Cómo diferencio unos de los otros?

Scripts

Comandos

Script = secuencia de bytes (cuando lo recibo)

0x00 = OP_0
0x51 = OP_1
…
0x60 = OP_16
0x76 = OP_DUP
0xa9 = OP_HASH160
0xac = OP_CHECKSIG

Scripts

Comandos

Script = secuencia de bytes (cuando lo recibo)

0x00 = OP_0
0x51 = OP_1
…
0x60 = OP_16
0x76 = OP_DUP
0xa9 = OP_HASH160
0xac = OP_CHECKSIG

Scripts

...mandos

Script = secuencia de bytes (c...

0x00 = OP_0
0x51 = OP_1
...
0x60 = OP_16
0x76 = OP_DUP
0xa9 = OP_HASH160
0xac = OP_CHECKSIG

Un salto entre 0 y 1!!!
Aquí van los datos

Documentación:
https://en.bitcoin.it/wiki/Script

Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

Si leo un byte *n* entre 0x01 y 0x4b, los siguiente *n* bytes son data

- Por ejemplo una firma DER
- Una llave pública SEC
- Un redeem script (para P2SH)

Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

Si leo un byte *n* entre 0x01 y 0x4b, los siguiente *n* bytes son data

- Por ejemplo una firma DER – 70 bytes
- Una llave pública SEC – 33/65 bytes
- Un redeem script (para P2SH) -- ???

Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

Si leo un byte *n* entre 0x01 y 0x4b, los siguiente *n* bytes son data

- Por ejemplo una firma DER – 70 bytes
- Una llave pública SEC – 33/65 bytes
- Un redeem script (para P2SH)

    OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG

    Small (24 bytes)

Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

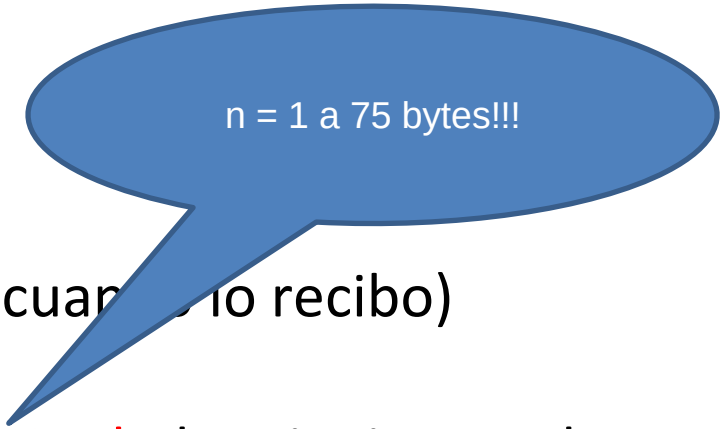
Si leo un byte *n* entre 0x01 y 0x4b, los siguiente *n* bytes son data

- Por ejemplo una firma DER – 70 bytes
- Una llave pública SEC – 33/65 bytes
- Un redeem script (para P2SH)

  MULTISIG 5 of 20!!! (20x SEC)

Scripts

Data



Script = secuencia de bytes (cuando lo recibo)

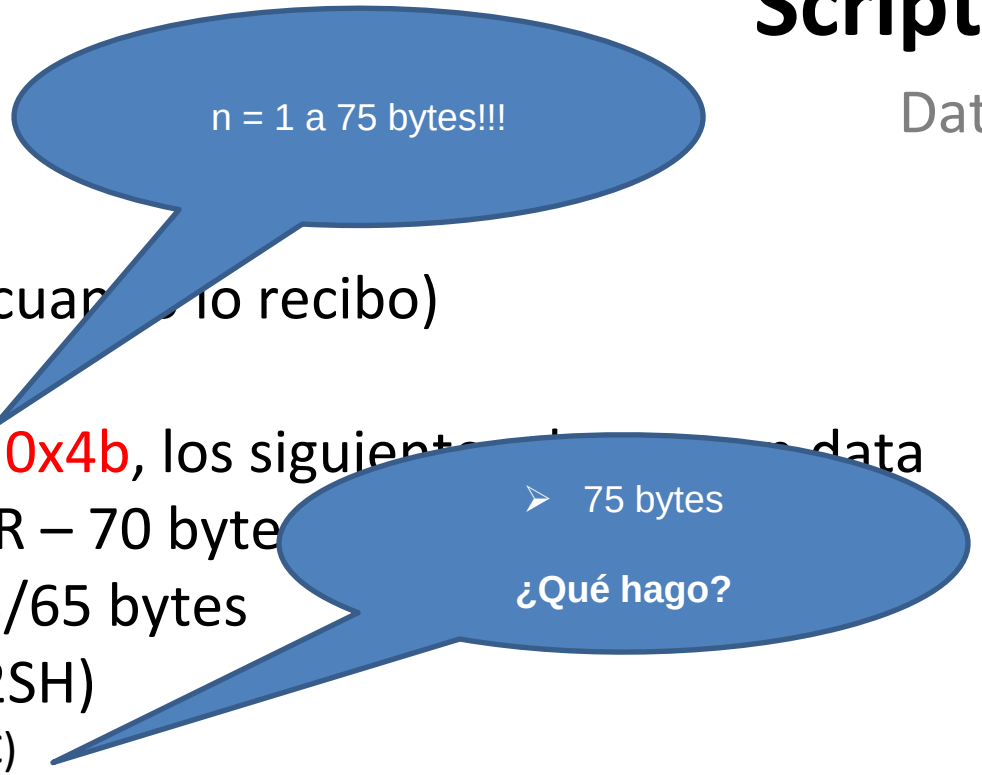
Si leo un byte *n* entre 0x01 y 0x4b, los siguiente *n* bytes son data

- Por ejemplo una firma DER – 70 bytes
- Una llave pública SEC – 33/65 bytes
- Un redeem script (para P2SH)

  MULTISIG 5 of 20!!! (20x SEC)

Scripts

Data



Script = secuencia de bytes (cuan[illegible] lo recibo)

Si leo un byte *n* entre 0x01 y 0x4b, los siguient[illegible] data

- Por ejemplo una firma DER – 70 byte
- Una llave pública SEC – 33/65 bytes
- Un redeem script (para P2SH)

  MULTISIG 5 of 20!!! (20x SEC)

Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

Si leo 0x4c = 76 = OP_PUSHDATA1 -- siguiente byte = largo de data
0x4c 0xff <data de largo 255 bytes>– de 75 a 255 bytes

Si leo 0x4d = 77 = OP_PUSHDATA2 -- siguiente 2 bytes = largo

0x4c b1 b2 <data de largo 520 bytes> – de 256 a 520 bytes

Scripts

Data

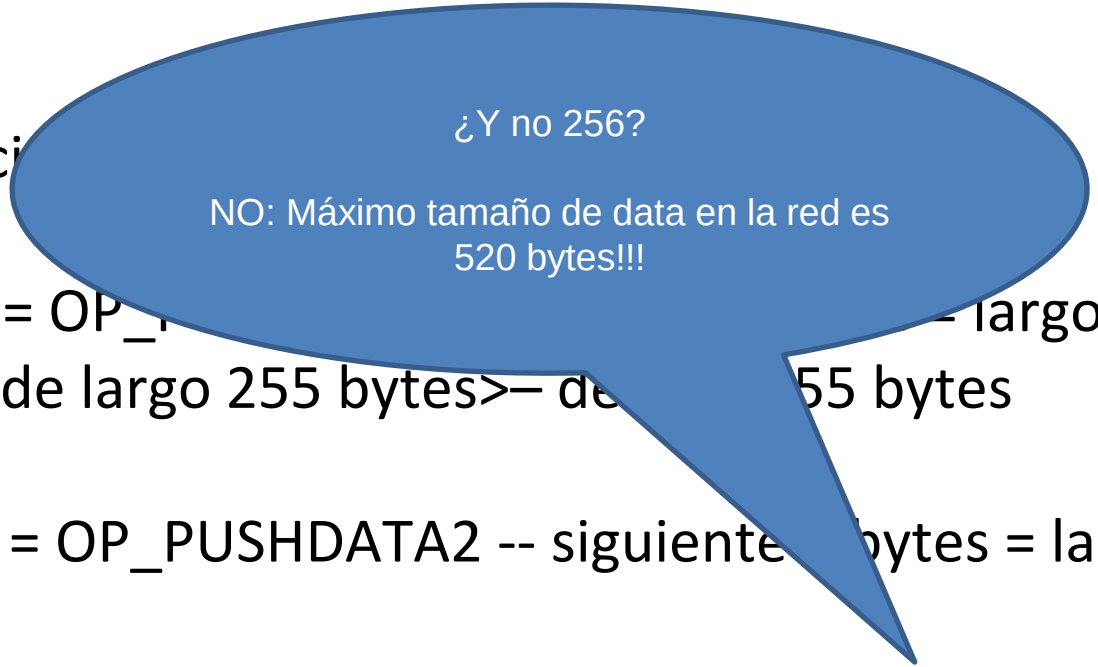
Script = secuenci

Si leo 0x4c = 76 = OP_ – largo de data

0x4c 0xff <data de largo 255 bytes>– de 55 bytes

Si leo 0x4d = 77 = OP_PUSHDATA2 -- siguiente bytes = largo

0x4c b1 b2 <data de largo 520 bytes> – de 256 a 520 bytes



Scripts

Data

Script = secuencia de bytes (cuando lo recibo)

0x4c = 76 = OP_PUSHDATA1: de 75 a 255 bytes

0x4d = 77 = OP_PUSHDATA2: de 256 a 520 bytes

0x4e = OP_PUSHDATA4: largo de 4 bytes; no sirve pa nada

Scripts

Comandos

Dos tipos:

1. OP_CODES
2. Bytes (data)

¿Qué ahora?

