Script

Implementation

What are we missing?

We already implemented Tx

To make transactions we still need to:

- Create/serialize scripts in Script!!!

To validate transactions we need to:

- Run the scripts written in Script!!!

Now we will explain how to do this

Scripts

Commands

Two types:

1. OP_CODES
2. Bytes (data)

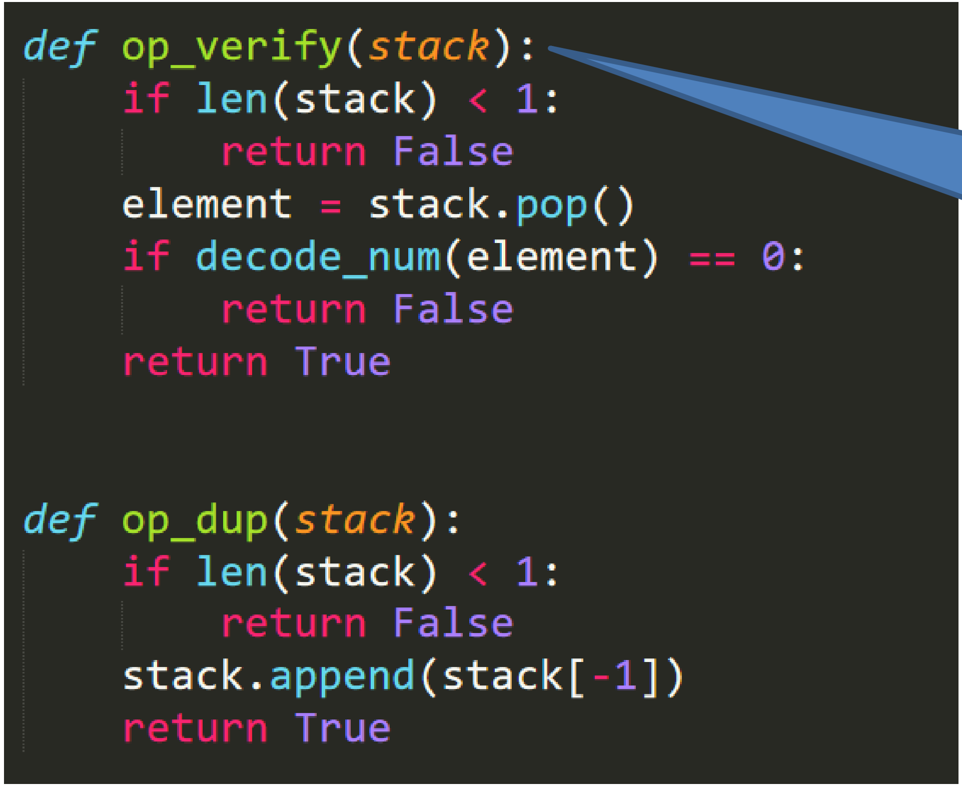
Script

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

Implementation of Op codes

OP_DUP, OP_VERIFY

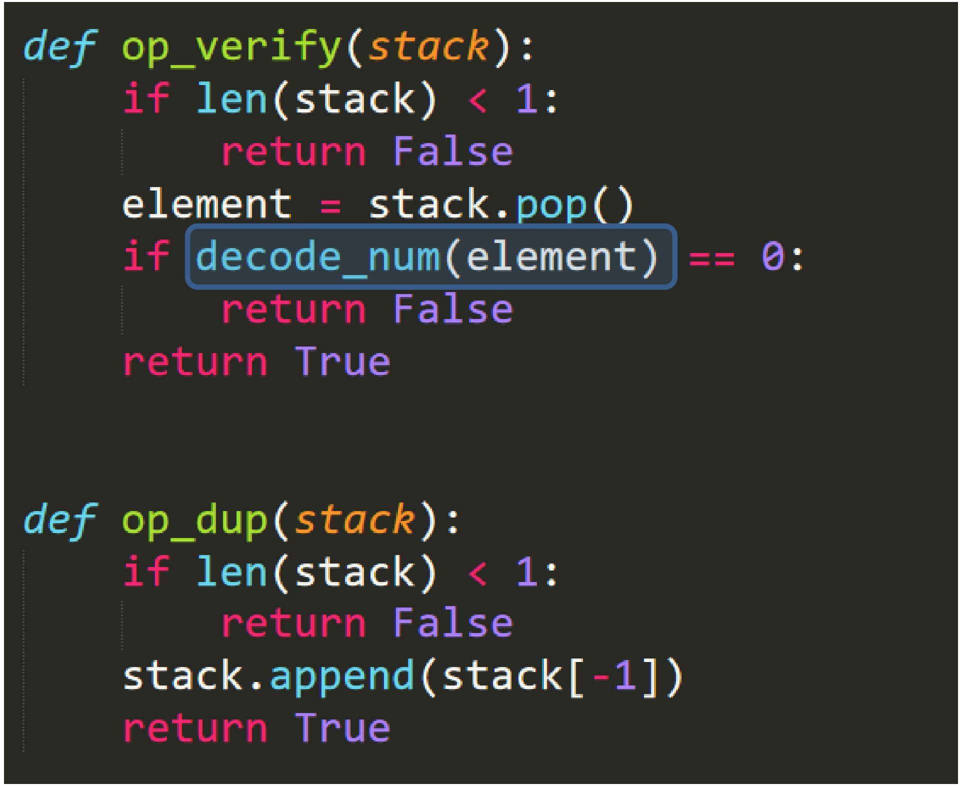


```
def op_verify(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    if decode_num(element) == 0:
        return False
    return True


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

Always manipulate the stack!!!

Implementation of Op codes

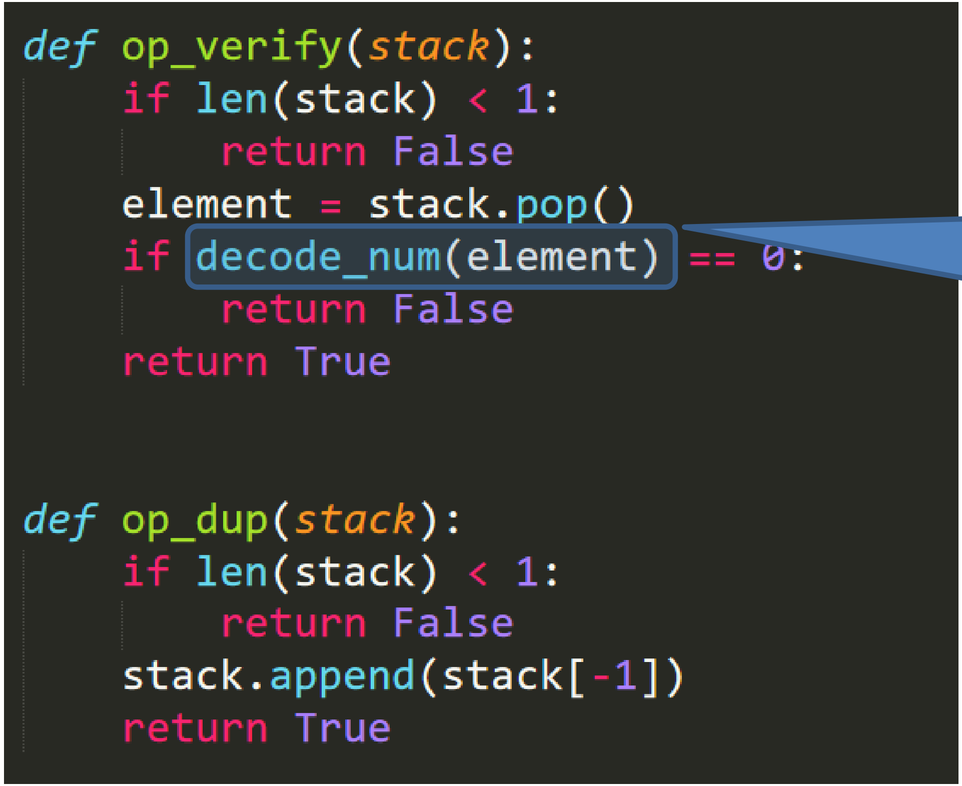
OP_DUP, OP_VERIFY



```
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
```

Implementation of Op codes

OP_DUP, OP_VERIFY

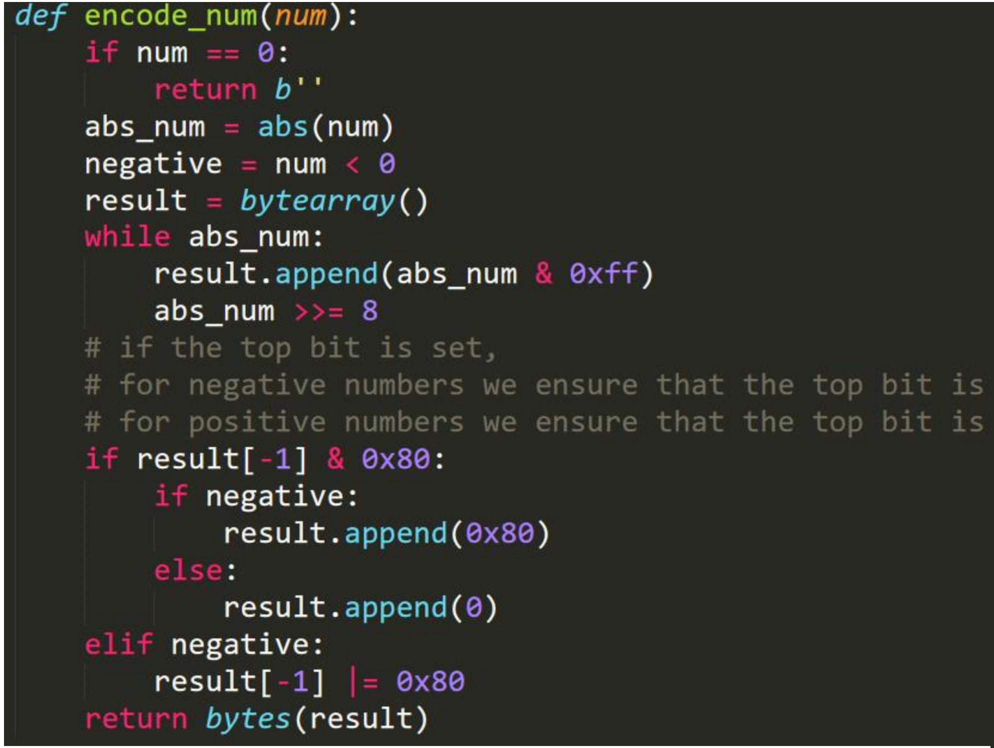


```python
def op_verify(stack):
    if len(stack) < 1:
        return False
    element = stack.pop()
    if decode_num(element) == 0:
        return False
    return True


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

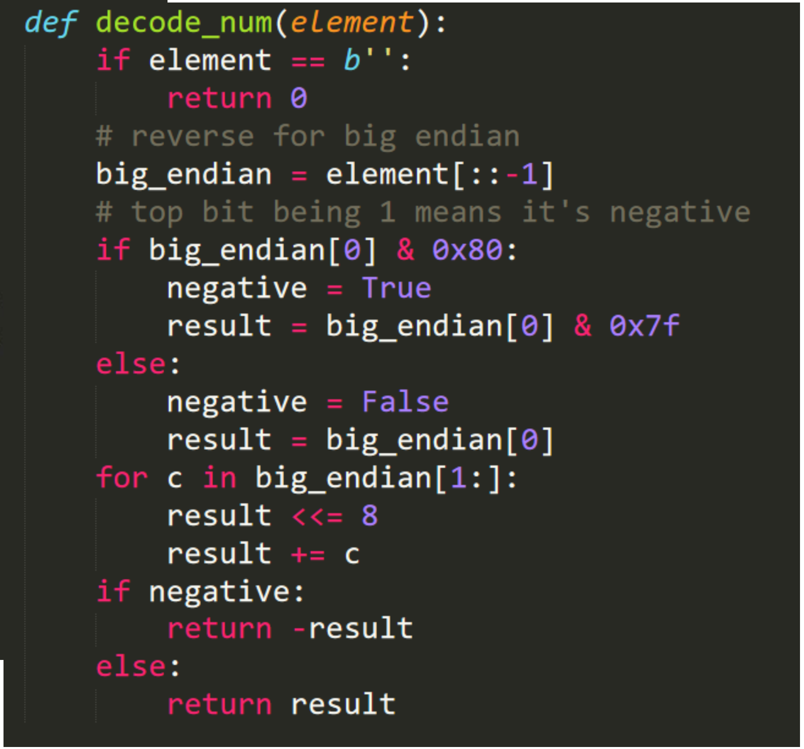
Numbers are a bit strange in Script!

Implementation of Op codes

Numbers in Script



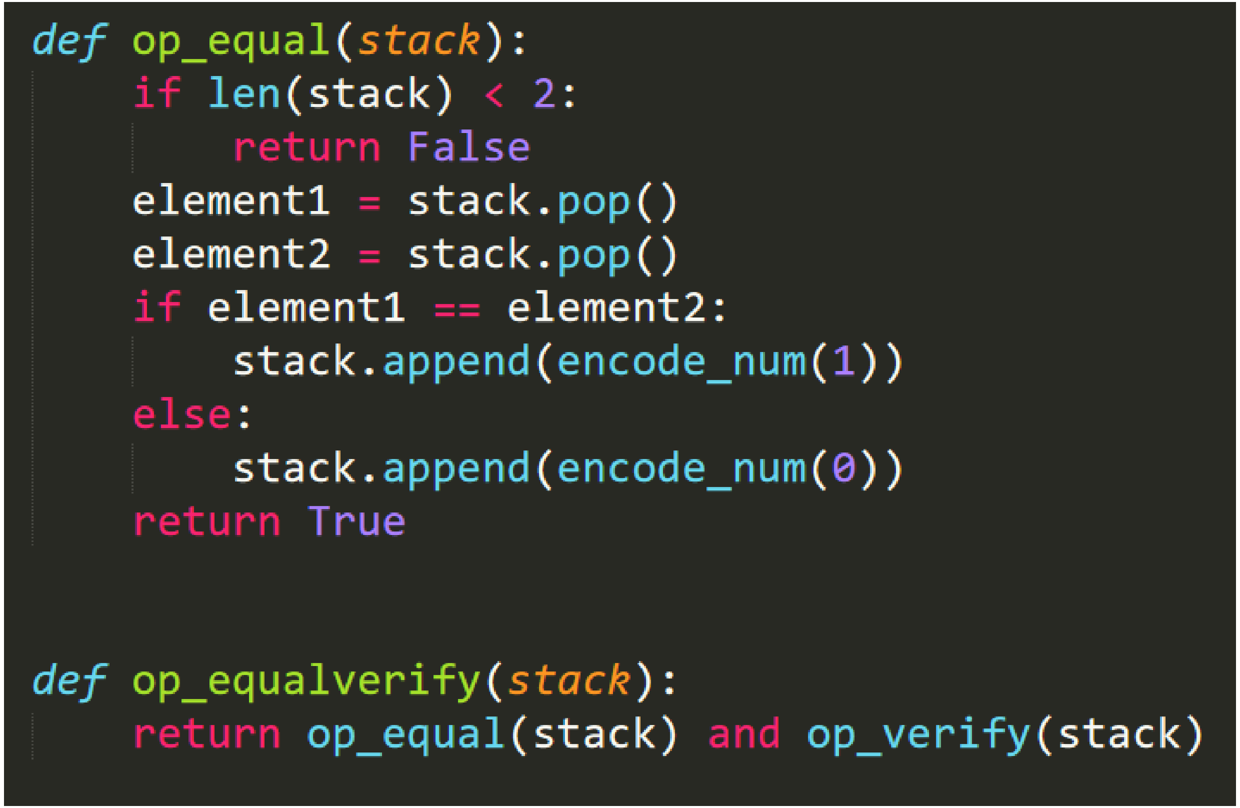
```python
def encode_num(num):
    if num == 0:
        return b''
    abs_num = abs(num)
    negative = num < 0
    result = bytearray()
    while abs_num:
        result.append(abs_num & 0xff)
        abs_num >>= 8
    # if the top bit is set,
    # for negative numbers we ensure that the top bit is
    # 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
```