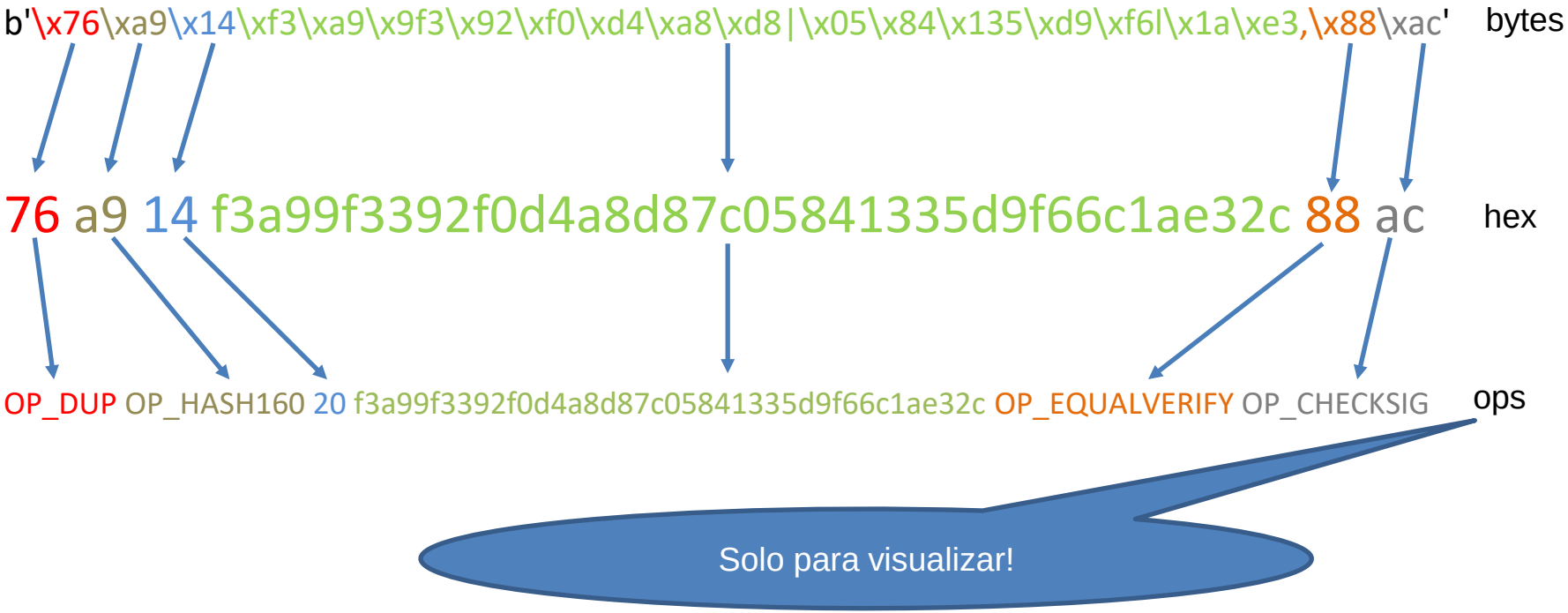
- Parse (bytes)
- Serialize (to bytes)

Scripts

Parse

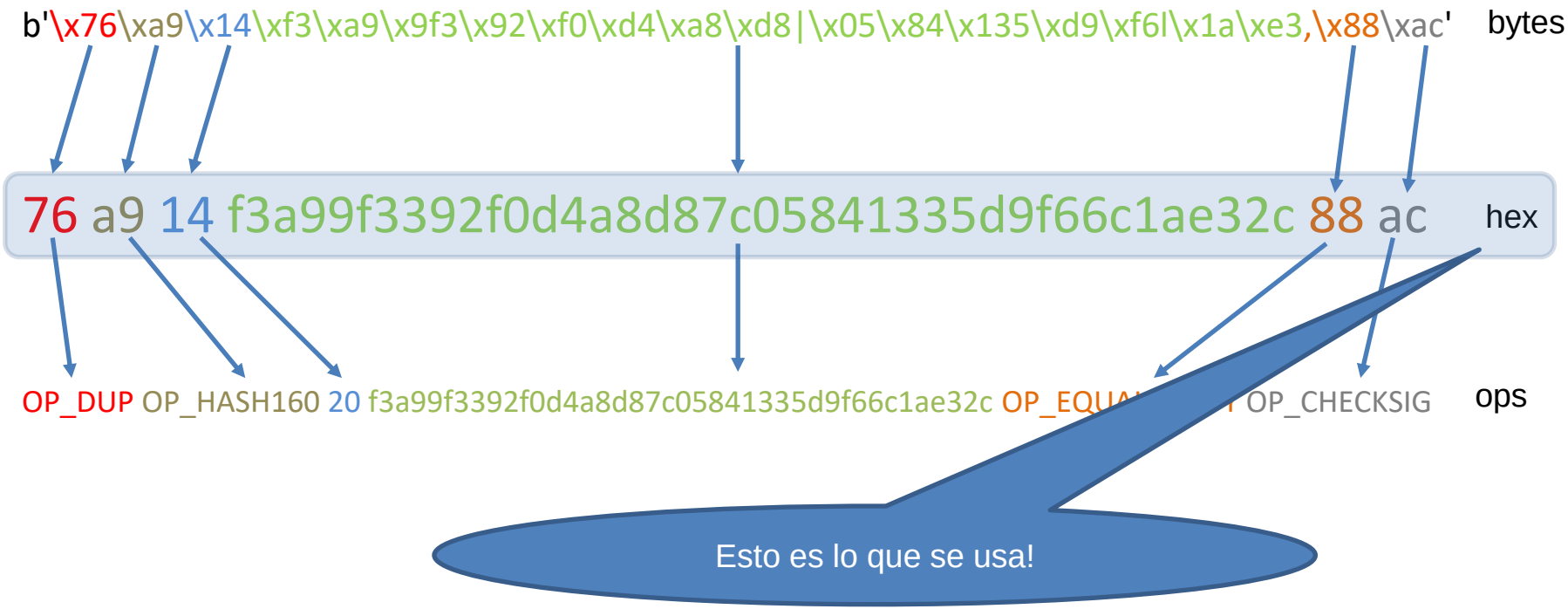
Scripts

Parse



Scripts

Parse



Scripts

Parse

b'\x76\xa9\x14\xf3\xa9\x9f3\x92\xf0\xd4\xa8\xd8|\x05 c' bytes

76 a9 14 f3a99f3392f0d4a8d87c0584 hex

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f6 ops

```
OP_CODE_NAMES = {
    0: 'OP_0',
    76: 'OP_PUSHDATA1',
    77: 'OP_PUSHDATA2',
    78: 'OP_PUSHDATA4',
    79: 'OP_1NEGATE',
    81: 'OP_1',
    82: 'OP_2',
    83: 'OP_3',

    ...
```

Esto es lo que se usa!

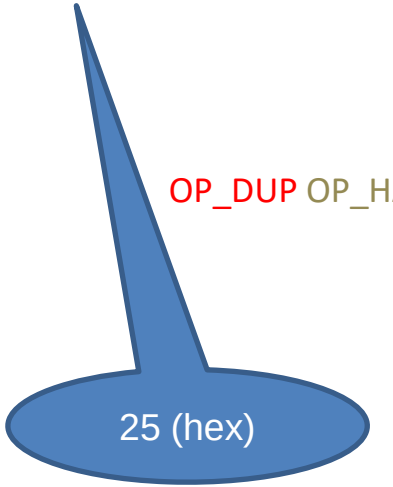
Scripts

Parse

Serialización empieza con el largo del script (varint)

19 76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG



Scripts

Parse

```
class Script:

    # The basic constructor; either empty list of commands, or some commands
    def __init__(self, cmds=None):
        if cmds is None:
            self.cmds = []
        else:
            self.cmds = cmds
```

Scripts

Parse

```python
@classmethod
def parse(cls, s):
    # get the length of the entire field
    length = read_varint(s)
    # initialize the cmds array
    cmds = []
    # initialize the number of bytes we've read to 0
    count = 0
    # loop until we've read length bytes
    while count < length:
        # get the current byte
        current = s.read(1)
        # increment the bytes we've read
        count += 1
        # convert the current byte to an integer
        current_byte = current[0]
        # if the current byte is between 1 and 75 inclusive
        if current_byte >= 1 and current_byte <= 75:
            # we have an cmd set n to be the current byte
            n = current_byte
            # add the next n bytes as an cmd
            cmds.append(s.read(n))
            # increase the count by n
            count += n
        # I'm omitting OP_PUSHDATA1 and OP_PUSHDATA2
        else:
            # we have an opcode. set the current byte to op_code
            op_code = current_byte
            # add the op_code to the list of cmds
            cmds.append(op_code)
    if count != length:
        raise SyntaxError('parsing script failed')
    return cls(cmds)
```

Scripts

Serialize

Scripts

Serialize

76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

\x19 \x76 \xa9 \x14 f3a99f3392… bytes

Scripts

Serialize

```python
def raw_serialize(self):
    # initialize what we'll send back
    result = b''
    # go through each cmd
    for cmd in self.cmds:
        # if the cmd is an integer, it's an opcode
        if type(cmd) == int:
            # turn the cmd into a single byte integer using int_to_little_endian
            result += int_to_little_endian(cmd, 1)
        else:
            # otherwise, this is an element
            # get the length in bytes
            length = len(cmd)
            # We'll assume at most 75 bytes of data
            if length >= 75:
                raise ValueError('too long an cmd')
                # Technically we should support up to 520 bytes; same as with OP_PUSHDATA1/2
            result += int_to_little_endian(length, 1)
            result += cmd
    return result
```

Scripts

Serialize

```
def raw_serialize(self):
    # initialize what we'll send back
    result = b''
    # go through each cmd
    for cmd in self.cmds:
        # if the cmd is an integer, it's an opcode
        if type(cmd) == int:
            # turn the cmd into a single byte integer using int_to_little_endian
            result += int_to_little_endian(cmd, 1)
        else:
    retur
```

```
# This just completes the job and wraps everything as needed
def serialize(self):
    # get the raw serialization (no prepended length)
    result = self.raw_serialize()
    # get the length of the whole thing
    total = len(result)
    # encode_varint the total length of the result and prepend
    return encode_varint(total) + result
```

Scripts

Evaluate

¿Qué nos falta?

- Ejecutar un Script

¿Cómo hacer esto?

- Comando por comando utilizando el stack

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we hav
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via      e that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an opcode, so do what the opcod
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking      GVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can a      d z
            if cmd in (172, 173):
                # these are signing operations, they need a
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution al
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

In op.py

```python
OP_CODE_FUNCTIONS = {
    105: op_verify,
    118: op_dup,
    135: op_equal,
    136: op_equalverify,
    147: op_add,
    169: op_hash160,
    170: op_hash256,
    172: op_checksig,
    173: op_checksigverify
}

OP_CODE_NAMES = {
    0: 'OP_0',
    76: 'OP_PUSHDATA1',
    77: 'OP_PUSHDATA2',
    78: 'OP_PUSHDATA4',
    79: 'OP_1NEGATE',
    81: 'OP_1',
    82: 'OP_2',
    83: 'OP_3',

    ...
```

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an a[illegible]s to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do wha[illegible]opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checki[illegible]are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can al[illegible]that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

sig_hash
i.e. Lo que hay que firmar

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

CHECKSIG usa z!!!

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has [illegible] acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do [illegible] the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for c[illegible]ig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you ca[illegible]k that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_h
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Solo para P2SH

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, do what the opcode says
            operation = OP_CODE_FUNCTIONS[
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Stack.init()
No implementaremos altstack

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, do what the opcode says
            operation = OP_CODE_FUNCTIONS[cm
            # The commands that will need the z king sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script yo check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a si
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Ejecución sigue hasta que hay comandos para ejecutar

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an p[illegible]ode, so do what the opcode says
            operation = OP_CODE_FUNCTI[illegible][cmd]
            # The commands that will need t[illegible] for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Scri[illegible]an also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

El comando es OPcode

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z fo hecking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you lso check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a si
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

¿Qué función ocuparemos?

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Para CHEKCSIG hay que pasar el hash como el argumento también

Scripts

Evaluate

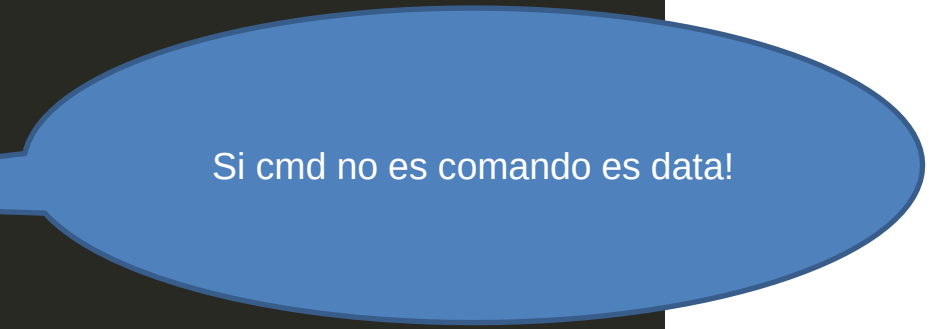
```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Otros comandos necesitan solo el stack!!!

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```



Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack th          ion failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Al final stack no puede ser vacio

Scripts

Evaluate

```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Al final stack no puede tener False!

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what the opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a sig_hash
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

Ejecución exitosa!

Scripts

Evaluate

```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an po[illegible]de, so do what the opcode says
            operation = OP_CODE_FUNCTIO[illegible]md]
            # The commands that will need th[illegible]n checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Scrip[illegible]n also check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need [illegible]
                # to check against
                if not operation(stack, z):
                    return False
            else:
                if not operation(stack):
                    return False
        else:
            # cmd is data, so add the cmd to the stack
            stack.append(cmd)

    # If we end up with an empty stack the execution failed
    if len(stack) == 0:
        return False
    # If we end up with False on the stack, the execution also failed
    # Need to tell you about numbers a bit more
    if stack.pop() == b'':
        return False
    return True
```

No implementamos:

1) IF/ELSE
2) Altstack
3) MULTISIG

Scripts

Verificación

```python
class Tx:
    ...
    def verify_input(self, input_index):
        '''Returns whether the input has a valid signature'''
        # get the relevant input
        tx_in = self.tx_ins[input_index]
        # grab the previous ScriptPubKey
        script_pubkey = tx_in.script_pubkey(testnet=self.testnet)

        # P2SH should be handled differently, but for now we only implement P2PKH
        z = self.sig_hash(input_index, None)

        # combine the current ScriptSig and the previous ScriptPubKey
        # This is checked once the transaction has been signed
        combined = tx_in.script_sig + script_pubkey
        # evaluate the combined script
        return combined.evaluate(z)

    def verify(self):
        '''Verify this transaction'''
        # check that we're not creating money
        if self.fee() < 0:
            return False
        # check that each input has a valid ScriptSig
        for i in range(len(self.tx_ins)):
            if not self.verify_input(i):
                return False
        return True
```

Scripts

Firmando

```
class Tx:

    ...

    # This sets the ScriptSig for spending stuff; it basically provides the correct signature
    # Useful for p2pk and p2pkh
    def sign_input(self, input_index, private_key):
        '''Signs the input using the private key'''
        # get the signature hash (z)
        z = self.sig_hash(input_index)
        # get der signature of z from private key
        der = private_key.sign(z).der()
        # append the SIGHASH_ALL to der (use SIGHASH_ALL.to_bytes(1, 'big'))
        sig = der + SIGHASH_ALL.to_bytes(1, 'big')
        # calculate the sec
        sec = private_key.point.sec()
        # initialize a new script with [sig, sec] as the cmds
        script_sig = Script([sig, sec])
        # change input's script_sig to new script
        self.tx_ins[input_index].script_sig = script_sig
        # return whether sig is valid using self.verify_input
        return self.verify_input(input_index)
```

Gastando plata

¿Cómo gasto mis bitcoins?

- ¿Una dirección es una llave pública?
- ¿Para qué sirve una dirección?
- ¿Cómo encuentro mis bitcoins?
- Waaaaa!!!

Gastando plata

¿Cómo gasto mis bitcoins?

- **¿Una dirección es una llave pública? – solo para P2PK**
- ¿Para qué sirve una dirección?
- ¿Cómo encuentro mis bitcoins?
- Waaaaa!!!

Gastando plata

¿Cómo gasto mis bitcoins?

- ¿Una dirección es una llave pública? – solo para P2PK
- **¿Para qué sirve una dirección? – P2PKH, P2SH, …**
- ¿Cómo encuentro mis bitcoins?
- Waaaaa!!!

Gastando plata

¿Cómo gasto mis bitcoins?

- ¿Una dirección es una llave pública? – solo para P2PK
- ¿Para qué sirve una dirección? – P2PKH, P2SH, …
- **¿Cómo encuentro mis bitcoins? – ya lo saben (full node)**
- Waaaaa!!!

Gastando plata

¿Cómo gasto mis bitcoins?

- ¿Una dirección es una llave pública? – solo para P2PK
- ¿Para qué sirve una dirección? – P2PKH, P2SH, …
- ¿Cómo encuentro mis bitcoins? – ya lo saben (full node)
- **Waaaaa**!!! – OK, le haré la vida más fácil!

Gastando plata

¿Cómo gasto mis bitcoins?

- ~~¿Una dirección es una llave pública? – solo para P2PK~~
- ~~¿Para qué sirve una dirección? – P2PKH, P2SH, …~~
- ~~¿Cómo encuentro mis bitcoins? – ya lo saben (full node)~~
- ~~**Waaaaa**!!! – OK, le haré la vida más fácil!~~

Hay que mandarlos a un script!!!

¿Cuál script?

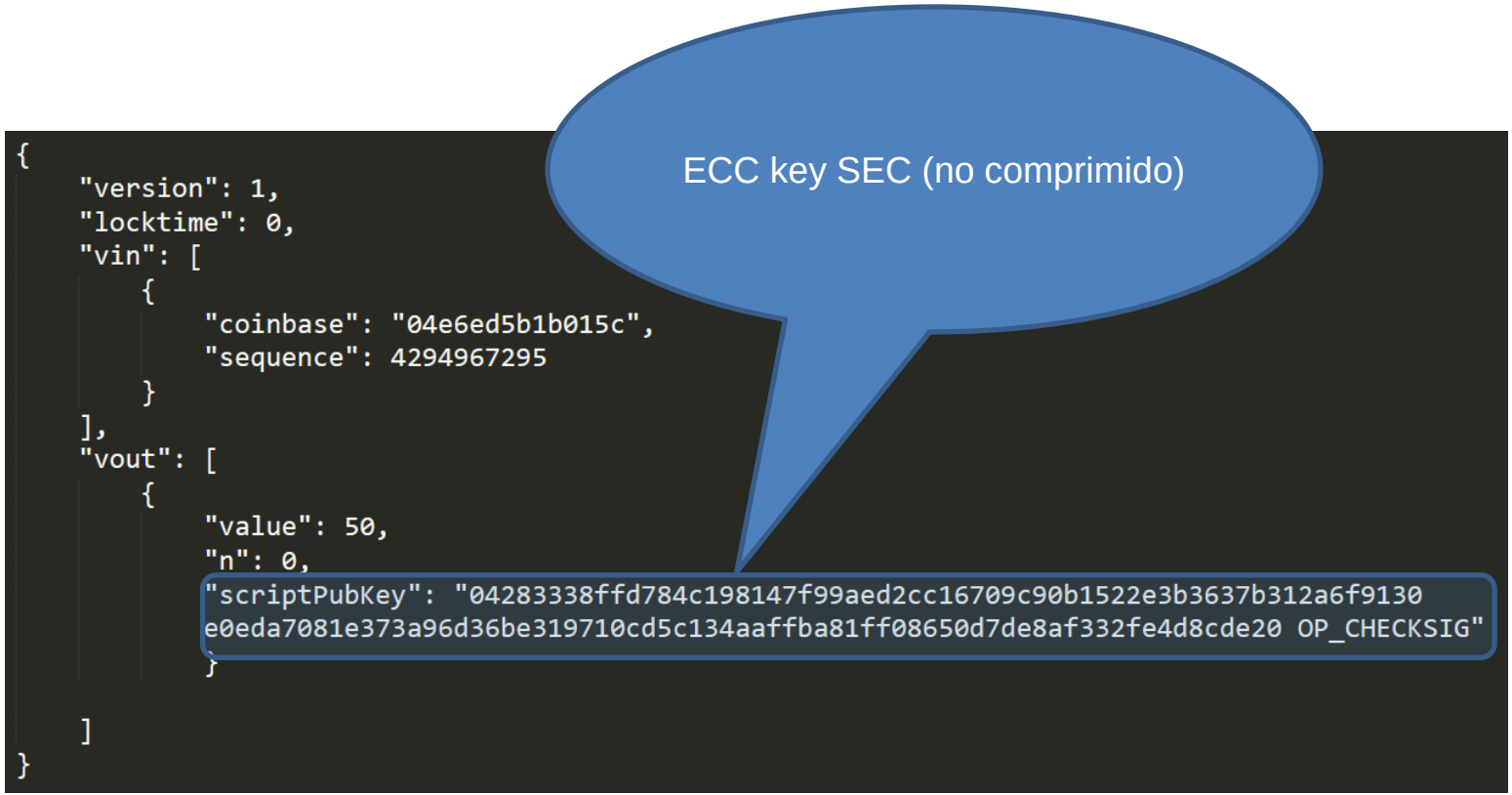
Direcciones

P2PK


```
{
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "coinbase": "04e6ed5b1b015c",
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 50,
            "n": 0,
            "scriptPubKey": "04283338ffd784c198147f99aed2cc16709c90b1522e3b3637b312a6f9130
            e0eda7081e373a96d36be319710cd5c134aaffba81ff08650d7de8af332fe4d8cde20 OP_CHECKSIG"
        }

    ]
}
```