Implementation of Op codes

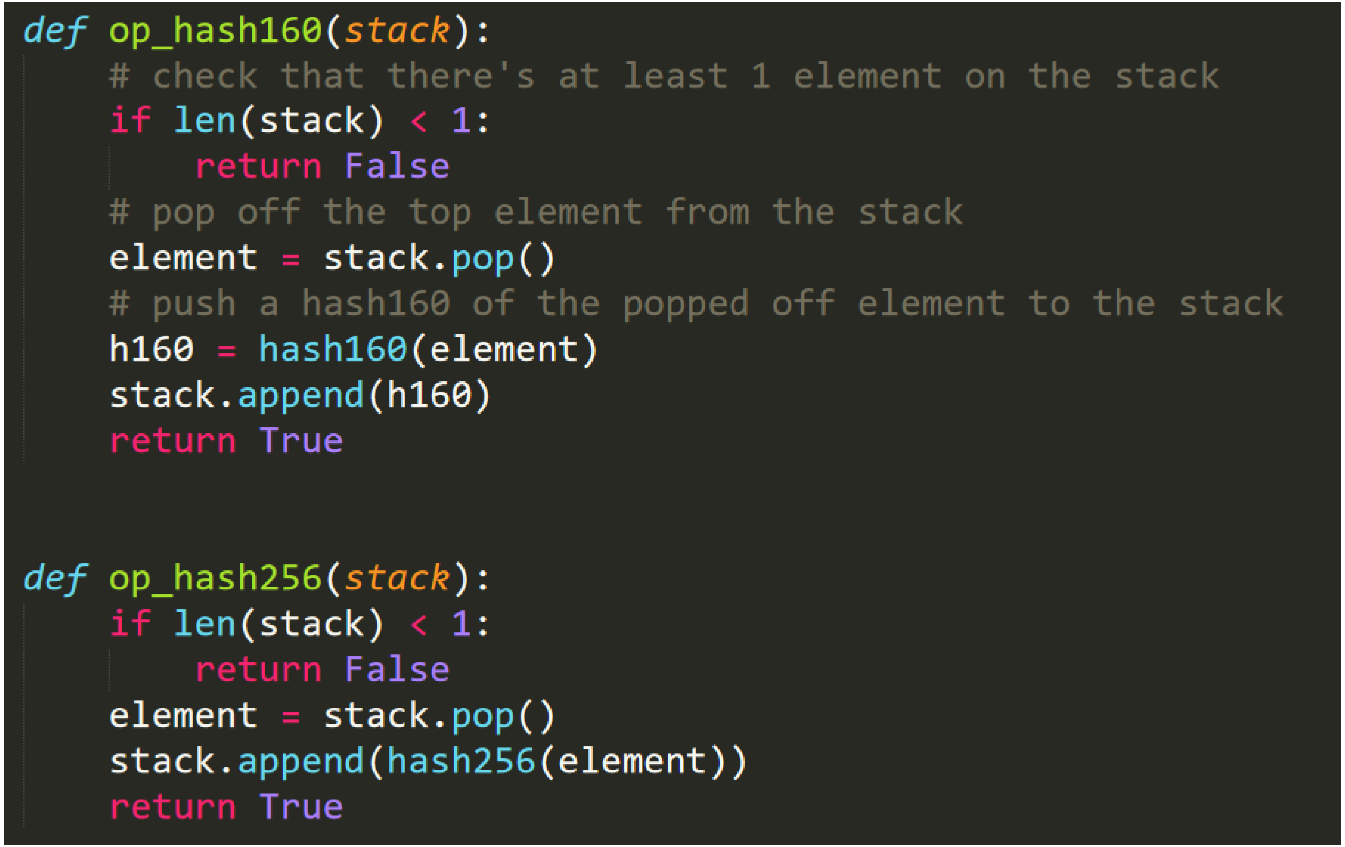
OP_EQUAL



```python
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)
```

Implementation of Op codes

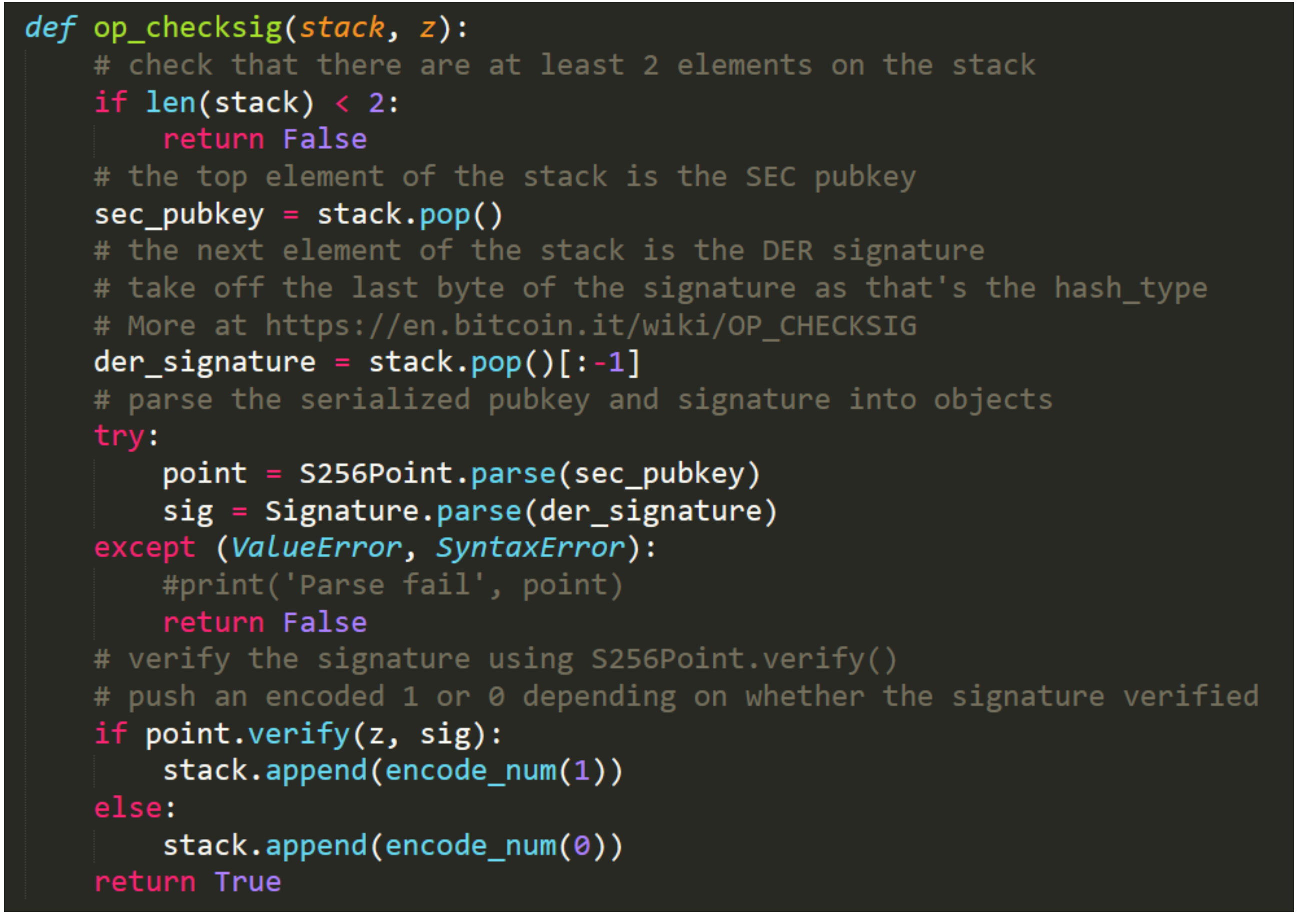
OP_HASH



```python
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
```

Implementation of Op codes

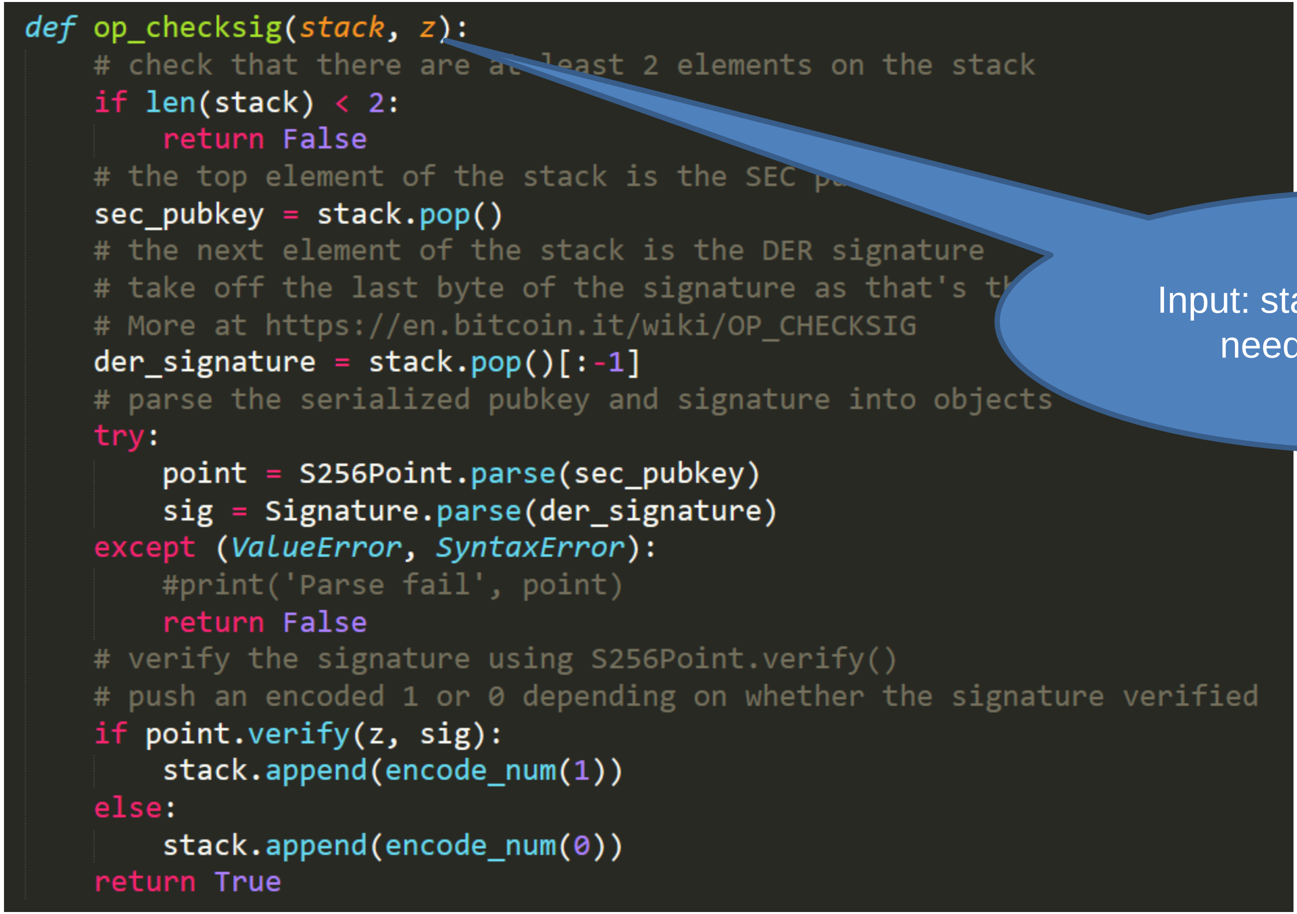
OP_CHECKSIG



```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
```

Implementation of Op codes

OP_CHECKSIG

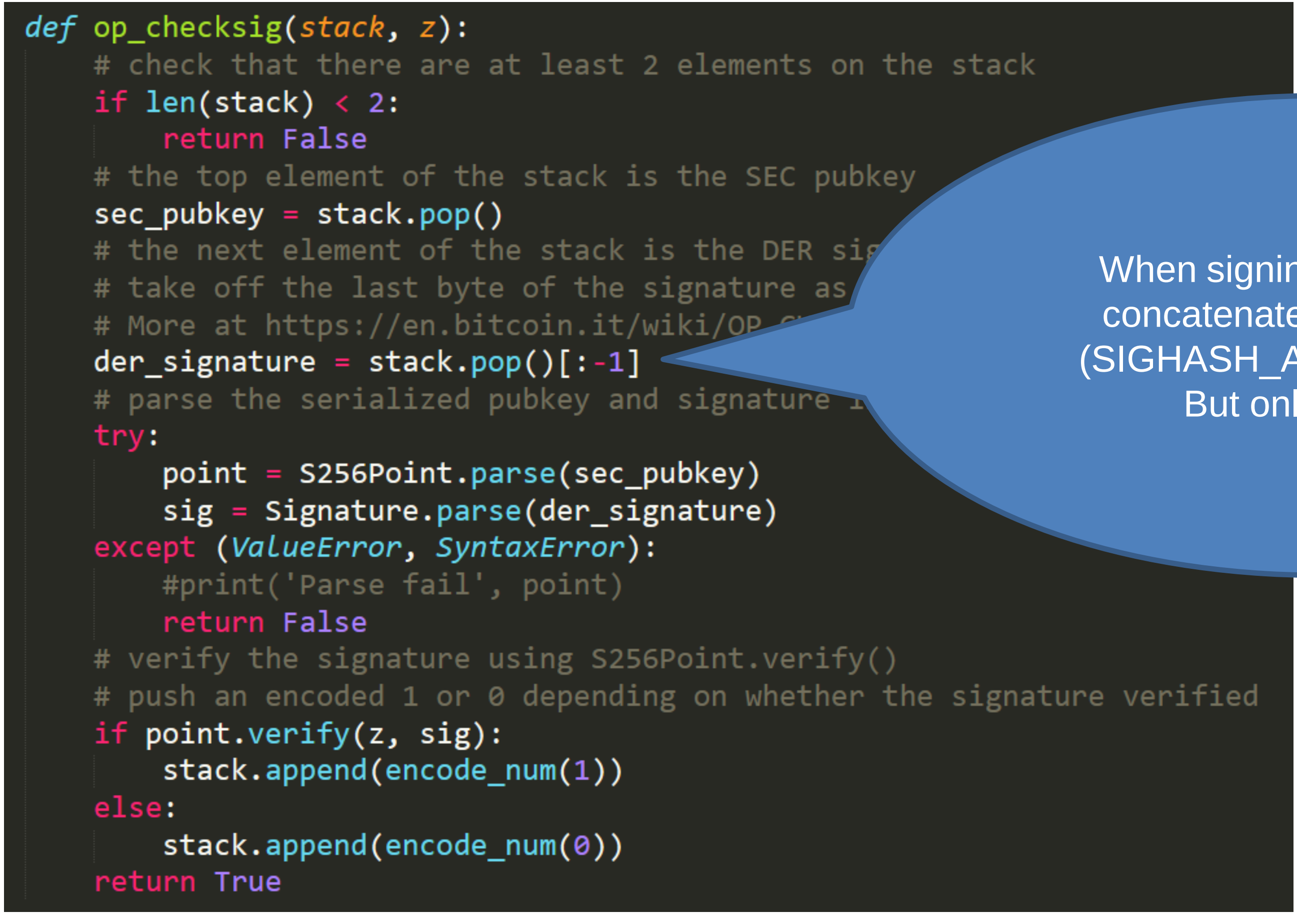


```
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pu
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER signature
    # take off the last byte of 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, and a hash we need to sign (as int)

Implementation of Op codes

OP_CHECKSIG

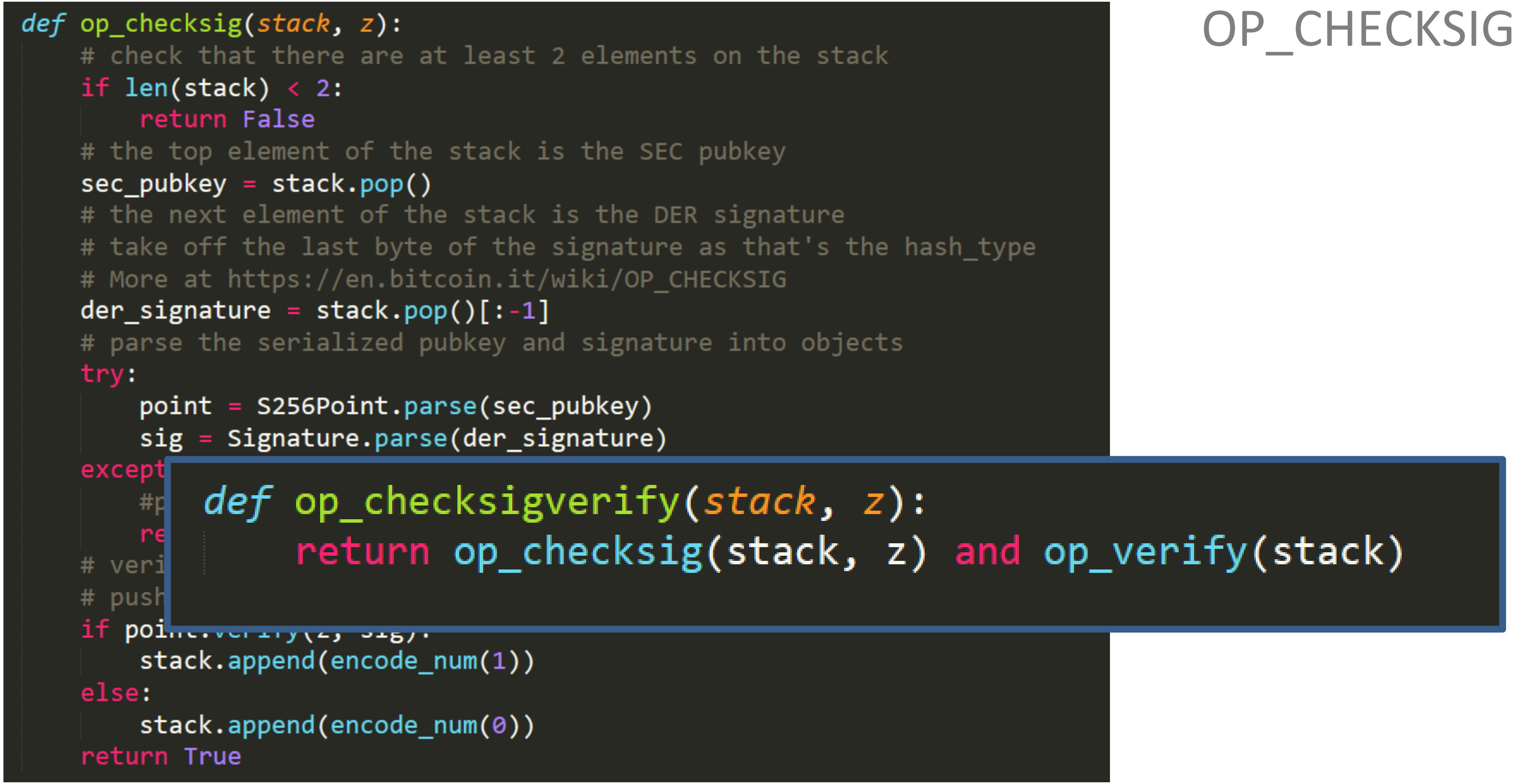


```python
def op_checksig(stack, z):
    # check that there are at least 2 elements on the stack
    if len(stack) < 2:
        return False
    # the top element of the stack is the SEC pubkey
    sec_pubkey = stack.pop()
    # the next element of the stack is the DER sig
    # take off the last byte 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
```

When signing in Bitcoin we concatenate the hashType (SIGHASH_ALL) at the end!!! But only 1 byte!!!

Implementation of 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 point.verify(z, sig):
        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

Commands

Two types:

1. OP_CODES
2. Bytes (data)

How to differentiate which is which?

Scripts

Commands

Script = sequence of bytes (when we receive it)

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

Scripts

Commands

Script = sequence of bytes (when we receive it)

0x00 = OP_0

0x51 = OP_1

…

0x60 = OP_16

0x76 = OP_DUP

0xa9 = OP_HASH160

0xac = OP_CHECKSIG

Read the docs:
https://en.bitcoin.it/wiki/Script

Scripts

mands

A jump between 0 and 1!!!
This is were the data goes

Script = sequence of bytes (w

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

Read the docs:
https://en.bitcoin.it/wiki/Script

Scripts

Data

Script = sequence of bytes (when we receive it)

If I read a byte *n* between 0x01 and 0x4b, the next *n* bytes are data

- For instance, a DER signature
- A SEC public key
- A redeem script (for P2SH)

Scripts

Data

Script = sequence of bytes (when we receive it)

If I read a byte *n* between 0x01 and 0x4b, the next *n* bytes are data

- For instance, a DER signature – 70 bytes
- A SEC public key – 33/65 bytes
- A redeem script (for P2SH) -- ???

Scripts

Data

Script = sequence of bytes (when we receive it)

If I read a byte *n* between 0x01 and 0x4b, the next *n* bytes are data

- For instance, a DER signature – 70 bytes
- A SEC public key – 33/65 bytes
- A redeem script (for P2SH)

OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG

Small (24 bytes)

Scripts

Data

Script = sequence of bytes (when we receive it)

If I read a byte *n* between 0x01 and 0x4b, the next *n* bytes are data

- For instance, a DER signature – 70 bytes
- A SEC public key – 33/65 bytes
- A redeem script (for P2SH)

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

Scripts

Data

n = 1 to 75 bytes!!!

Script = sequence of bytes (wher e receive it)

If I read a byte *n* between 0x01 and 0x4b, the next *n* bytes are data

- For instance, a DER signature – 70 bytes
- A SEC public key – 33/65 bytes
- A redeem script (for P2SH)

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

Scripts

Data

n = 1 to 75 bytes!!!

Script = sequence of bytes (wher e receive it)

If I read a byte *n* between 0x01 and 0x4b, the are data

- For instance, a DER signature – 70 by
- A SEC public key – 33/65 bytes
- A redeem script (for P2SH)

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

➢ 75 bytes

What now?

Scripts

Data

Script = sequence of bytes (when we receive it)

If I read 0x4c = 76 = OP_PUSHDATA1 -- next byte = length of data
0x4c 0xff <data of length 255 bytes> – from 75 to 255 bytes

If I read 0x4d = 77 = OP_PUSHDATA2 -- next 2 bytes = length

0x4d b1 b2 <data of length 520 bytes> – from 256 to 520 bytes

Scripts

Data

Script = sequenc

Is it not 65535?

NO: Max size on the network is 520 bytes!!!

If I read 0x4c = 76 = OP_ ength of data

0x4c 0xff <data of length 255 bytes> – fr to 255 bytes

If I read 0x4d = 77 = OP_PUSHDATA2 -- next 2 tes = length

0x4d b1 b2 <data of length 520 bytes> – from 256 to 520 bytes

Scripts

Data

Script = sequence of bytes (when we receive it)

0x4c = 76 = OP_PUSHDATA1: from 75 to 255 bytes

0x4d = 77 = OP_PUSHDATA2: from 256 to 520 bytes

0x4e = OP_PUSHDATA4: length 4 bytes; not currently in use

Scripts

Commands

Two types:

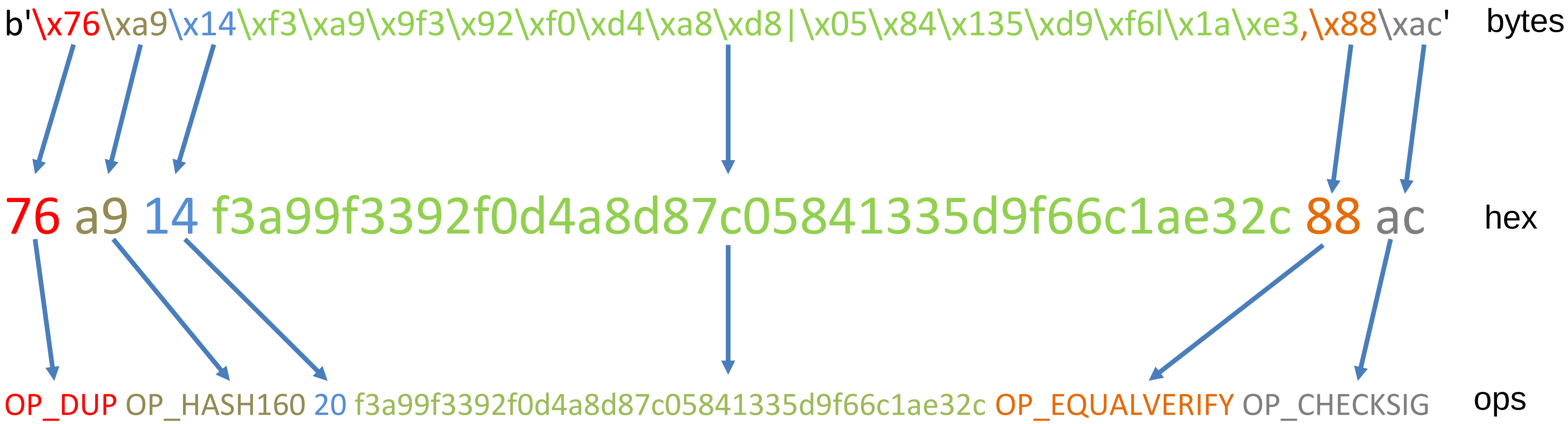
1. OP_CODES
2. Bytes (data)

What now?

- Parse (bytes)
- Serialize (to bytes)

Scripts

Parse



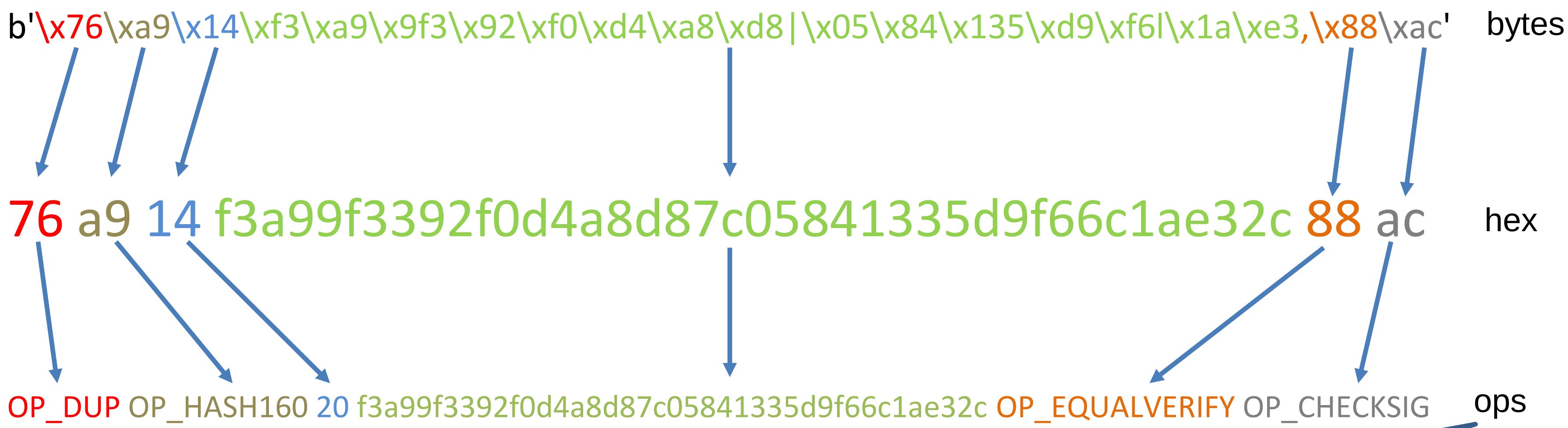
b'\x76\xa9\x14\xf3\xa9\x9f3\x92\xf0\xd4\xa8\xd8|\x05\x84\x135\xd9\xf6l\x1a\xe3,\x88\xac' bytes

76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG ops

Scripts

Parse



b'\x76\xa9\x14\xf3\xa9\x9f3\x92\xf0\xd4\xa8\xd8|\x05\x84\x135\xd9\xf6l\x1a\xe3,\x88\xac' bytes

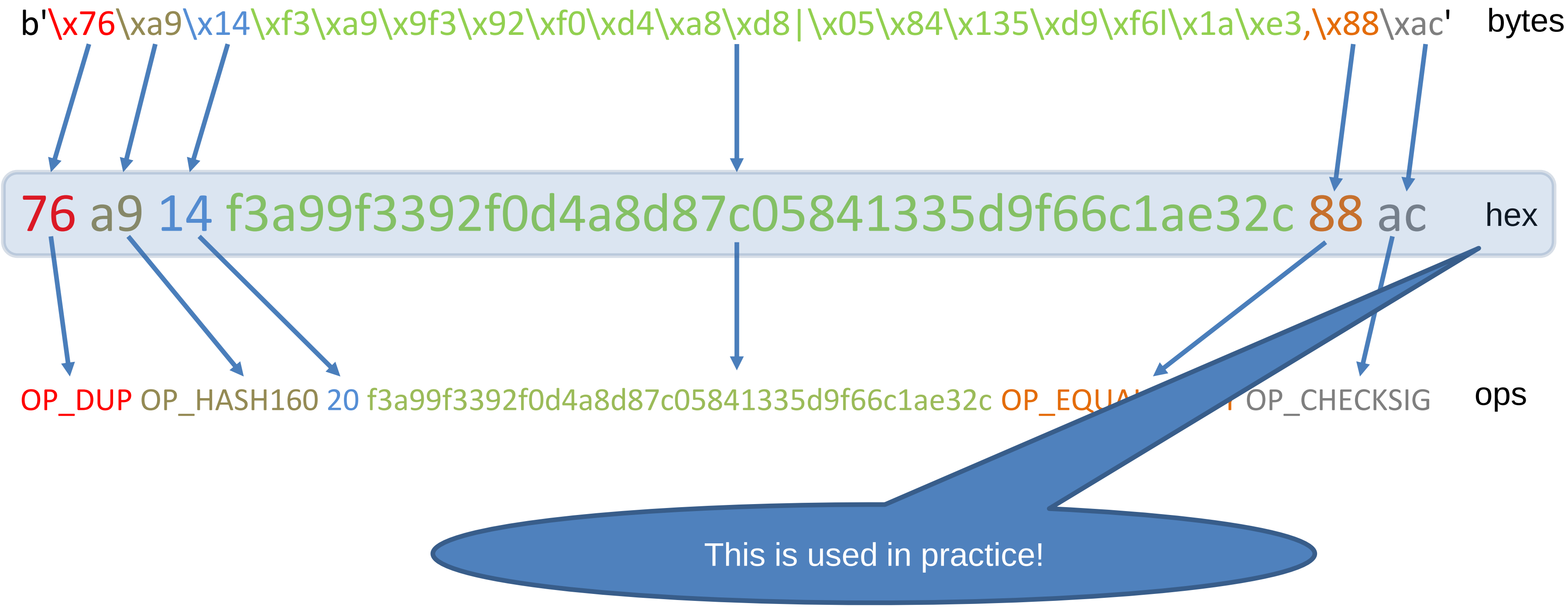
76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG ops

Only for visualization!

Scripts

Parse



b'\x76\xa9\x14\xf3\xa9\x9f3\x92\xf0\xd4\xa8\xd8|\x05\x84\x135\xd9\xf6l\x1a\xe3,\x88\xac' bytes

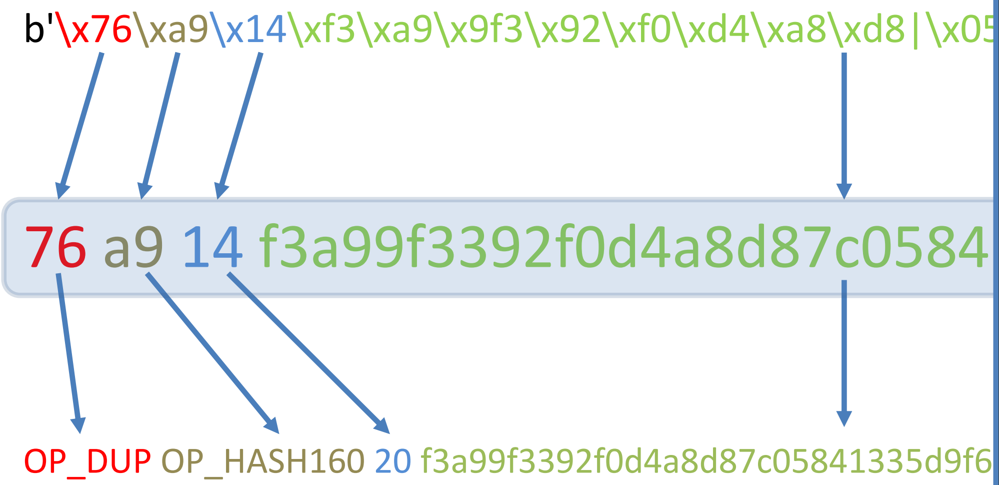
76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUA OP_CHECKSIG ops

This is used in practice!

Scripts

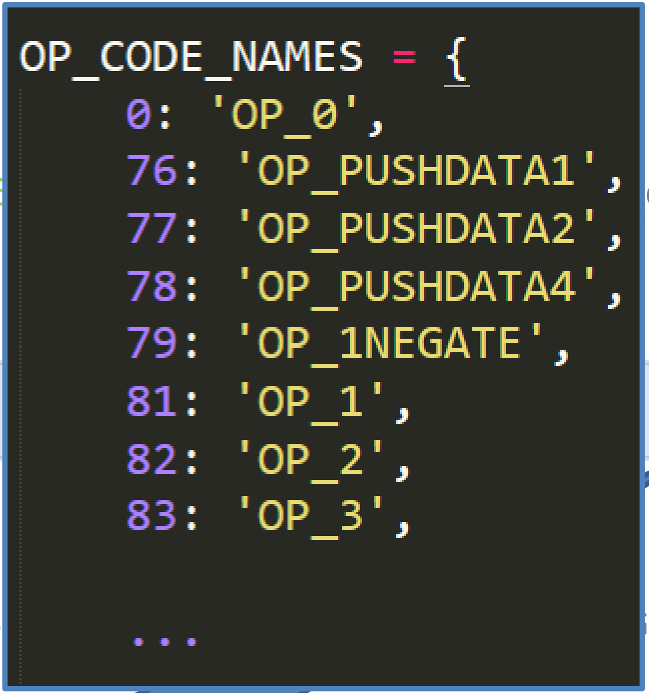
Parse



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',

    ...
```