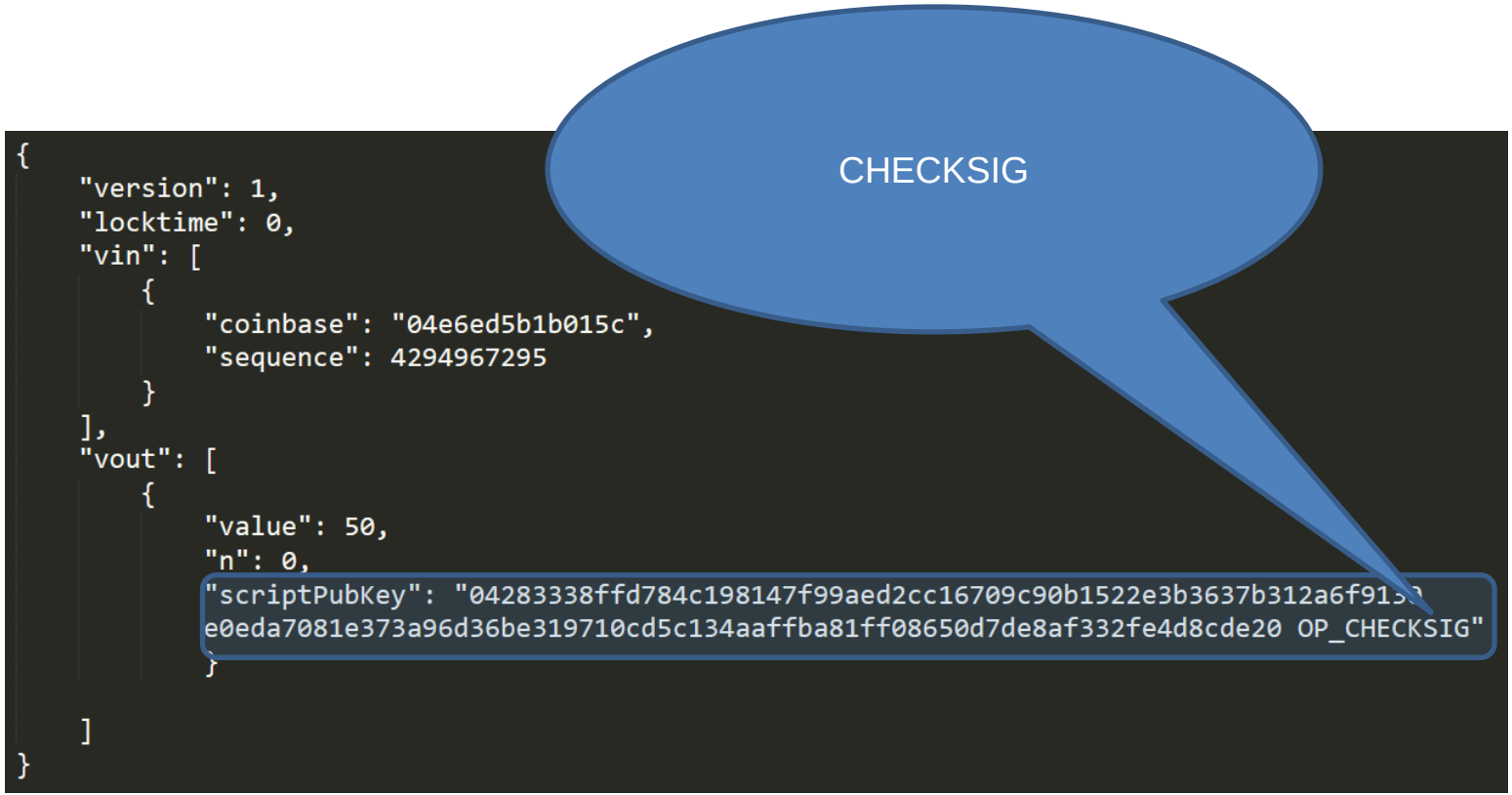
Direcciones

P2PK



Direcciones

P2PK

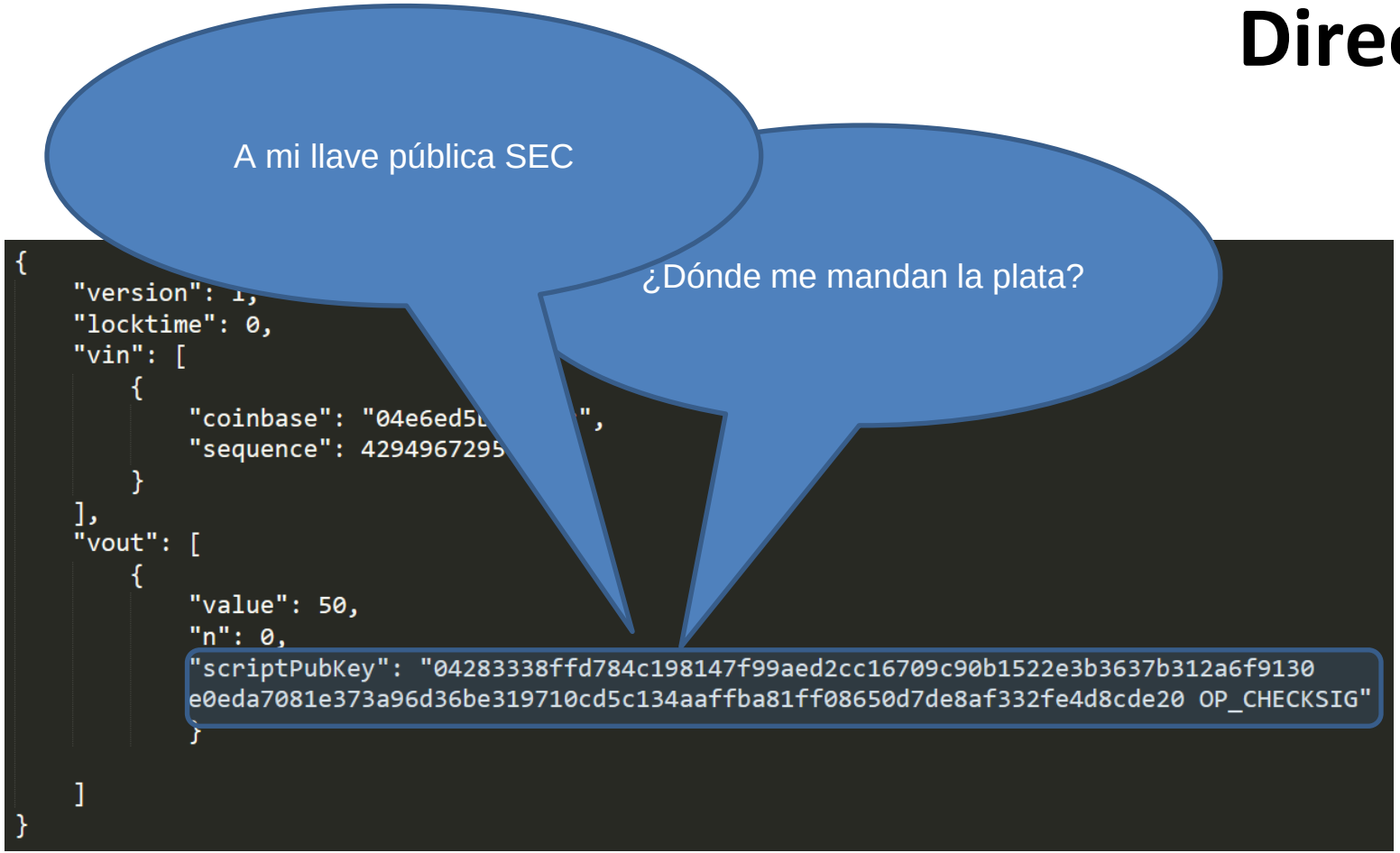


Direcciones

P2PK

Direcciones

P2PK



Direcciones

P2PK

~~A mi llave pública SEC~~

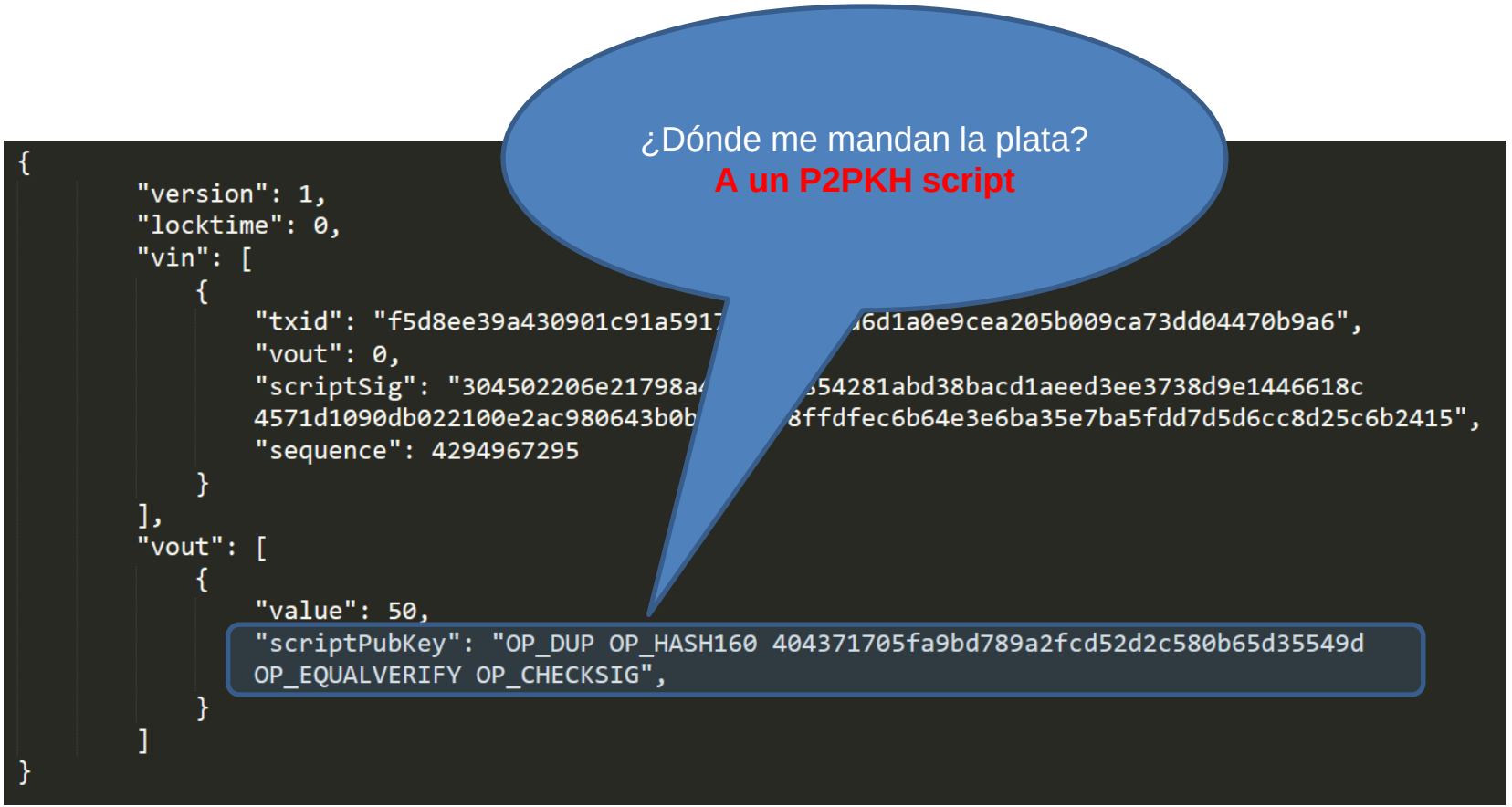
Script: SEC + OP_CHECKSIG

¿Dónde me mandan la plata?

```
{
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "coinbase": "04e6ed5[illegible]",
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 50,
            "n": 0,
            "scriptPubKey": "04283338ffd784c198147f99aed2cc16709c90b1522e3b3637b312a6f9130
            e0eda7081e373a96d36be319710cd5c134aaffba81ff08650d7de8af332fe4d8cde20 OP_CHECKSIG"
        }

    ]
}
```

Direcciones

P2PKH



Direcciones

P2PKH

¿Cómo les digo esto?

Con mi dirección!

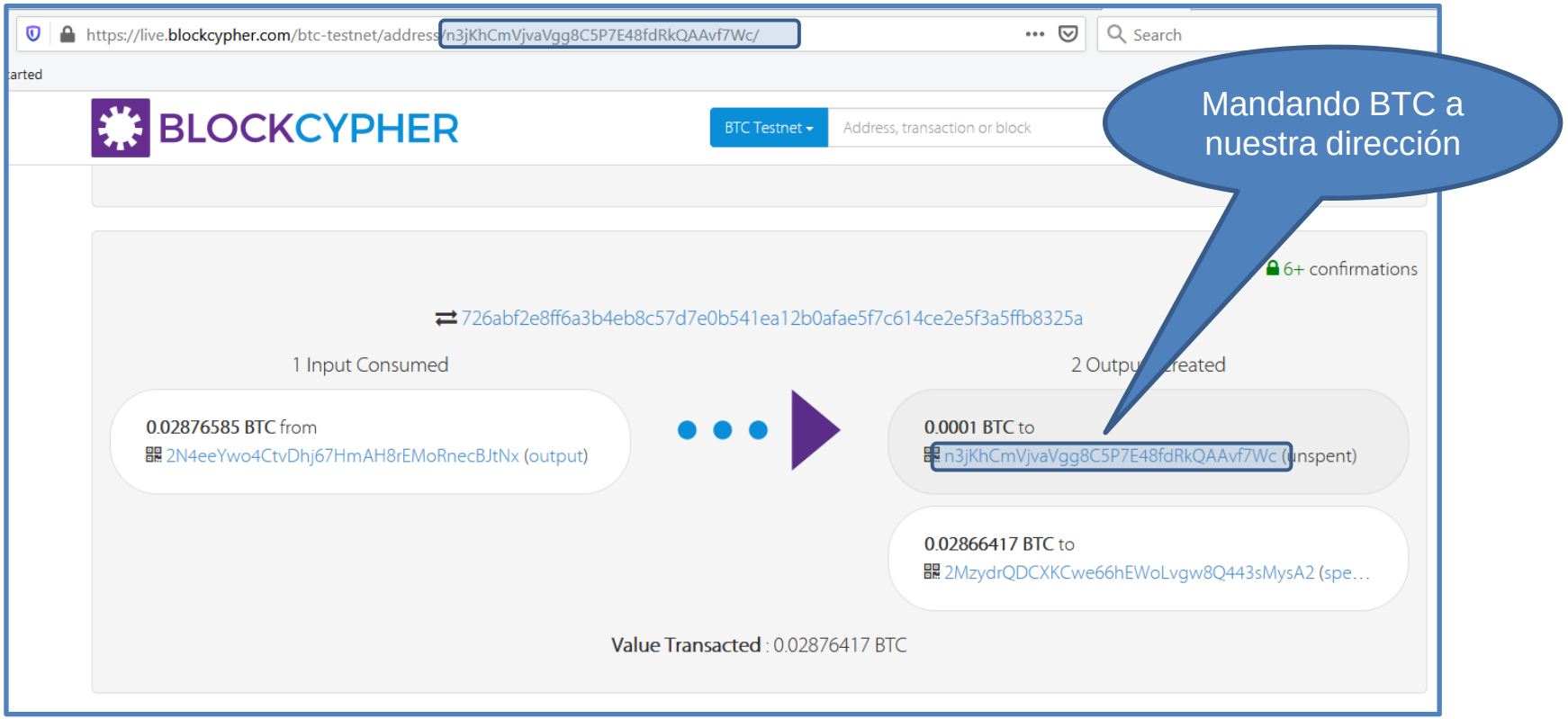
Dónde me mandan la plata?

A un P2PKH script


```
{
        "version": 1,
        "locktime": 0,
        "vin": [
            {
                "txid": "f5d8ee39a430901c91a5917                6d1a0e9cea205b009ca73dd04470b9a6",
                "vout": 0,
                "scriptSig": "304502206e21798a4                54281abd38bacd1aeed3ee3738d9e1446618c
                4571d1090db022100e2ac980643b0b                8ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b2415",
                "sequence": 4294967295
            }
        ],
        "vout": [
            {
                "value": 50,
                "scriptPubKey": "OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d
                OP_EQUALVERIFY OP_CHECKSIG",
            }
        ]
}
```

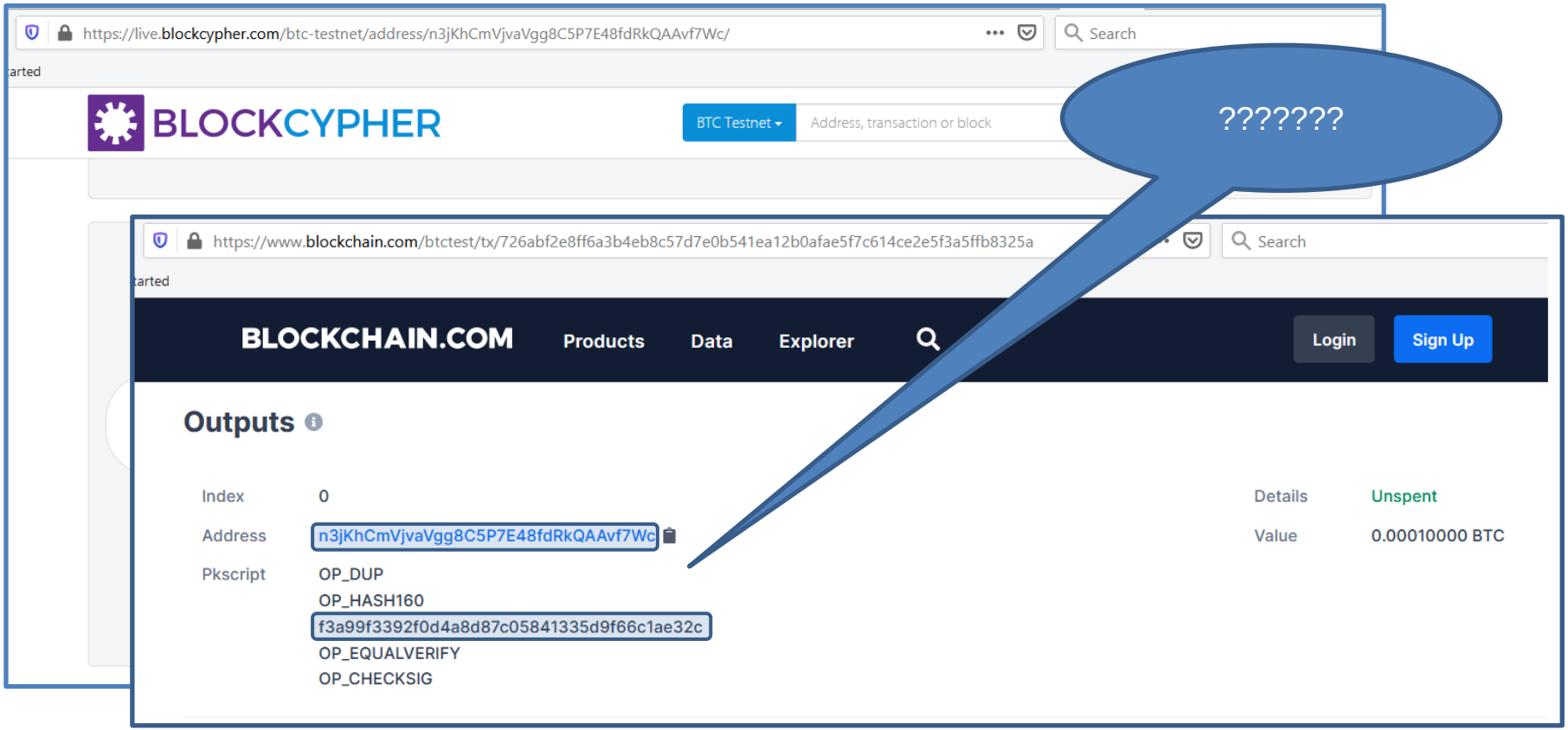
Direcciones

P2PKH



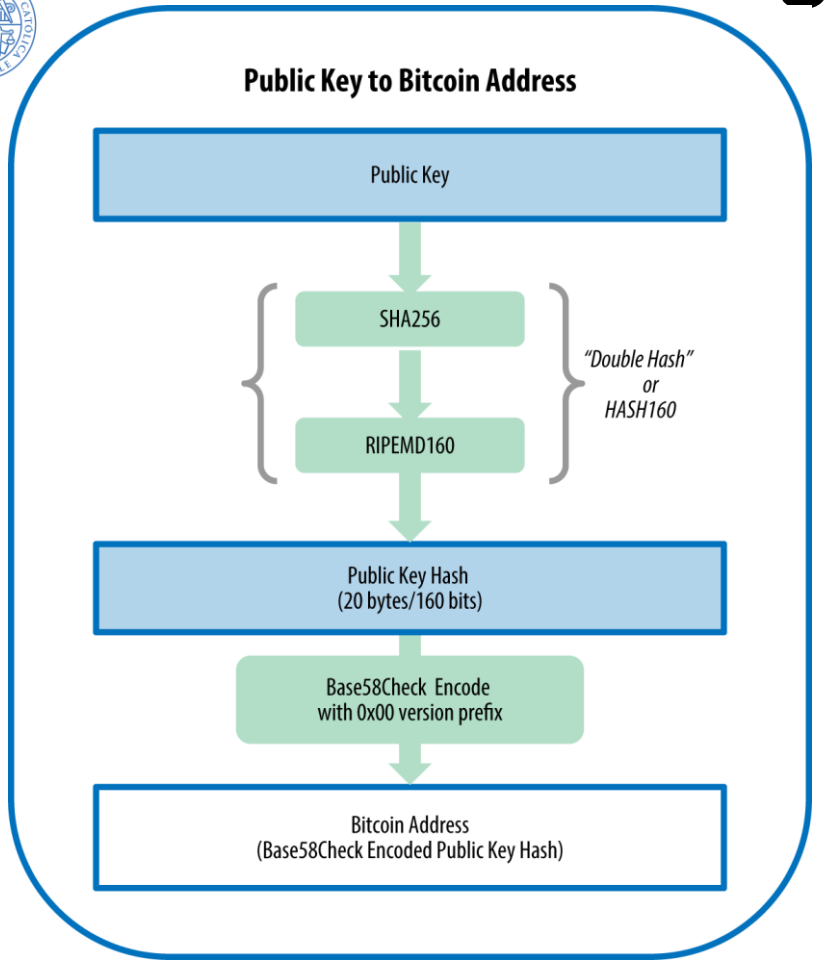
Direcciones

P2PKH



Una dirección de Bitcoin

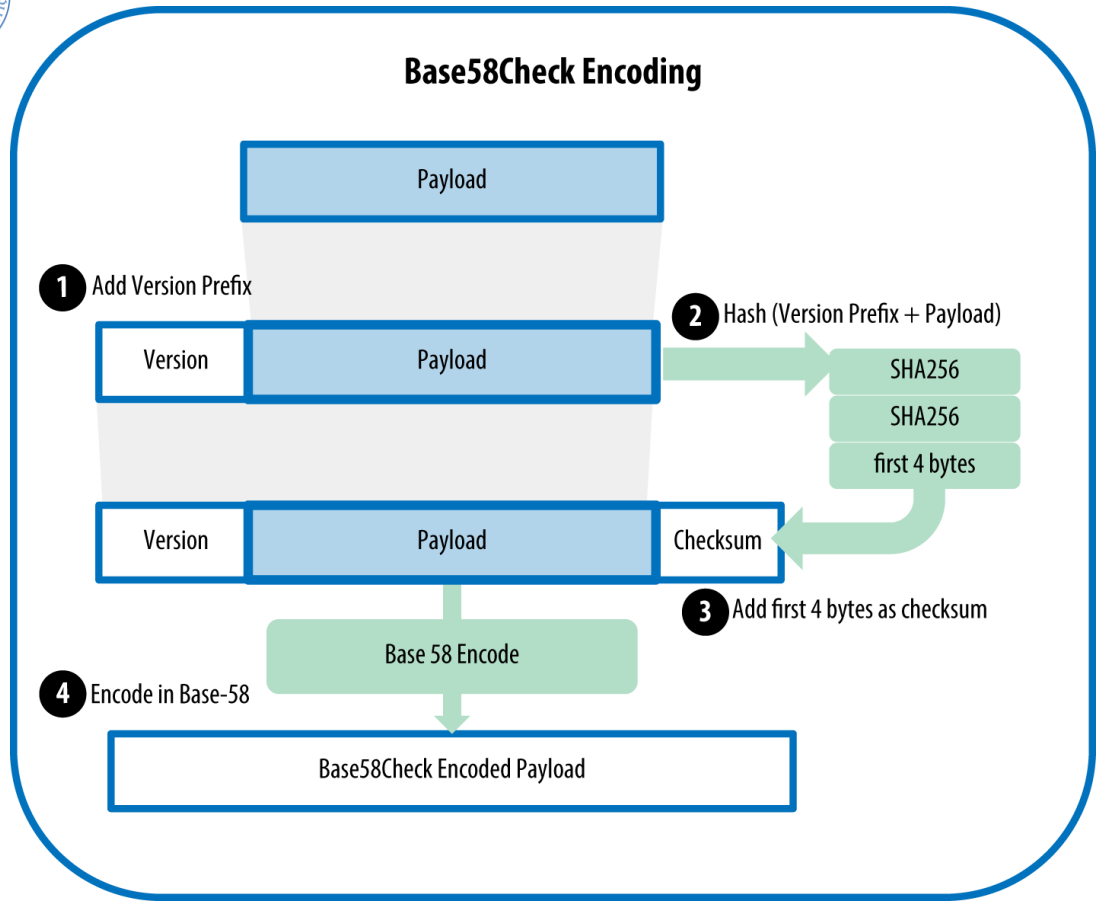
No es una llave publica



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc

Una dirección de Bitcoin

No es una llave publica



https://github.com/bitcoinbook/bitcoinbook/blob/develop/ch04.asciidoc

Direcciones de BitCoin

Para P2PKH!!!!