This is used in practice!

Scripts

Parse

Serializations starts with the length of the script (varint)

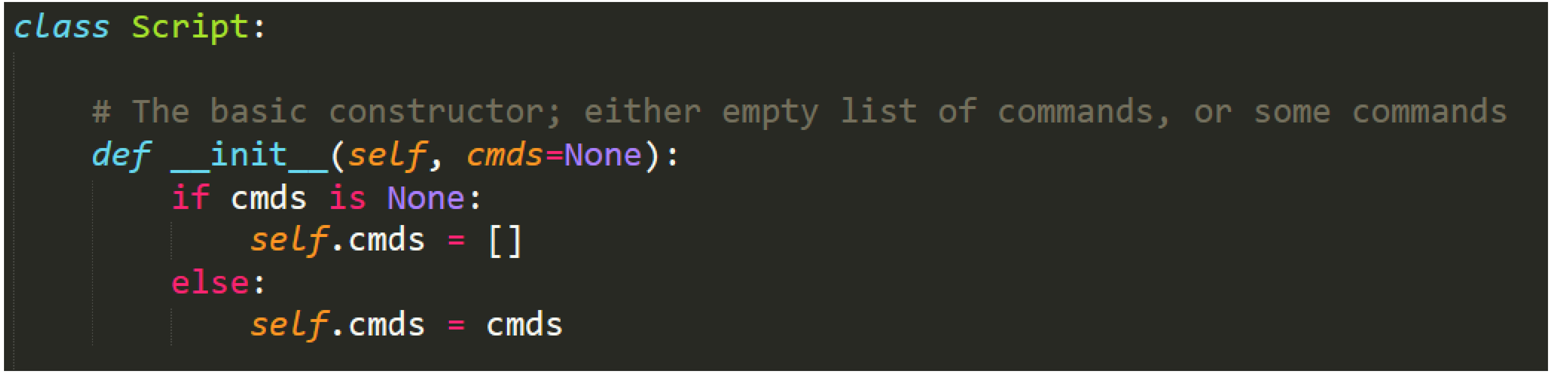
19 76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac

OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG

25 (hex)

Scripts

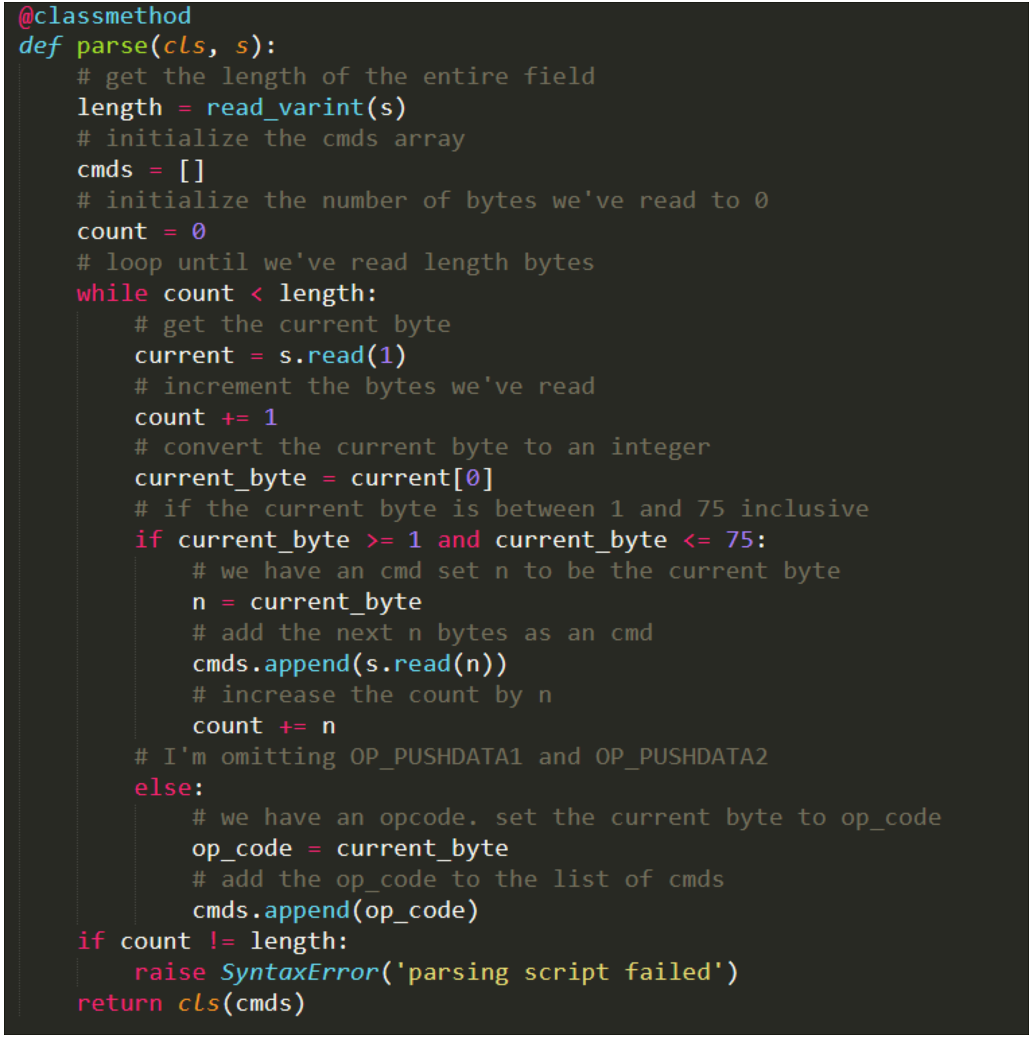
Parse



```python
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

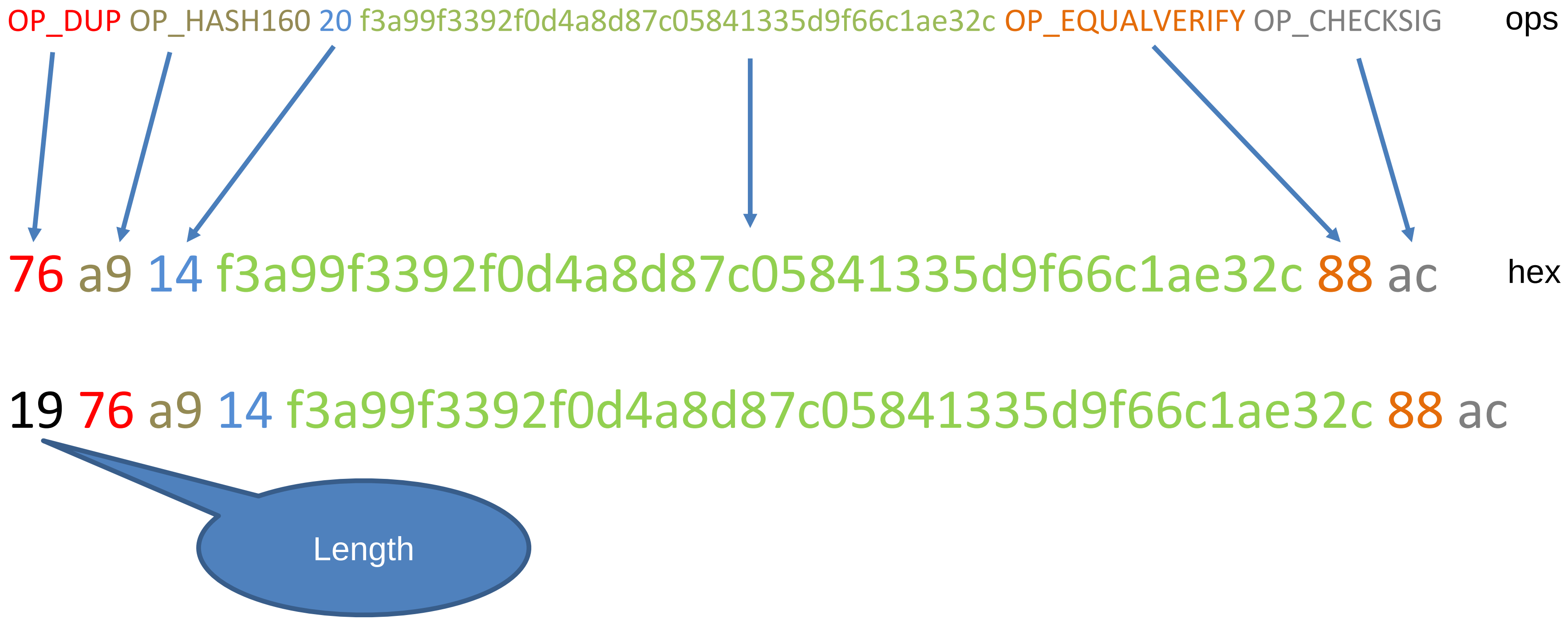
Parse



```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



OP_DUP OP_HASH160 20 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG ops

76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

19 76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac

Length

Scripts

Serialize

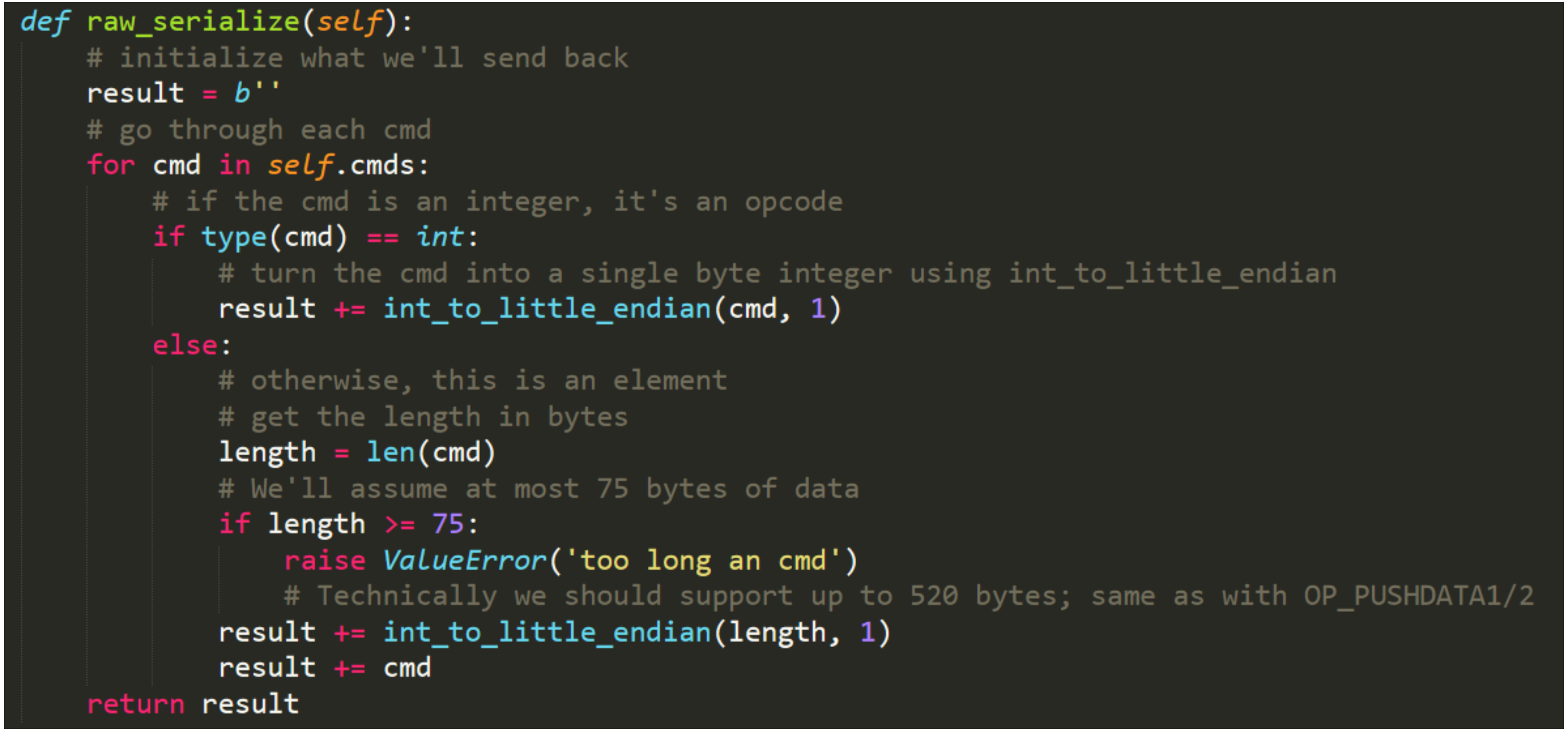
76 a9 14 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c 88 ac hex

\x19 \x76 \xa9 \x14 f3a99f3392... bytes

In reality!

Scripts

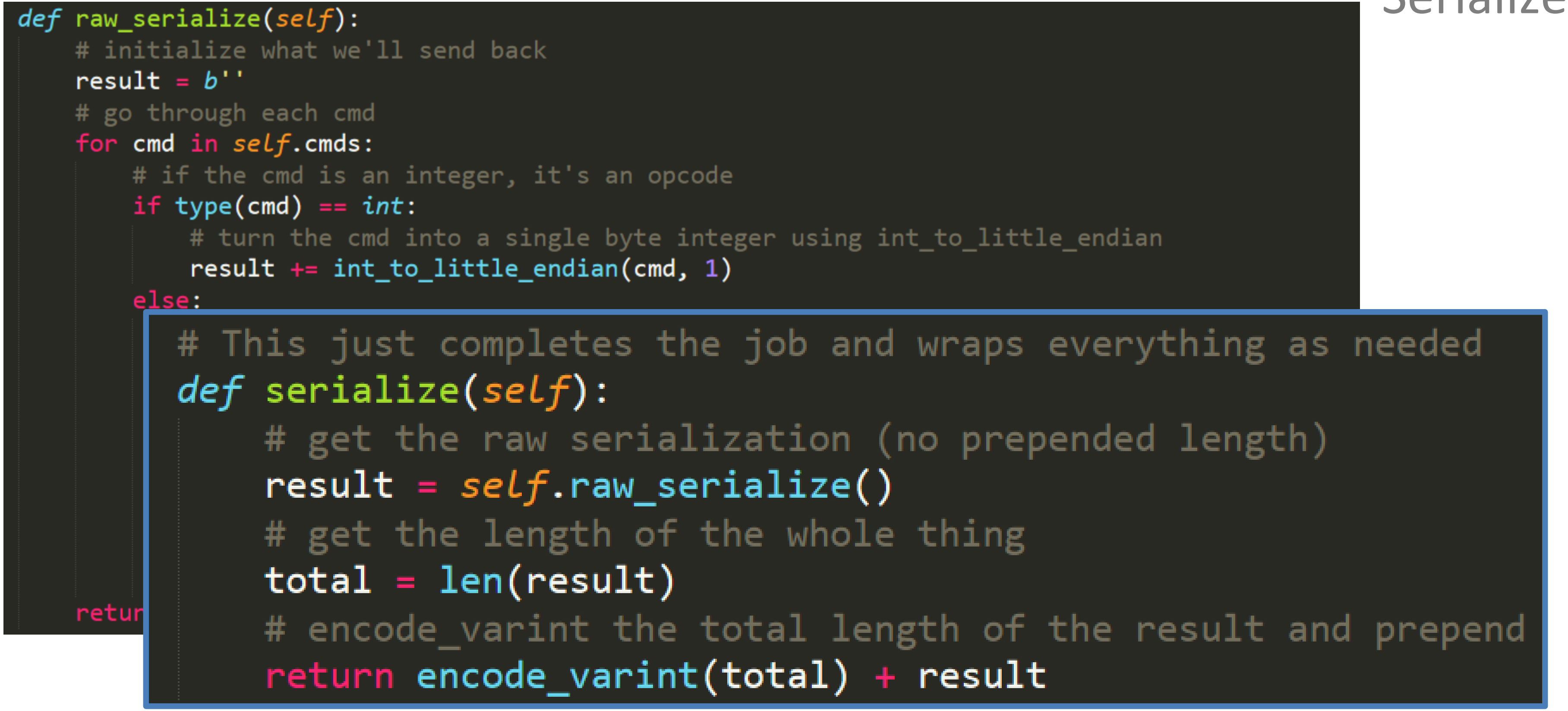
Serialize



```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

What are we missing?

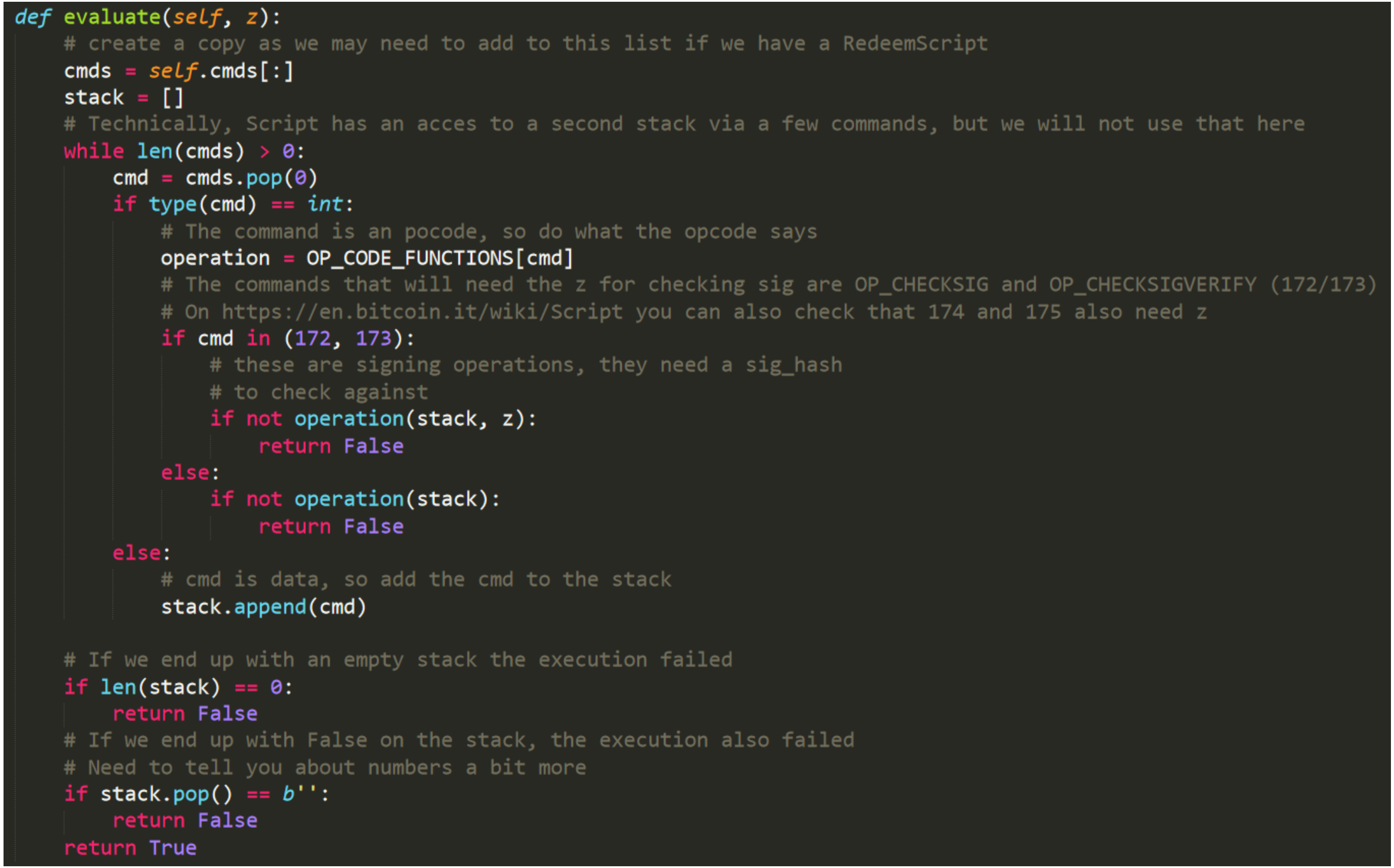
- We need to execute Script

How to do this?

- Command by command using the stack

Scripts

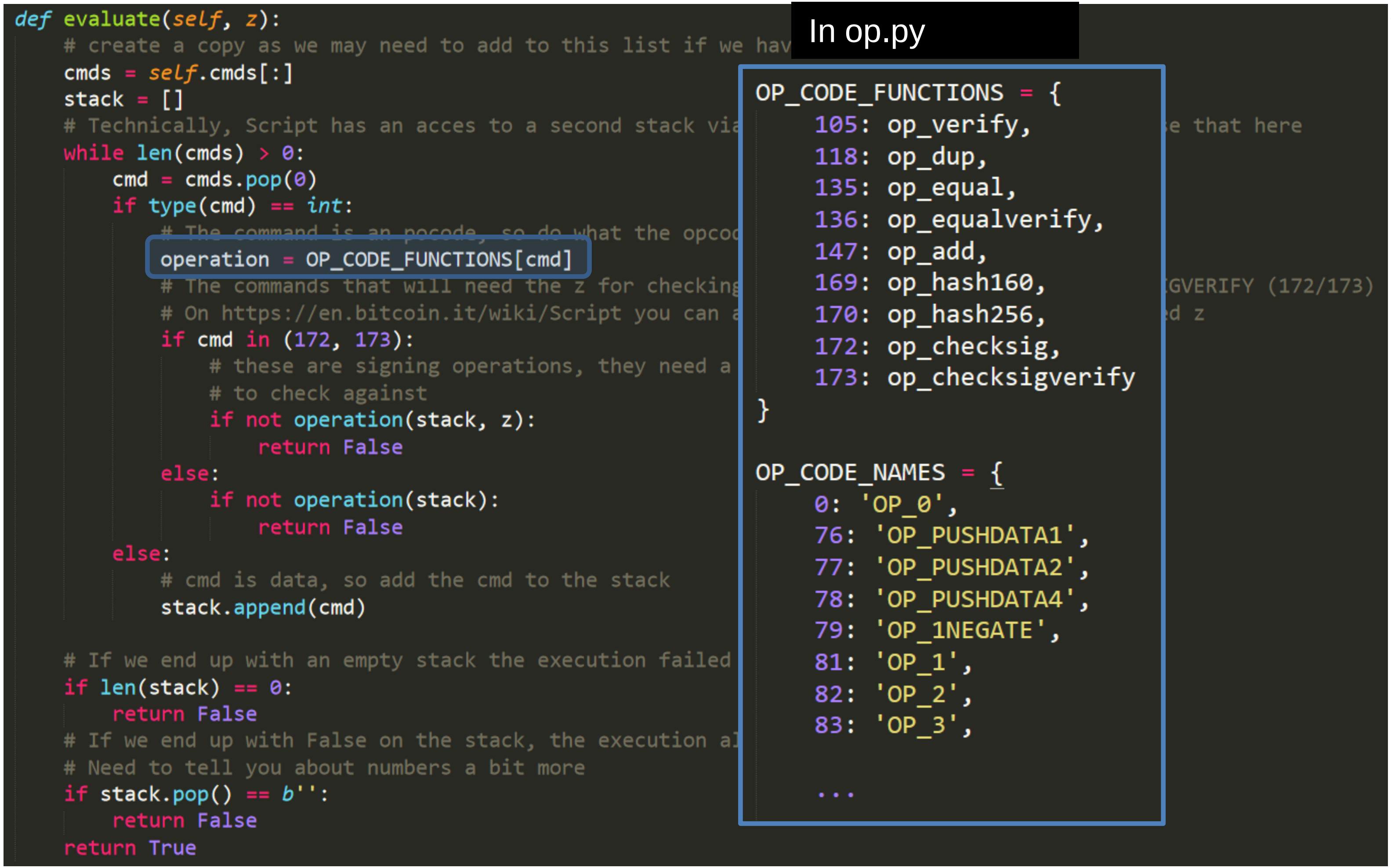
Evaluate



```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



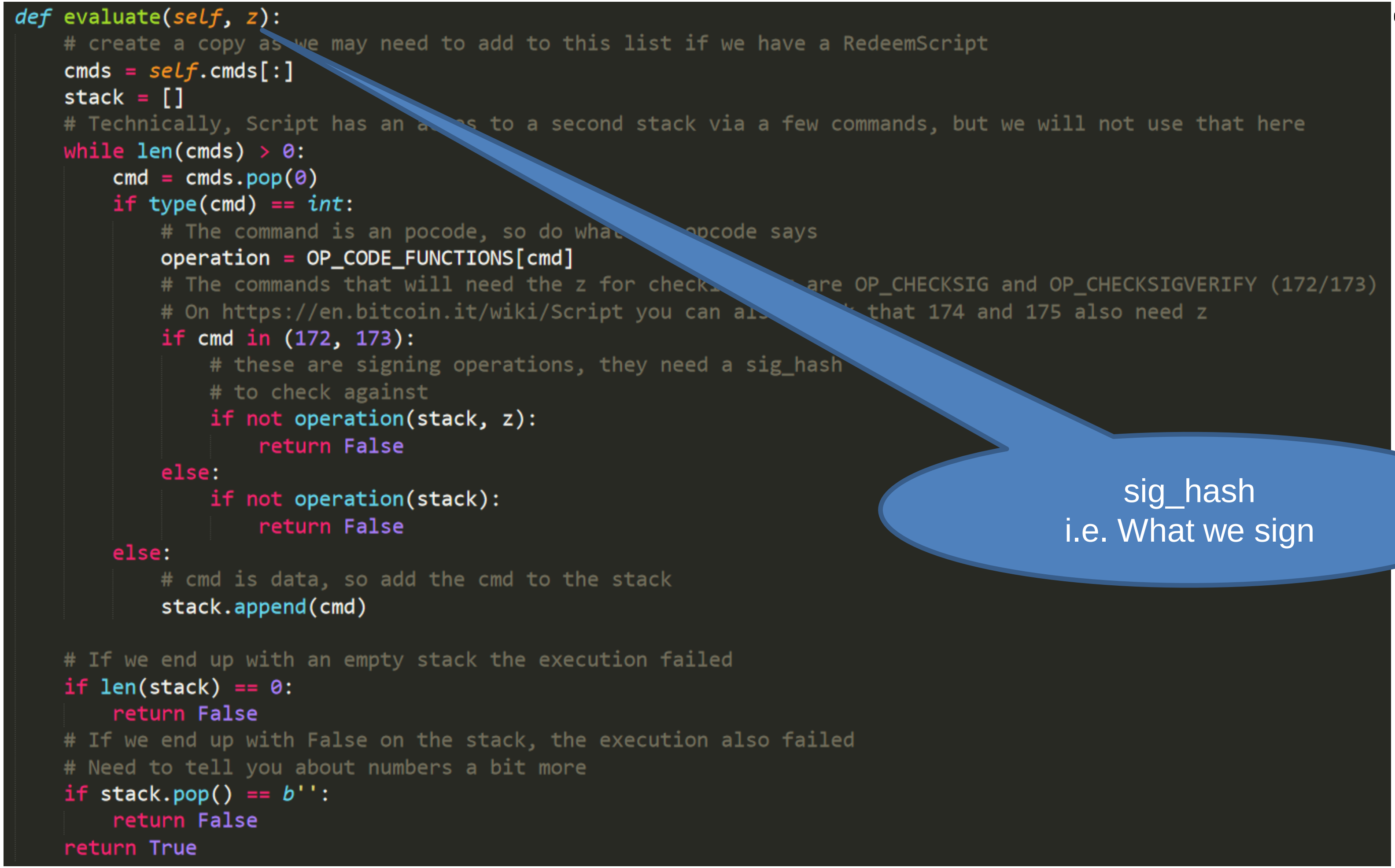
```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we hav
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via                    e that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # 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