P una llave en SEC

¿Cómo saco la dirección de Bitcoin?

1. *version = 0x00* para mainnet, y *0x6f* para testnet
2. *key = ripemd160(sha256(P))*
3. *checksum = sha256(sha256(version + key))*[:4]
4. *res = version + key + checksum*

Return encode_base58(res):

- Produce una cosa empezando con 1 (mainnet), o m/n (testnet)
- Este prefijo me dice que tipo de dirección es!
- https://en.bitcoin.it/wiki/List_of_address_prefixes

Direcciones de BitCoin

Para P2PKH!!!!

Una dirección con prefijo *0x00* o *0x6f* (1,m,n en base58):

- Me dice: paga me a P2PKH
- OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG

¿A cuál <hash>?

- D mi dirección P2PKH
- Convierto D desde base 58 a bytes
- Saco el prefijo
- Saco el checksum
- El resultado es mi <hash>

Direcciones de BitCoin

Para P2PKH!!!!

```python
def decode_base58(s):
    num = 0
    for c in s:
        num *= 58
        num += BASE58_ALPHABET.index(c)
    combined = num.to_bytes(25, byteorder='big')
    checksum = combined[-4:]
    if hash256(combined[:-4])[:4] != checksum:
        raise ValueError('bad address: {} {}'.format(checksum, hash256(combined[:-4])[:4]))
    return combined[1:-4]
```

Verifica checksum

Saca el prefijo y checksum

Direcciones de BitCoin

Para P2PKH!!!!

Una dirección con prefijo *0x00* o *0x6f* (1,m,n en base58):

- Me dice: paga me a P2PKH
- OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG
- Esto será el scriptPubKey a cual se paga!!!

```
# A shortcut for creating a P2PKH script from the p2pkh address
def p2pkh_script_from_address(address):
    '''Takes a p2pkh address and returns the p2pkh ScriptPubKey for this address'''

    # The address format is what you would see on the network; e.g n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc
    # THIS IS NOT IN BYTES!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

    # Check the prefix: https://en.bitcoin.it/wiki/List_of_address_prefixes
    if not ((address[0] == '1') or (address[0] == 'm') or (address[0] == 'n')):
        raise ValueError('not a valid p2pkh address')

    h160 = decode_base58(address)

    return Script([0x76, 0xa9, h160, 0x88, 0xac])
```

Gastando plata

P2PKH

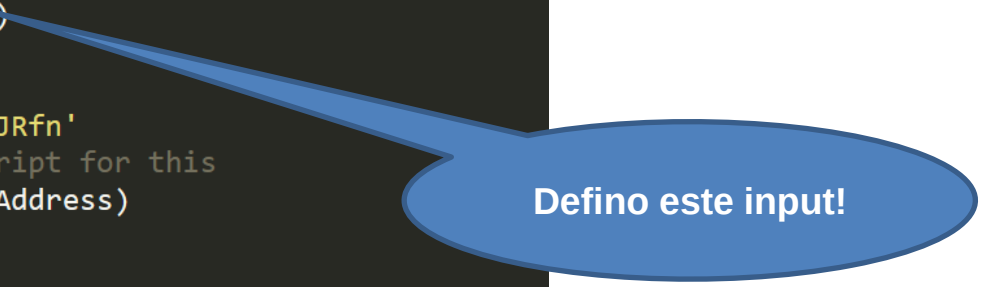
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# Defining the input of our transaction (i.e. the output we will be spending)
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```

Tengo que saber
qué gasto!

Gastando plata

P2PKH

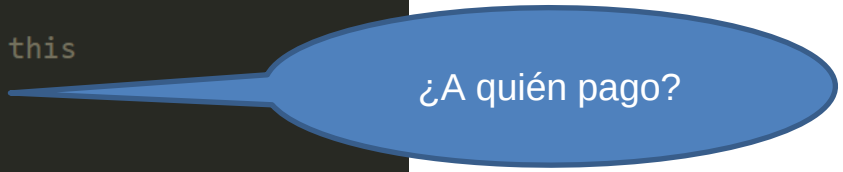
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# Defining the input of our transaction (i.e. the output we will be spending)
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```



Gastando plata

P2PKH

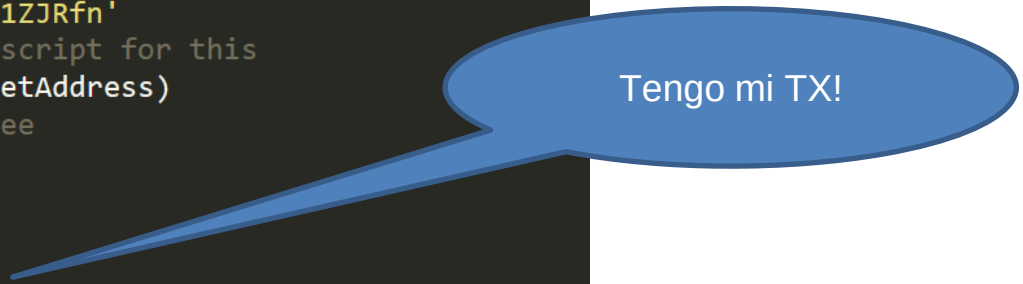
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# Defining the input of our transaction (i.e. the output we will be spending)
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```



Gastando plata

P2PKH

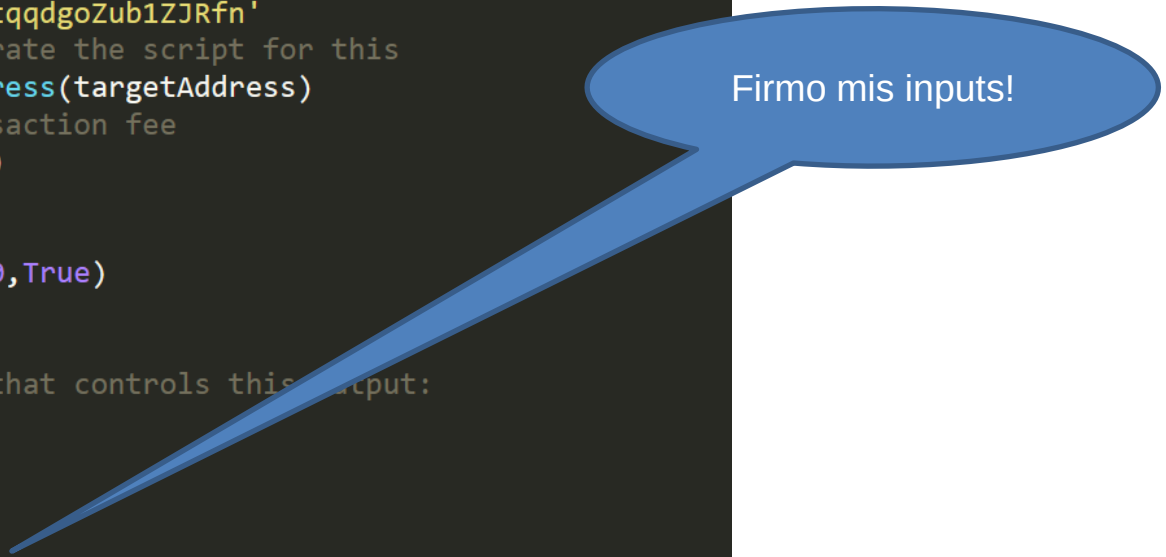
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# Defining the input of our transaction (i.e. the output we will be spending)
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```



Gastando plata

P2PKH

```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# Defining the input of our transaction (i.e. the output we will be spending)
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```



Gastando plata

P2PKH

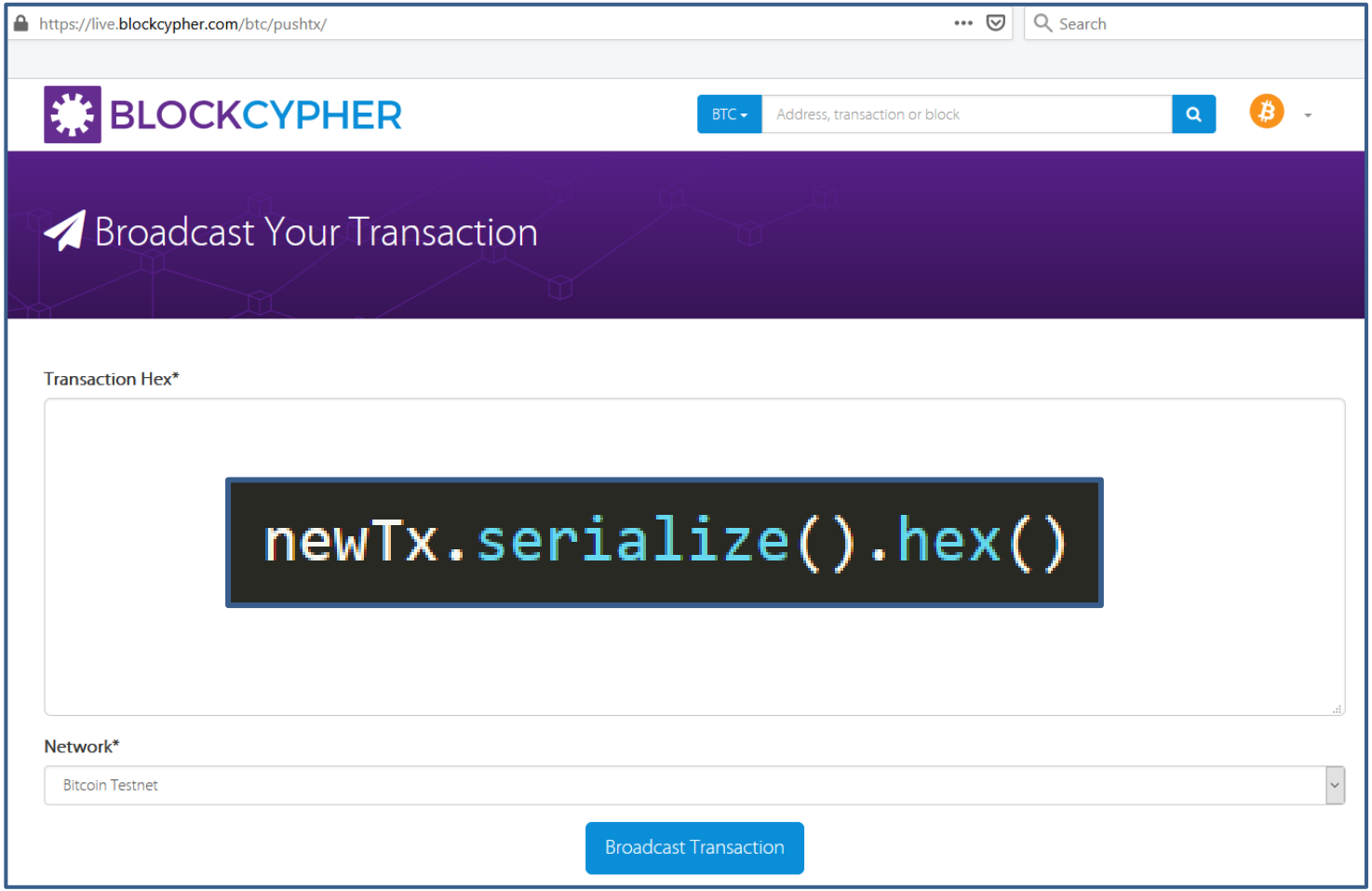
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4
tx_index = 1

# Defining the input of our transaction (i.e. the output we wi
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = 'mipcBbFg9gMiCh81Kj8tqqdgoZub1ZJRfn'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address(targetAddress)
# Define the output, leave some t
newOutput = TxOut(10000,ScriptPub

# Define a new testnet transactio
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```

Esto mando a la red!!!