```
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

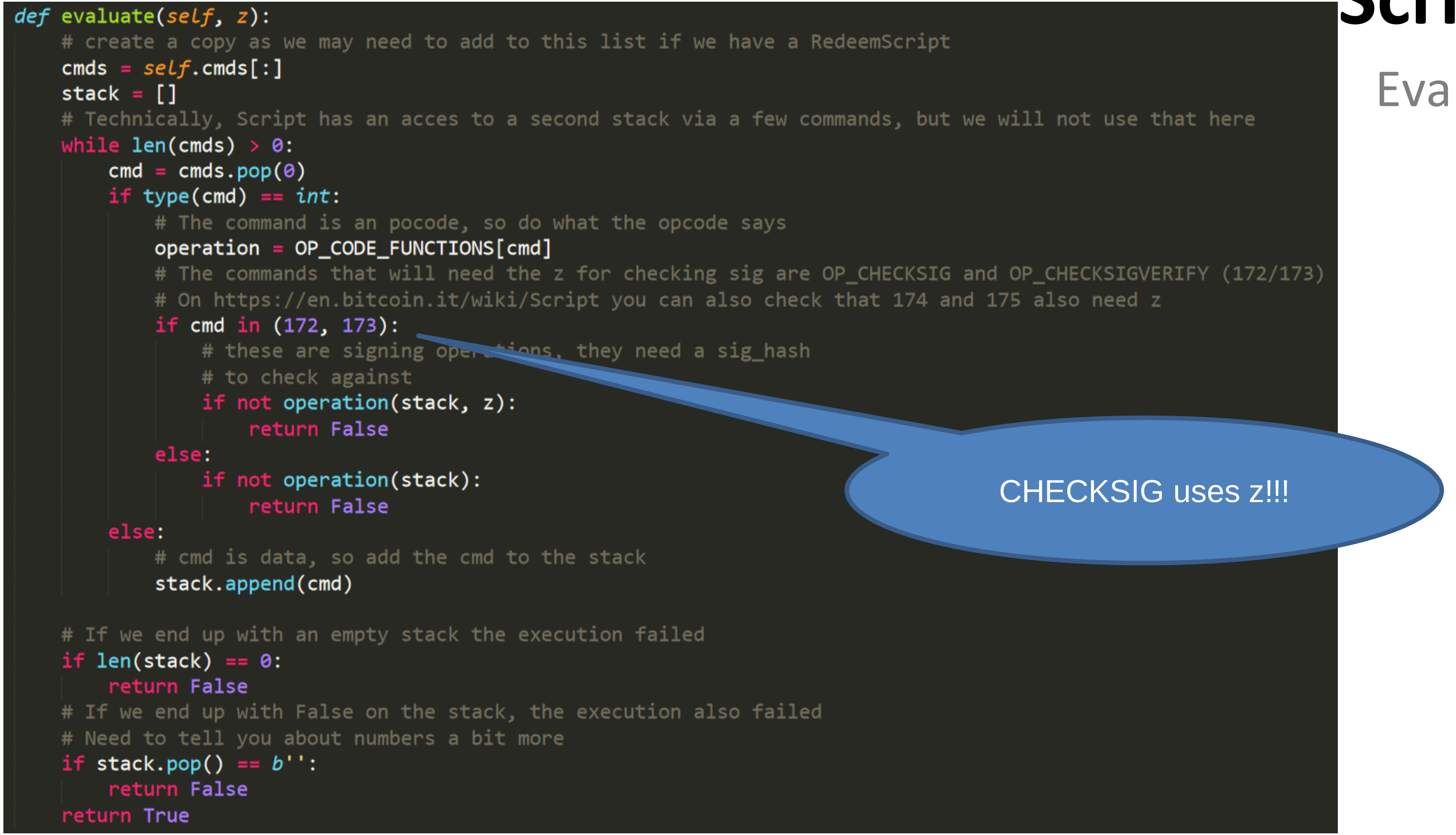


```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an a   s to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode, so do what     opcode says
            operation = OP_CODE_FUNCTIONS[cmd]
            # The commands that will need the z for check         are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you can al       k 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. What we sign

Scripts

Evaluate

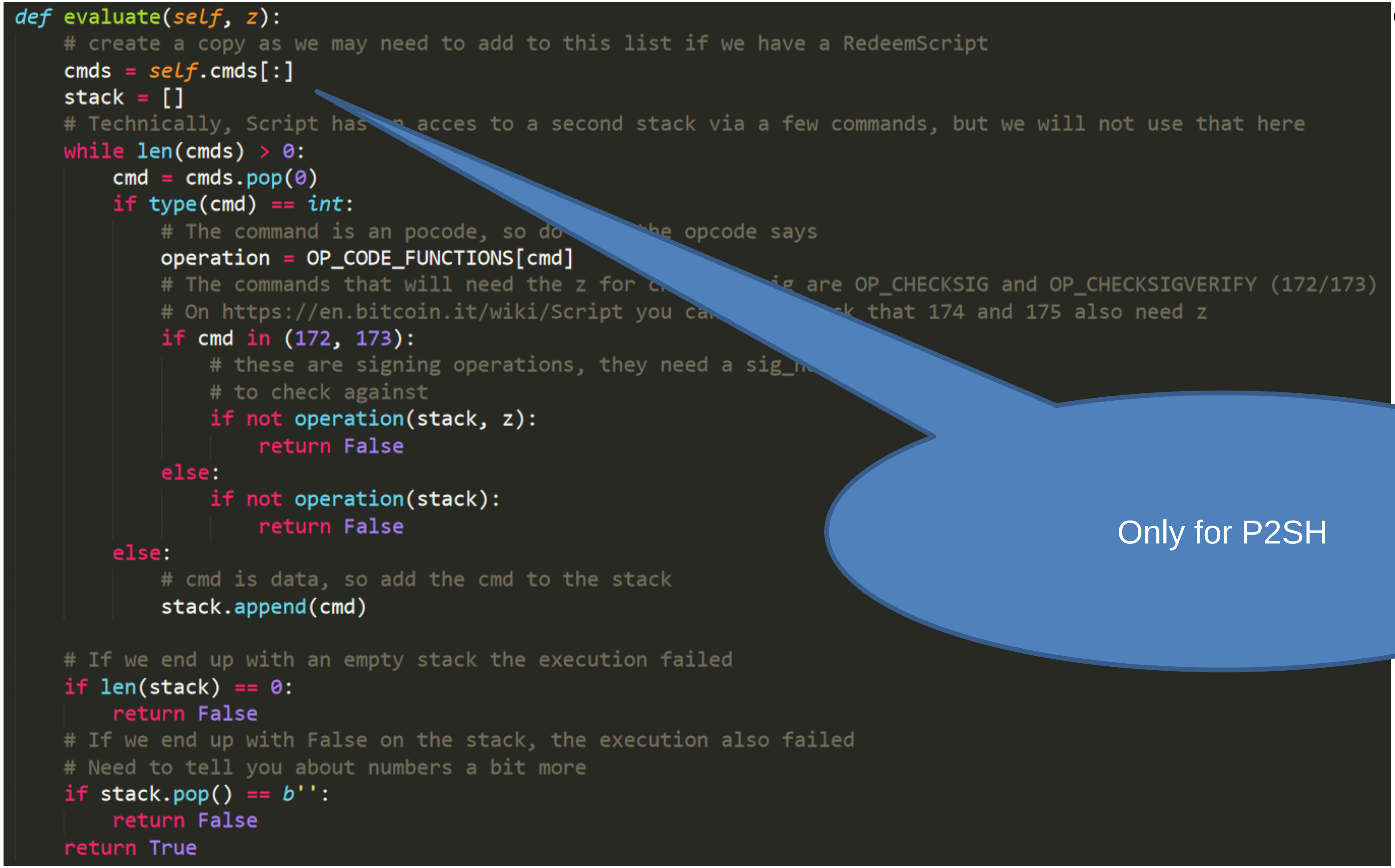


```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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 uses z!!!

Scripts

Evaluate

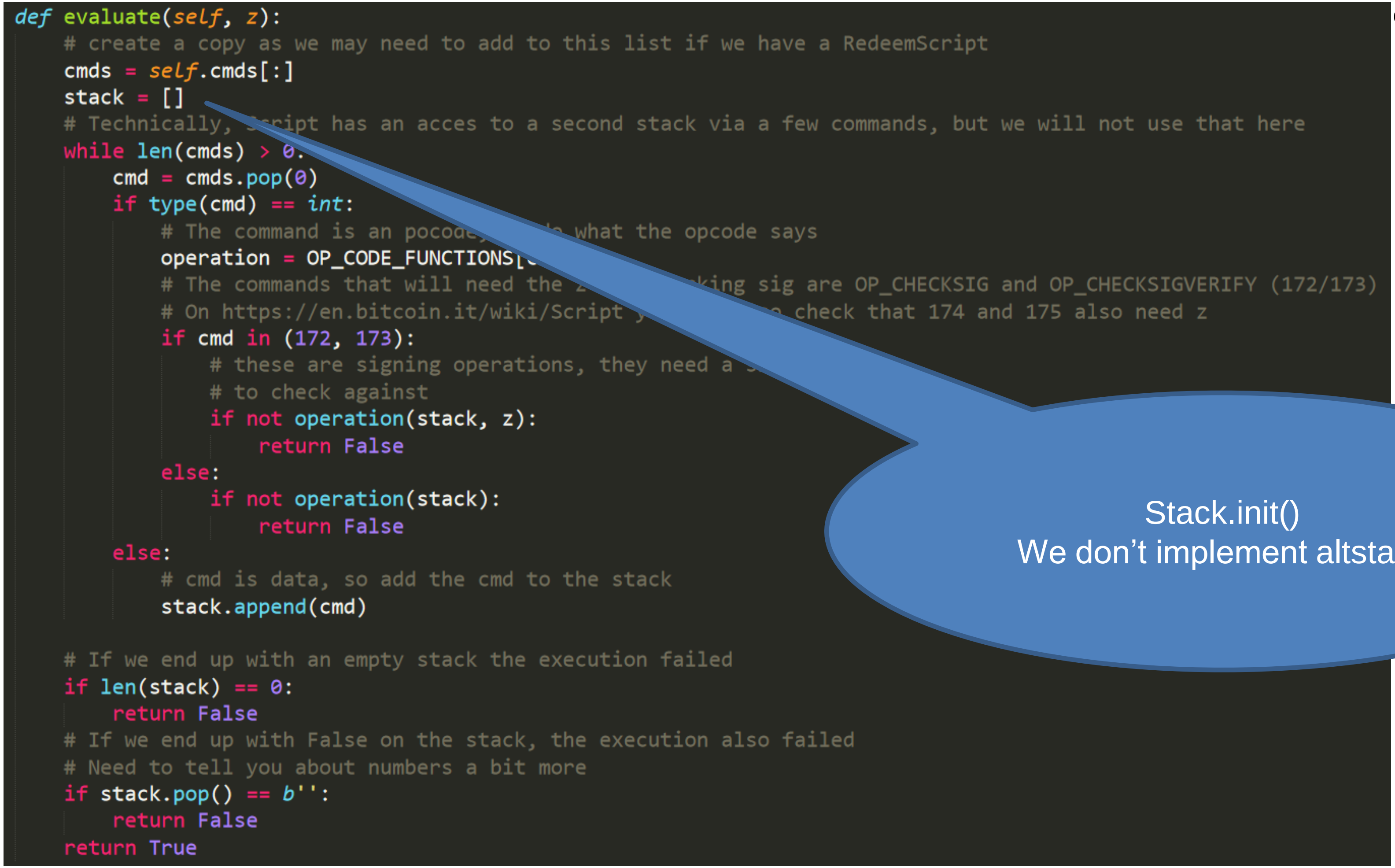


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has  n acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)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

Only for P2SH

Scripts

Evaluate

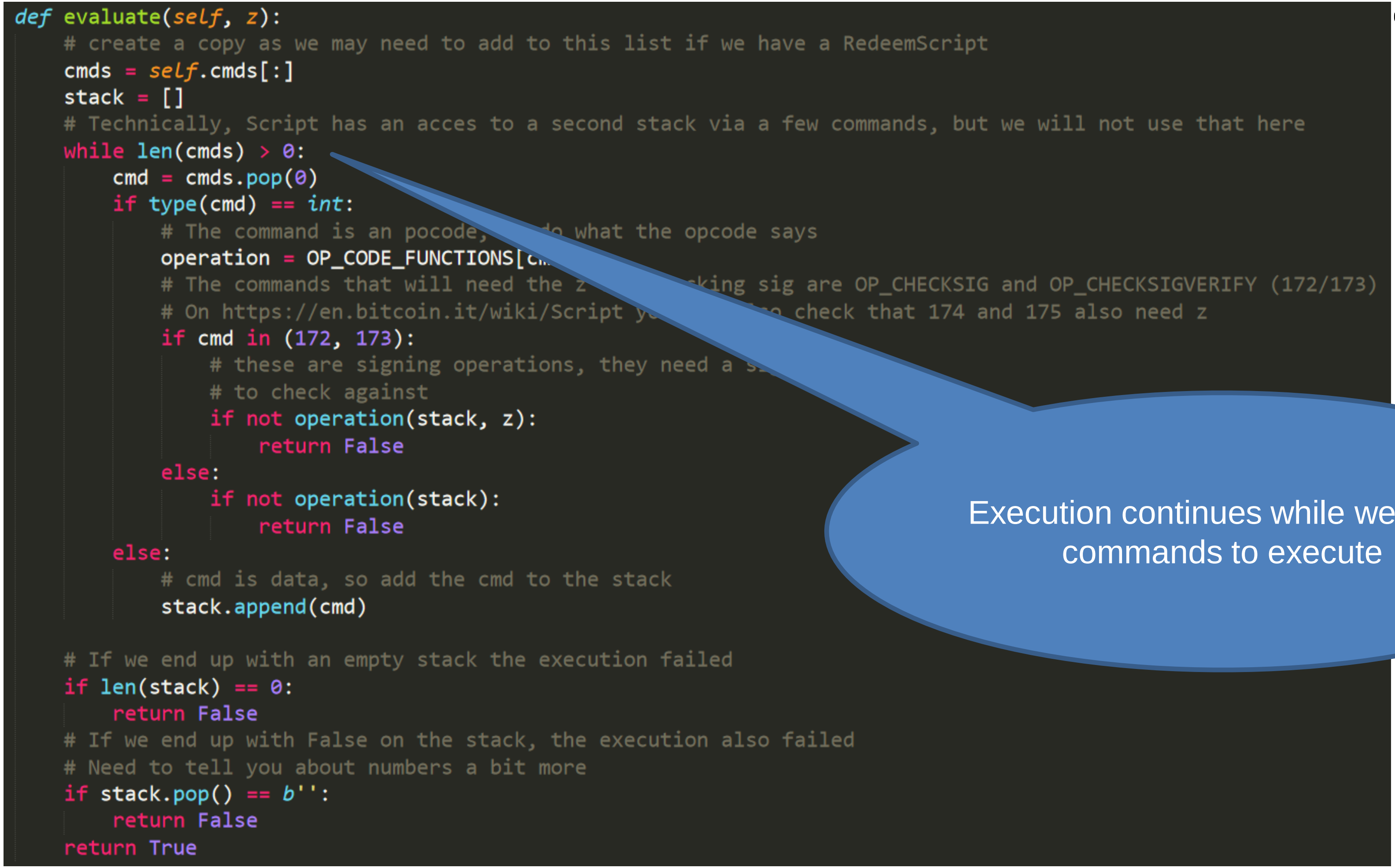


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode      what the opcode says
            operation = OP_CODE_FUNCTIONS[
            # The commands that will need the z         king sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script y          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()
We don't implement altstack