```
newTx.serialize().hex()
```

Gastando plata

P2PKH



Direcciones

P2SH

Una dirección con prefijo *0x05* o *0xce* (3,2 en base58):

- Me dice: paga me a P2SH
- OP_HASH160 <hash> OP_EQUAL

¿A cuál <hash>?

- D mi dirección P2SH
- Convierto D desde base 58 a bytes
- Saco el prefijo
- Saco el checksum
- El resultado es mi <hash>

Direcciones

P2SH

Una dirección con prefijo *0x05* o *0xc4* (3,2 en base58, mainnet/testnet):

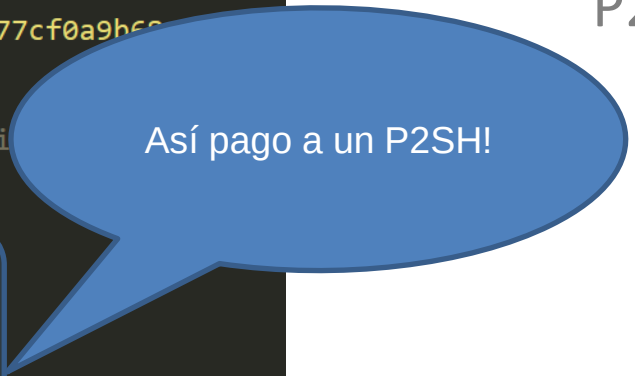
- Me dice: paga me a P2SH
- OP_HASH160 <hash> OP_EQUAL
- Pago a este script:

```
# A shortcut for creating a P2SH script from the p2sh address that appears in the script
def p2sh_script_from_address(address):
    '''Takes a hash160 and returns the p2sh ScriptPubKey'''

    # The address format is what you would see on the network; e.g 3P14159f73E4gFr7JterCCQh9QjiTjiZrG

    # Check the prefix: https://en.bitcoin.it/wiki/List_of_address_prefixes
    if not ((address[0] == '3') or (address[0] == '2')):
        raise ValueError('not a valid p2sh address')

    h160 = decode_base58(address)

    return Script([0xa9, h160, 0x87])
```

Gastando plata

P2SH

```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b6
tx_index = 1

# Defining the input of our transaction (i.e. the output we wi
newInput = TxIn(bytes.fromhex(tx_hash),tx_index)

# To define the output, we need an address
targetAddress = '2NGZrVvZG92qGYqzTLjCAewvPZ7JE8S8VxE'
# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2sh_script_from_address(targetAddress)
# Define the output, leave some transaction fee
newOutput = TxOut(10000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)

# We need to sign our input
# For this, we need the private key that controls this output:
secret = hash256(b'IIC3272Sucks')
intSecret = int(secret.hex(),16)
privKey = PrivateKey(intSecret)

input_index = 0
newTx.sign_input(input_index,privKey)
```



Direcciones

P2SH

¿Cómo genero una P2SH dirección?

- Desde mi redeem script (su serialización en bytes)
- **Necesito tener mi redeem script!!!** (o poder reconstruirlo)

¿Cómo gasto mi P2SH output?

- Con el redeem script
- Y con el unlocking script para este redeem script
- La firma es distinta (scriptSig se reemplaza con el redeem script)

Veamos esto en detalle!

Generando dirección P2SH

ómo genero una P2SH dirección?

```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

newSecret = hash256(b'Jedan2Tri4Pet#$JKl45')
newIntSecret = int(newSecret.hex(),16)
newPrivKey = PrivateKey(newIntSecret)
#newAddress = newPrivKey.point.address(compressed = True, testnet = True)

#print(newAddress)

# Hard code the new p2pkh address (same as newAddress):
newAddress = 'n4LzQsUVB69f8mqytRrBzKLadFnR6go4dg'

# Generate script to be wrapped in P2SH:
redeemScript = p2pkh_script_from_address(newAddress)
# This needs to be raw serialized (no length prefix attached)
h160 = hash160(redeemScript.raw_serialize())

# The hash160 allows us to generate a p2sh address to receive funds
address = addressP2SH(h160, testnet = True)

# The redeem script is wrapped up in this address (same as address; hardcoded)
miP2SHaddress = '2NGFxbNsuYN1dR7JkhBfUs4aMh4iXgtvWM9'
```

Esto genera la dirección desde un h160

Generando dirección P2SH

ómo genero una P2SH dirección?

```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

newSecret = hash256(b'Jedan2Tri4Pet#$JKl45')
newIntSecret = int(newSecret.hex(),16)
newPrivKey = PrivateKey(newIntSecret)
#newAddress = newPrivKey.point.address(compressed = True, testnet = True)

#print(newAddress)

# Hard code the new p2pkh address (same as newAddress):
newAddress = 'n4LzQsUVB69f8mqytRrBzKLadFnR6go4dg'

# Generate script to be wrapped in P2SH:
redeemScript = p2pkh_script_from_address(newAddress)
# This needs to be raw serialized (no length prefix attached)
h160 = hash160(redeemScript.raw_serialize())

# The hash160 allows us to generate a p2sh address to receive funds
address = addressP2SH(h160, testnet = True)

# The redeem script is wrapped up in this address (same as address; hardcoded)
miP2SHaddress = '2NGFxbNsuYN1dR7JkhBfUs4aMh4iXgtvWM9'
```

Primero genero mi redeem script

Generando dirección P2SH

ómo genero una P2SH dirección?

```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

newSecret = hash256(b'Jedan2Tri4Pet#$JKl45')
newIntSecret = int(newSecret.hex(),16)
newPrivKey = PrivateKey(newIntSecret)
#newAddress = newPrivKey.point.address(compressed = True, testnet = True)

#print(newAddress)

# Hard code the new p2pkh address (same as newAddress):
newAddress = 'n4LzQsUVB69f8mqytRrBzKLadFnR6go4dg'

# Generate script to be wrapped in P2SH:
redeemScript = p2pkh_script_from_address(newAddress)
# This needs to be raw serialized (no length prefix attached)
h160 = hash160(redeemScript.raw_serialize())

# The hash160 allows us to generate a p2sh address to receive funds
address = addressP2SH(h160, testnet = True)

# The redeem script is wrapped up in this address (same as address; hardcoded)
miP2SHaddress = '2NGFxbNsuYN1dR7JkhBfUs4aMh4iXgtvWM9'
```

Con esto genero mi dirección P2SH

Generando dirección P2SH

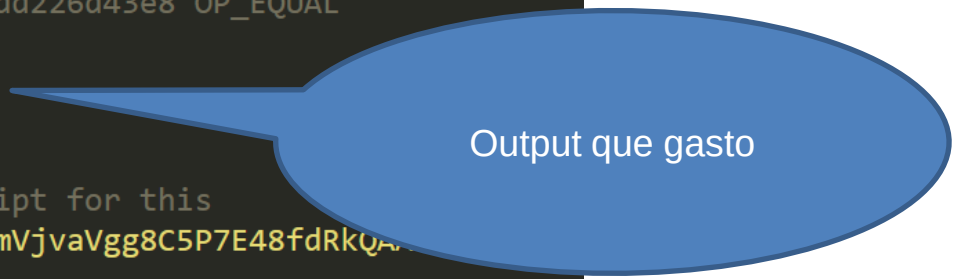
ómo genero una P2SH dirección?

```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

newSecret = hash256(b'Jedan2Tri4Pet#$JKl45')
newIntSecret = int(newSecret.hex(),16)
newPrivKey = PrivateKey(newIntSecret)
#newAddress = newPrivKey.point.address(compressed = True, testnet = True)

#print(newAddress)

# Hard code the new p2pkh address (same as newAddress):
newAddress = 'n4LzQsUVB69f8mqytRrBzKLadFnR6go4dg'

# Generate script to be wrapped in P2SH:
redeemScript = p2pkh_script_from_address(newAddress)
# This needs to be raw serialized (no length prefix attached)
h160 = hash160(redeemScript.raw_serialize())

# The hash160 allows us to generate a p2sh address to receive funds
address = addressP2SH(h160, testnet = True)

# The redeem script is wrapped up in this address (same as address; hardcoded)
miP2SHaddress = '2NGFxbNsuYN1dR7JkhBfUs4aMh4iXgtvWM9'
```

IMPORTANTE:
raw_serialize()
i.e. sin len(script)

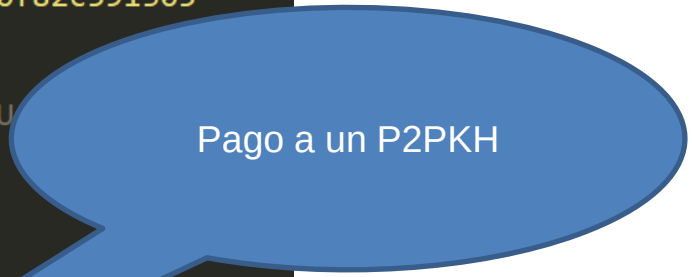
Gastando un P2SH output

```
# Define the input:
tx_hash = '2510f721161210c19abb6f45848f29e645641cb9d60b5e9df6ee20f82c591305'
tx_index = 0

#10000:OP_HASH160 cab93154ada73a646bb01efc393ad5dd226d43e8 OP_EQUAL

newInput = TxIn(bytes.fromhex(tx_hash),tx_index)
# Input defined

# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address('n3jKhCmVjvaVgg8C5P7E48fdRkQA
# input value = 10000 output 5000
newOutput = TxOut(5000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)
# Still unsigned
#print(newTx.serialize().hex())

input_index = 0
newTx.sign_input(input_index,newPrivKey,redeemScript)

to_spend = newTx.serialize().hex()
```



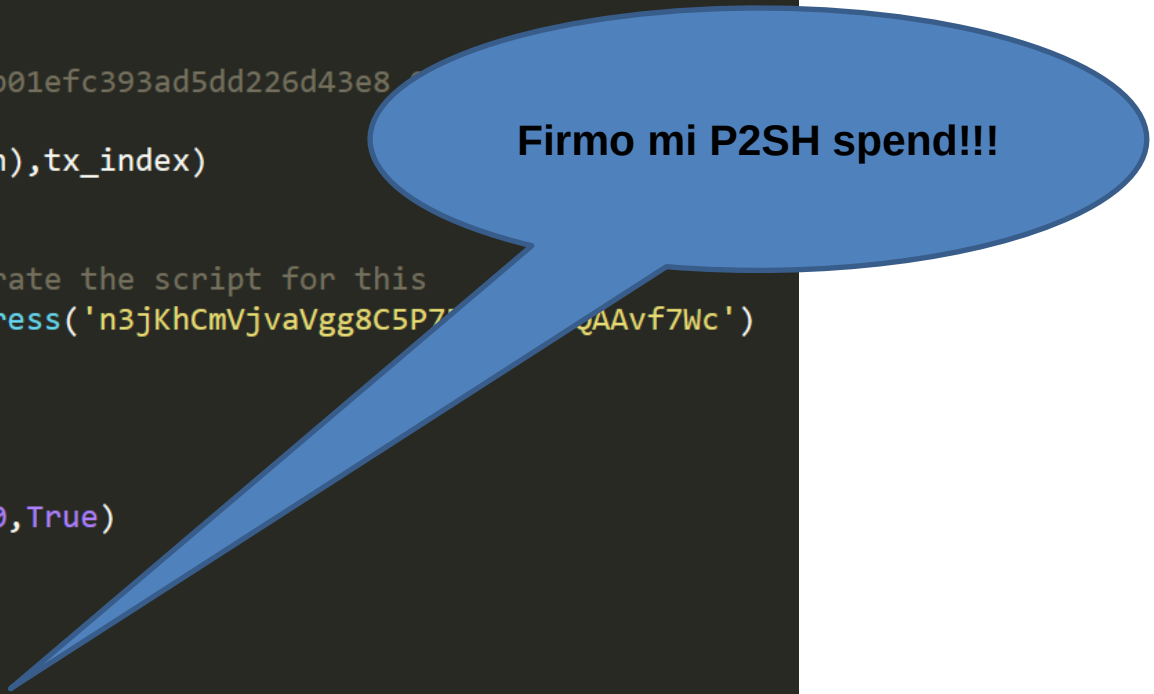
Gastando un P2SH output

```
# Define the input:
tx_hash = '2510f721161210c19abb6f45848f29e645641cb9d60b5e9df6ee20f82c591305'
tx_index = 0

#10000:OP_HASH160 cab93154ada73a646bb01efc393ad5dd226d43e8 OP_EQU

newInput = TxIn(bytes.fromhex(tx_hash),tx_index)
# Input defined

# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address('n3jKhCmVjvaVgg8C5                dRkQAAvf7Wc')
# input value = 10000 output 5000
newOutput = TxOut(5000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)
# Still unsigned
#print(newTx.serialize().hex())

input_index = 0
newTx.sign_input(input_index,newPrivKey,redeemScript)

to_spend = newTx.serialize().hex()
```



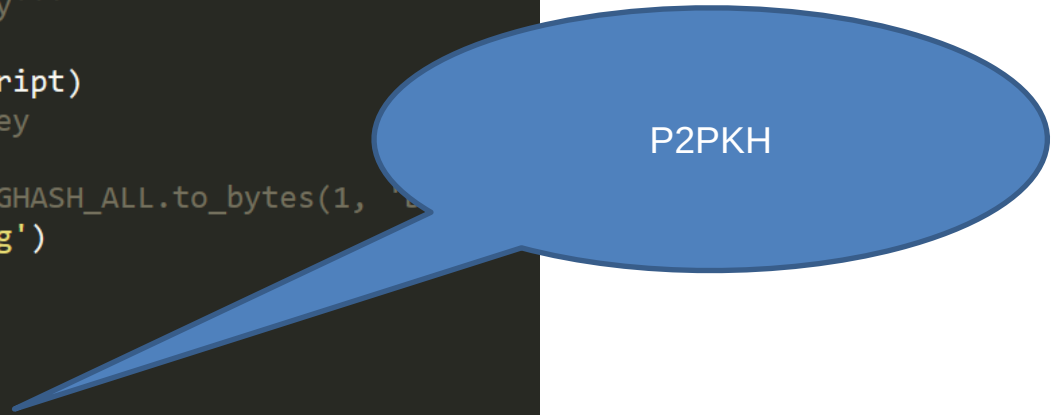
Gastando un P2SH output

```
# Define the input:
tx_hash = '2510f721161210c19abb6f45848f29e645641cb9d60b5e9df6ee20f82c591305'
tx_index = 0

#10000:OP_HASH160 cab93154ada73a646bb01efc393ad5dd226d43e8

newInput = TxIn(bytes.fromhex(tx_hash),tx_index)
# Input defined

# Since it's a p2pkh address we generate the script for this
ScriptPubkey = p2pkh_script_from_address('n3jKhCmVjvaVgg8C5P7                QAAvf7Wc')
# input value = 10000 output 5000
newOutput = TxOut(5000,ScriptPubkey)

# Define a new testnet transaction
newTx = Tx(1,[newInput],[newOutput],0,True)
# Still unsigned
#print(newTx.serialize().hex())

input_index = 0
newTx.sign_input(input_index,newPrivKey,redeemScript)

to_spend = newTx.serialize().hex()
```



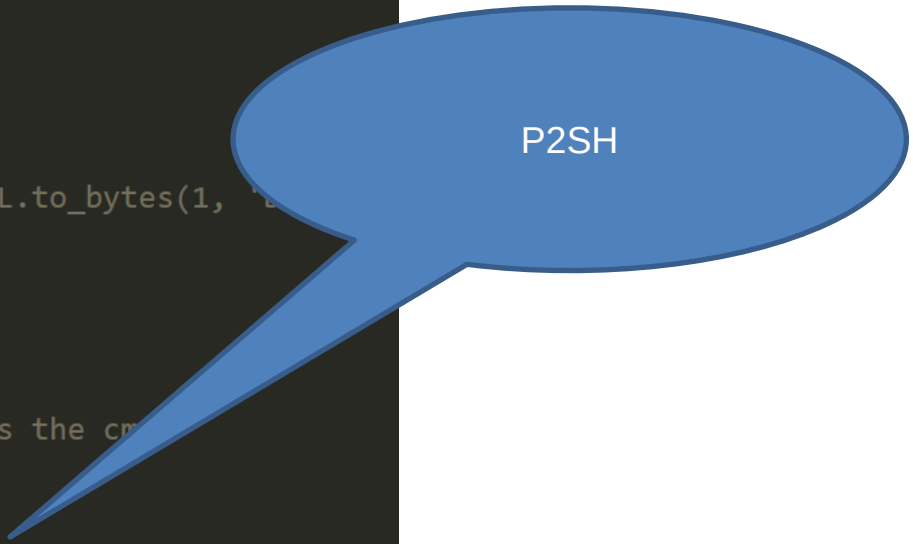
Gastando un P2SH output

```
def sign_input(self, input_index, private_key, redeem_script=None):
    '''Signs the input using the private key'''
    # get the signature hash (z)
    z = self.sig_hash(input_index,redeem_script)
    # get der signature of z from private key
    der = private_key.sign(z).der()
    # append the SIGHASH_ALL to der (use SIGHASH_ALL.to_bytes(1, '
    sig = der + SIGHASH_ALL.to_bytes(1, 'big')
    # calculate the sec
    sec = private_key.point.sec()
    # Handle p2pkh first
    if redeem_script == None:
        # initialize a new script with [sig, sec] as the cmds
        script_sig = Script([sig, sec])
    # Else we are dealing with a p2sh
    else:
        script_sig = Script([sig, sec, redeem_script.raw_serialize()])
    # change input's script_sig to new script
    self.tx_ins[input_index].script_sig = script_sig
    # return whether sig is valid using self.verify_input
    return self.verify_input(input_index,redeem_script)
```



Gastando un P2SH output

```
def sign_input(self, input_index, private_key, redeem_script=None):
    '''Signs the input using the private key'''
    # get the signature hash (z)
    z = self.sig_hash(input_index,redeem_script)
    # get der signature of z from private key
    der = private_key.sign(z).der()
    # append the SIGHASH_ALL to der (use SIGHASH_ALL.to_bytes(1, '
    sig = der + SIGHASH_ALL.to_bytes(1, 'big')
    # calculate the sec
    sec = private_key.point.sec()
    # Handle p2pkh first
    if redeem_script == None:
        # initialize a new script with [sig, sec] as the cm
        script_sig = Script([sig, sec])
    # Else we are dealing with a p2sh
    else:
        script_sig = Script([sig, sec, redeem_script.raw_serialize()])
    # change input's script_sig to new script
    self.tx_ins[input_index].script_sig = script_sig
    # return whether sig is valid using self.verify_input
    return self.verify_input(input_index,redeem_script)
```



Gastando un P2SH output

```
def sign_input(self, input_index, private_key, redeem_script=None):
    '''Signs the input using the private key'''
    # get the signature hash (z)
    z = self.sig_hash(input_index,redeem_script)
    # get der signature of z from private key
    der = private_key.sign(z).der()
    # append the SIGHASH_ALL to der (use SIGHASH_ALL.to_bytes(1, '
    sig = der + SIGHASH_ALL.to_bytes(1, 'big')
    # calculate the sec
    sec = private_key.point.sec()
    # Handle p2pkh first
    if redeem_script == None:
        # initialize a new script with [sig, sec] as the cmds
        script_sig = Script([sig, sec])
    # Else we are dealing with a p2sh
    else:
        script_sig = Script([sig, sec, redeem_script.raw_serialize()])
    # change input's script_sig to new script
    self.tx_ins[input_index].script_sig = script_sig
    # return whether sig is valid using self.verify_input
    return self.verify_input(input_index,redeem_script)
```

Ojo: para P2SH paso el redeem_script

Gastando un P2SH output

```
def sig_hash(self, input_index, redeem_script=None):
    '''Returns the integer representation of the hash that needs to get
    signed for index input_index'''
    # redeem_script is used in p2sh transaction to replace ScriptSig
    # if input_index is not in tx_ins, then all ScriptSigs will be empty!!!
    # start the serialization with version
    # use int_to_little_endian in 4 bytes
    s = int_to_little_endian(self.version, 4)
    # add how many inputs there are using encode_varint
    s += encode_varint(len(self.tx_ins))
    # loop through each input using enumerate, so we have the input index
    for i, tx_in in enumerate(self.tx_ins):
        # if the input index is the one we're signing
        if i == input_index:
            # if the RedeemScript was passed in, that's the ScriptSig
            if redeem_script:
                script_sig = redeem_script
            # otherwise the previous tx's ScriptPubkey is the ScriptSig
            else:
                script_sig = tx_in.script_pubkey(self.testnet)
        # Otherwise, the ScriptSig is empty
        else:
            script_sig = None
        # add the serialization of the input with the ScriptSig we want

        ...

        s += TxIn(
```

Para P2SH ScriptSig se reemplaza por el redeem_script

Importante

P2SH

Los métodos que tenemos en tx.py nos permiten:

- Crear un P2Sh spend

Los métodos que tenemos en tx.py ****no**** nos permiten:

- Validar un P2Sh script (correctamente)
- Si les molesta esto lo pueden implementar!!!

Referencias

- Jimmy Song, Programming Bitcoin, capítulos 5,6,7,8
- https://en.bitcoin.it/wiki/Script
- https://en.bitcoin.it/wiki/OP_CHECKSIG
- https://bitcoin.stackexchange.com/questions/3374/how-to-redeem-a-basic-tx