Scripts

Evaluate

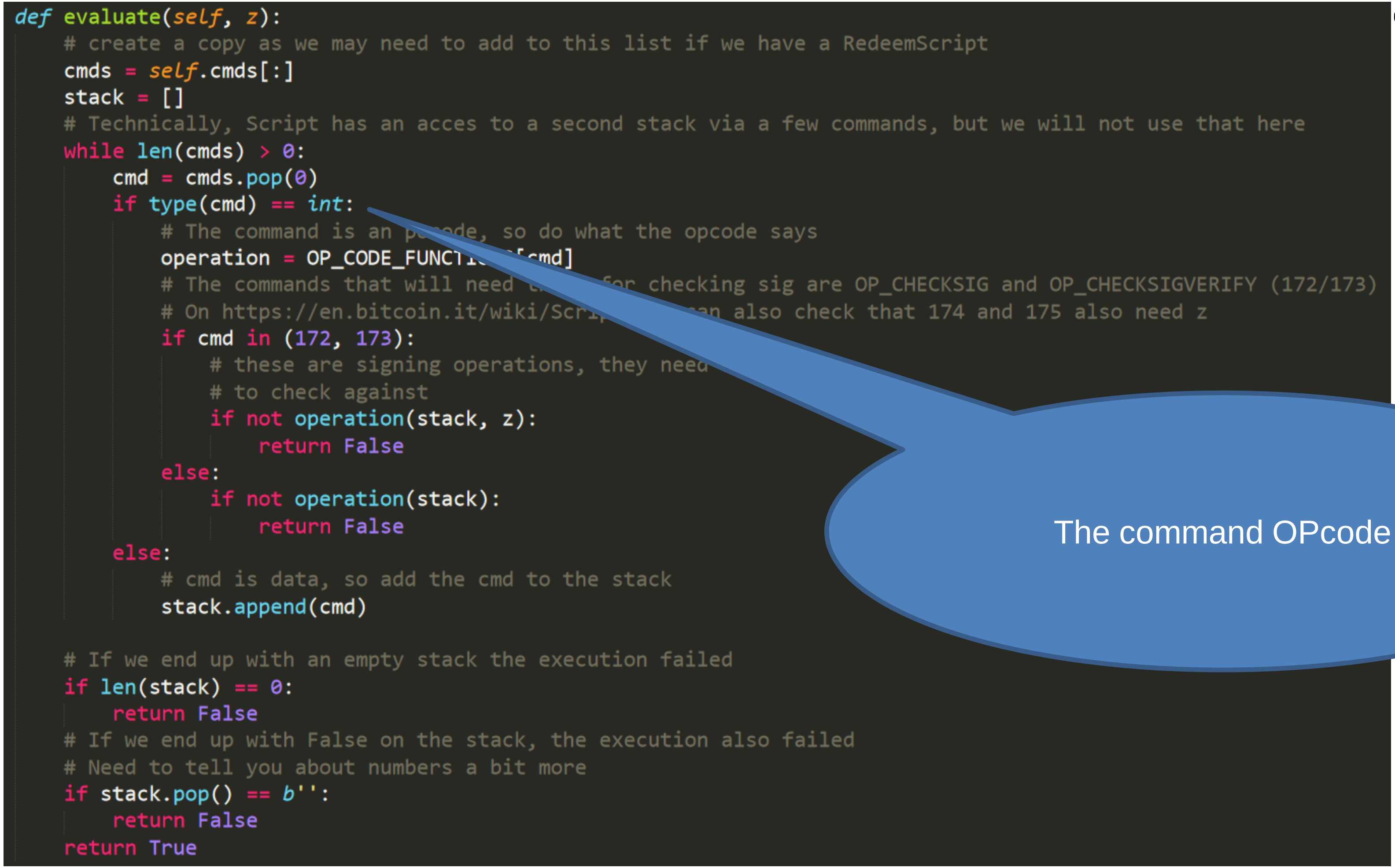


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        if type(cmd) == int:
            # The command is an pocode,      o what the opcode says
            operation = OP_CODE_FUNCTIONS[c
            # 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            o check that 174 and 175 also need z
            if cmd in (172, 173):
                # these are signing operations, they need a s
                # 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
```

Execution continues while we have commands to execute

Scripts

Evaluate

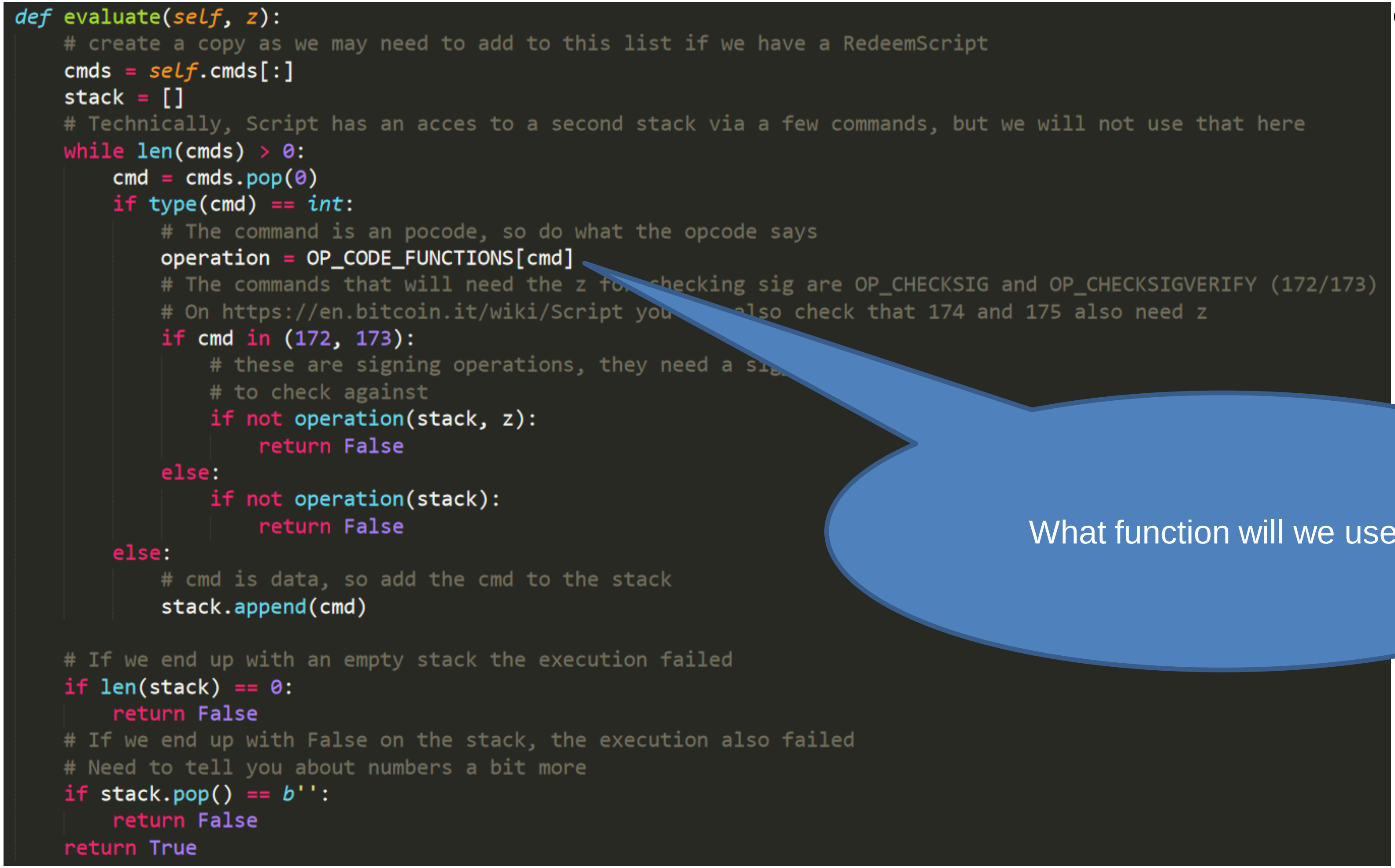


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)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

The command OPcode

Scripts

Evaluate

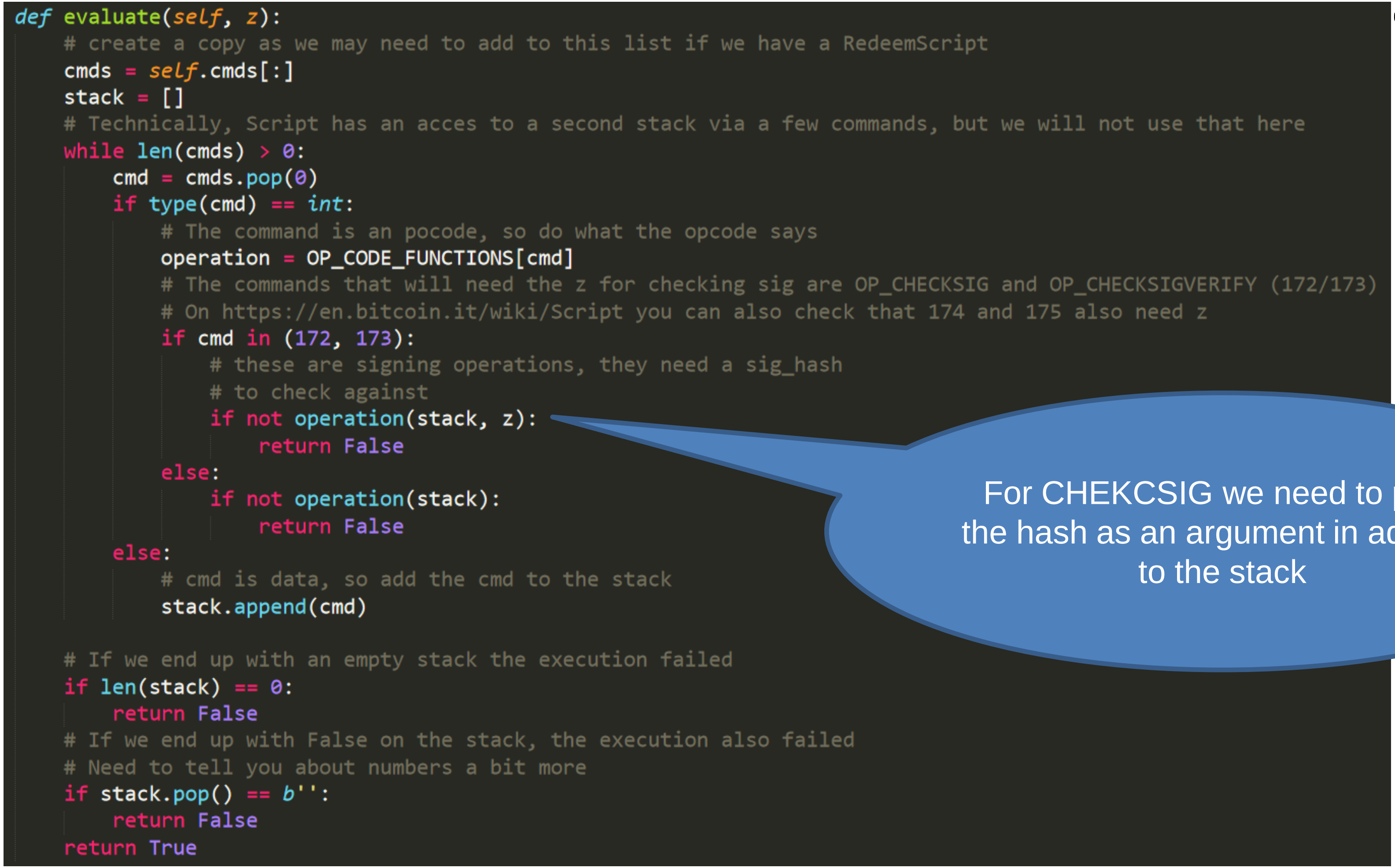


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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   checking sig are OP_CHECKSIG and OP_CHECKSIGVERIFY (172/173)
            # On https://en.bitcoin.it/wiki/Script you      also 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
```

What function will we use?

Scripts

Evaluate

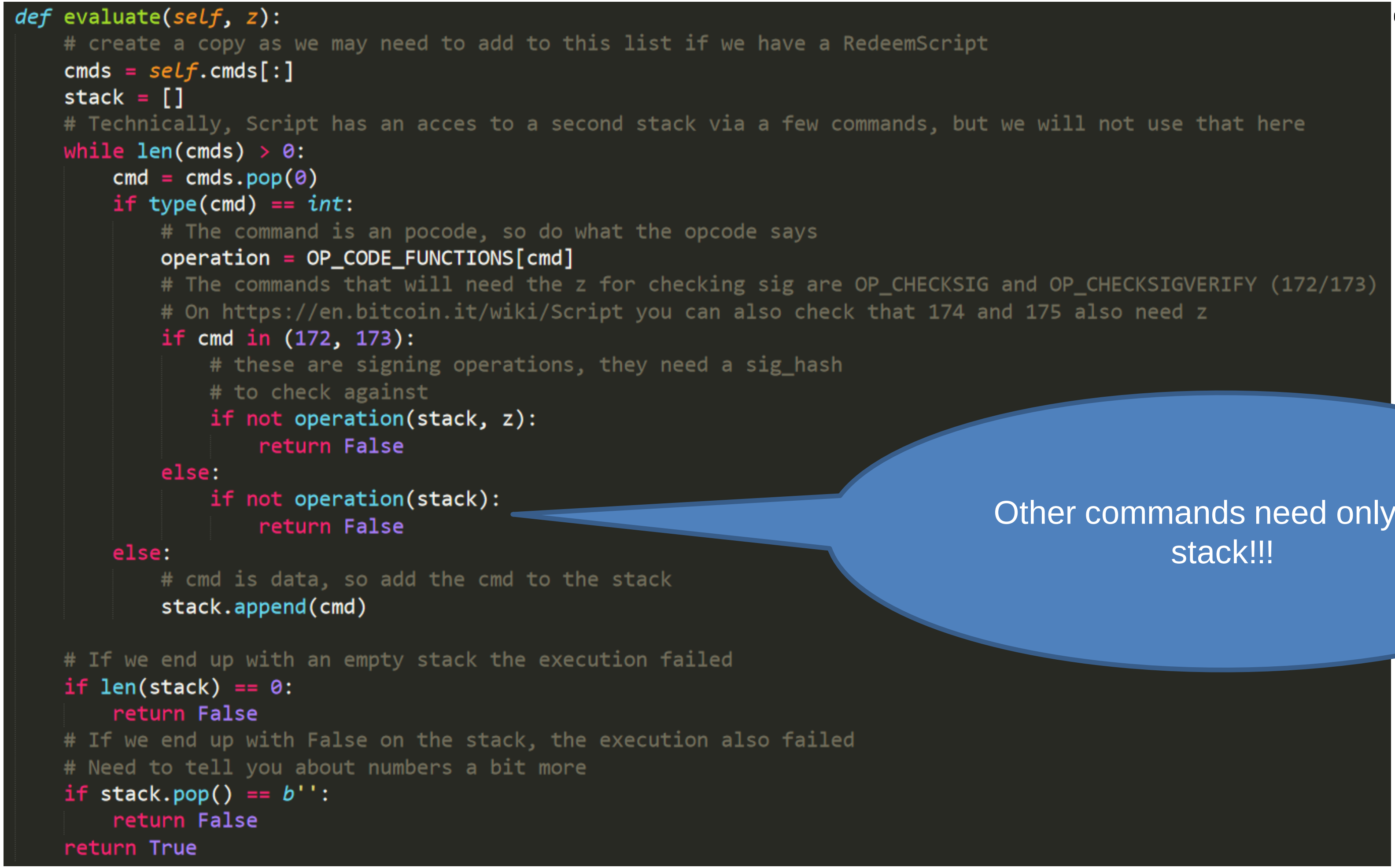


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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
```

For CHEKCSIG we need to pass the hash as an argument in addition to the stack

Scripts

Evaluate

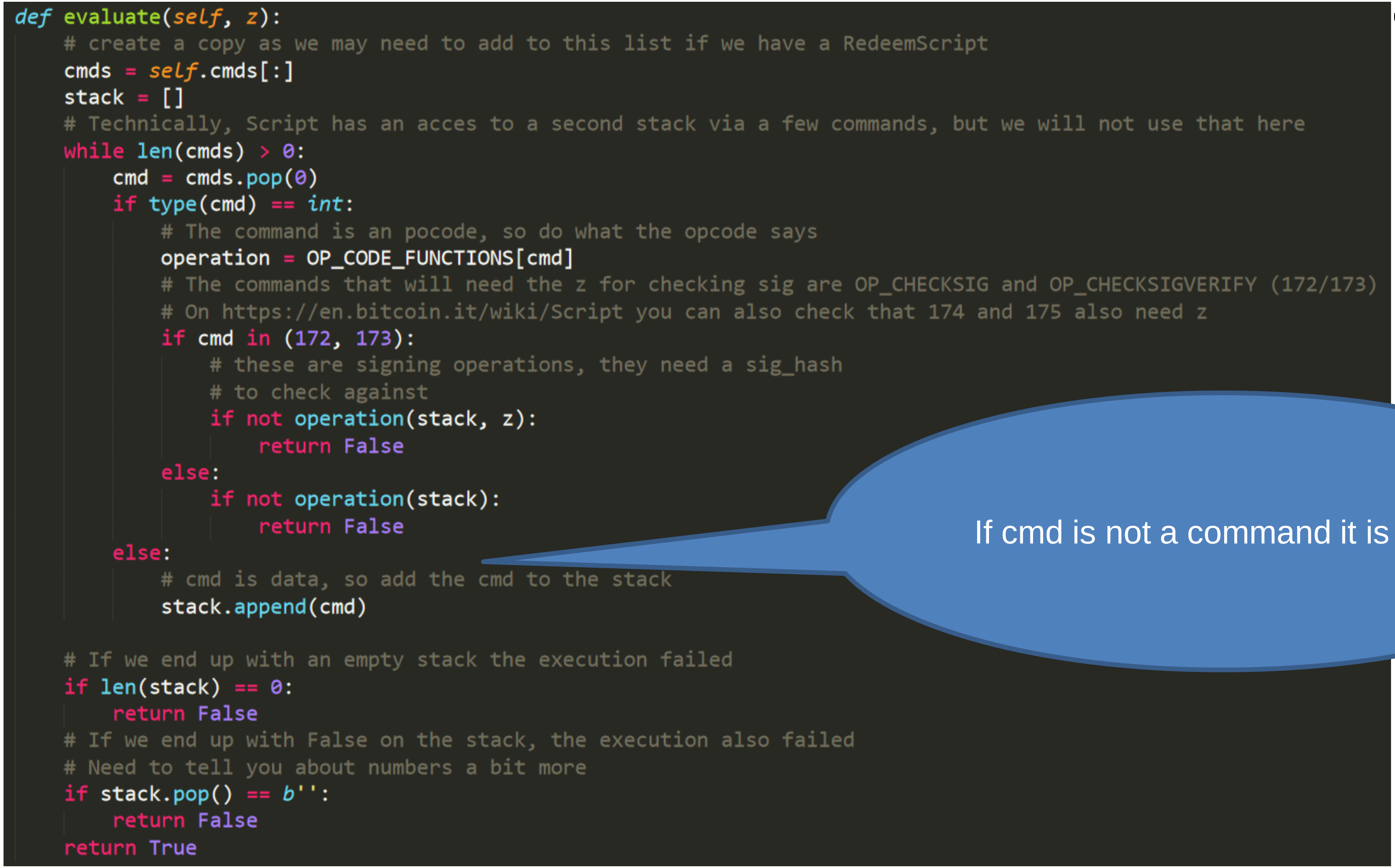


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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
```

Other commands need only the stack!!!

Scripts

Evaluate

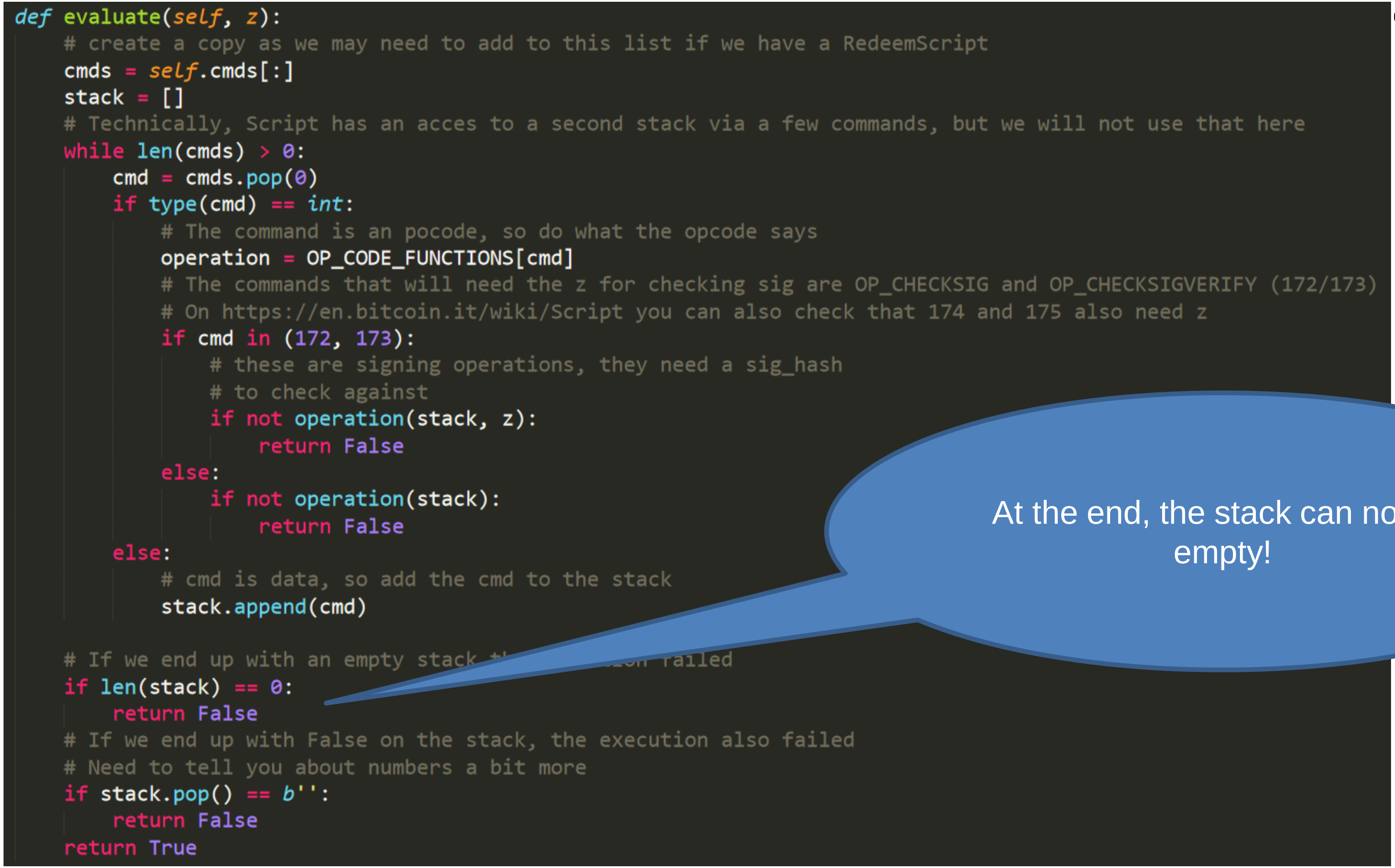


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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
```

If cmd is not a command it is data!

Scripts

Evaluate

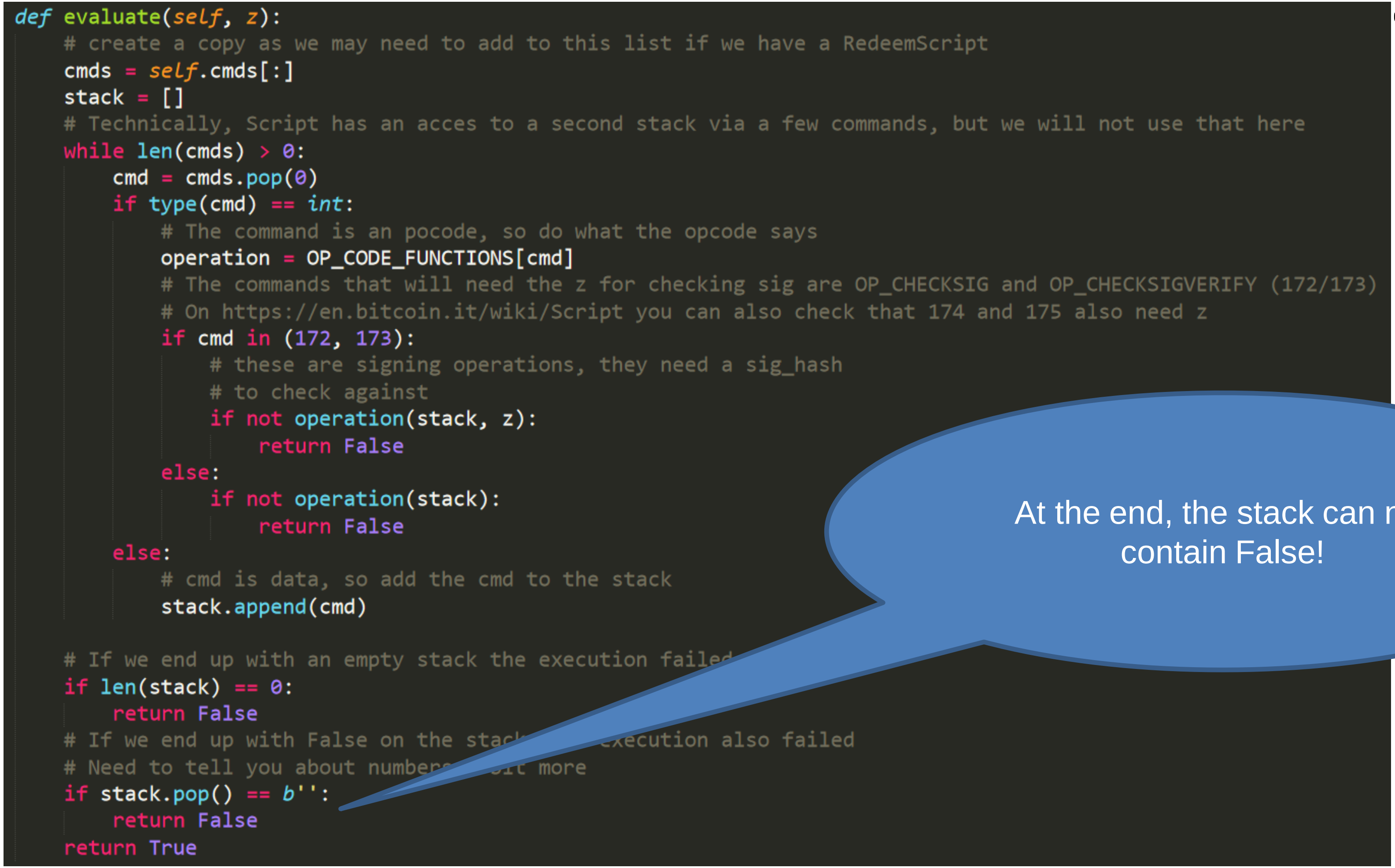


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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 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
```

At the end, the stack can not be empty!

Scripts

Evaluate

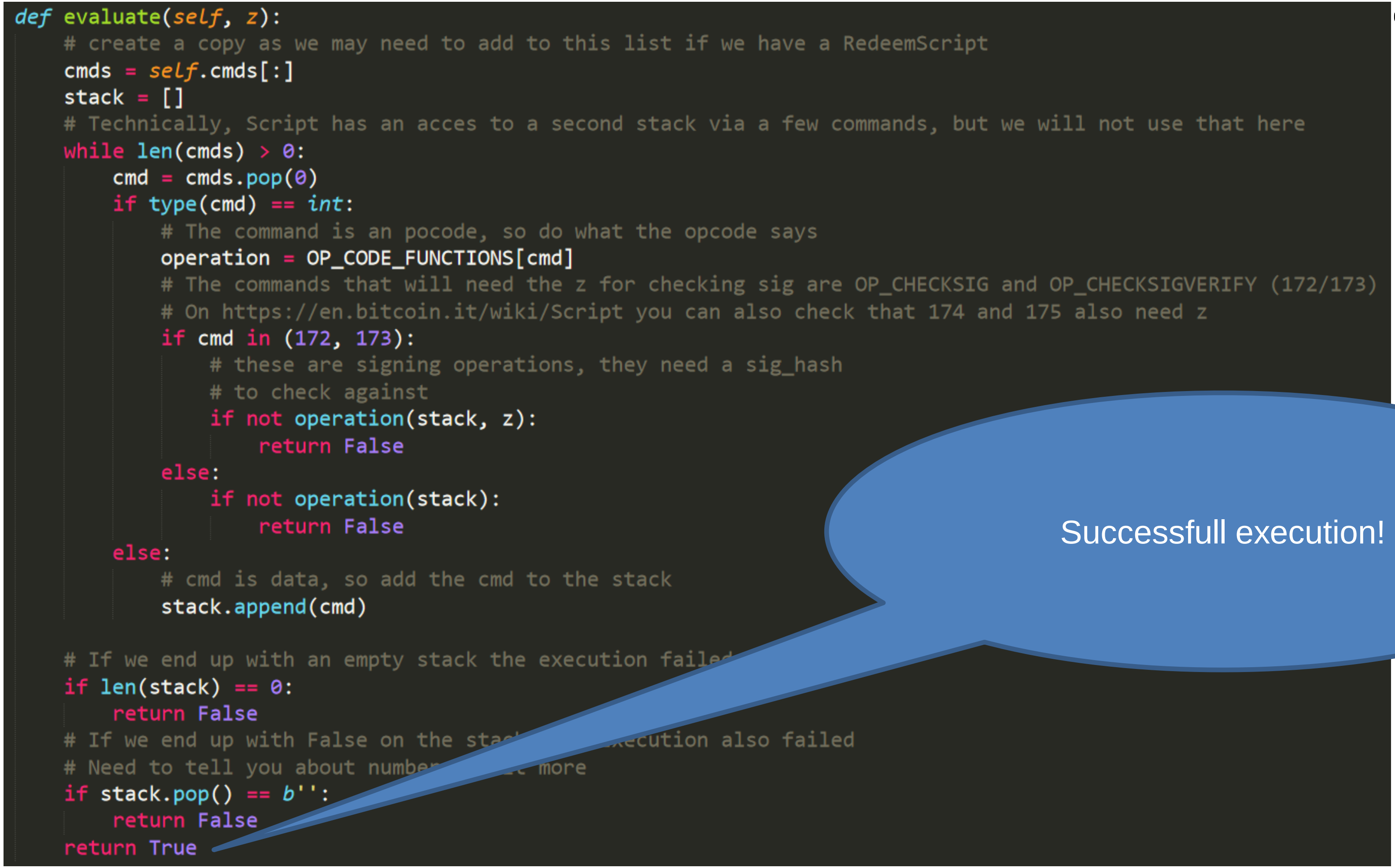


```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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          execution also failed
    # Need to tell you about numbers         bit more
    if stack.pop() == b'':
        return False
    return True
```

At the end, the stack can not contain False!

Scripts

Evaluate

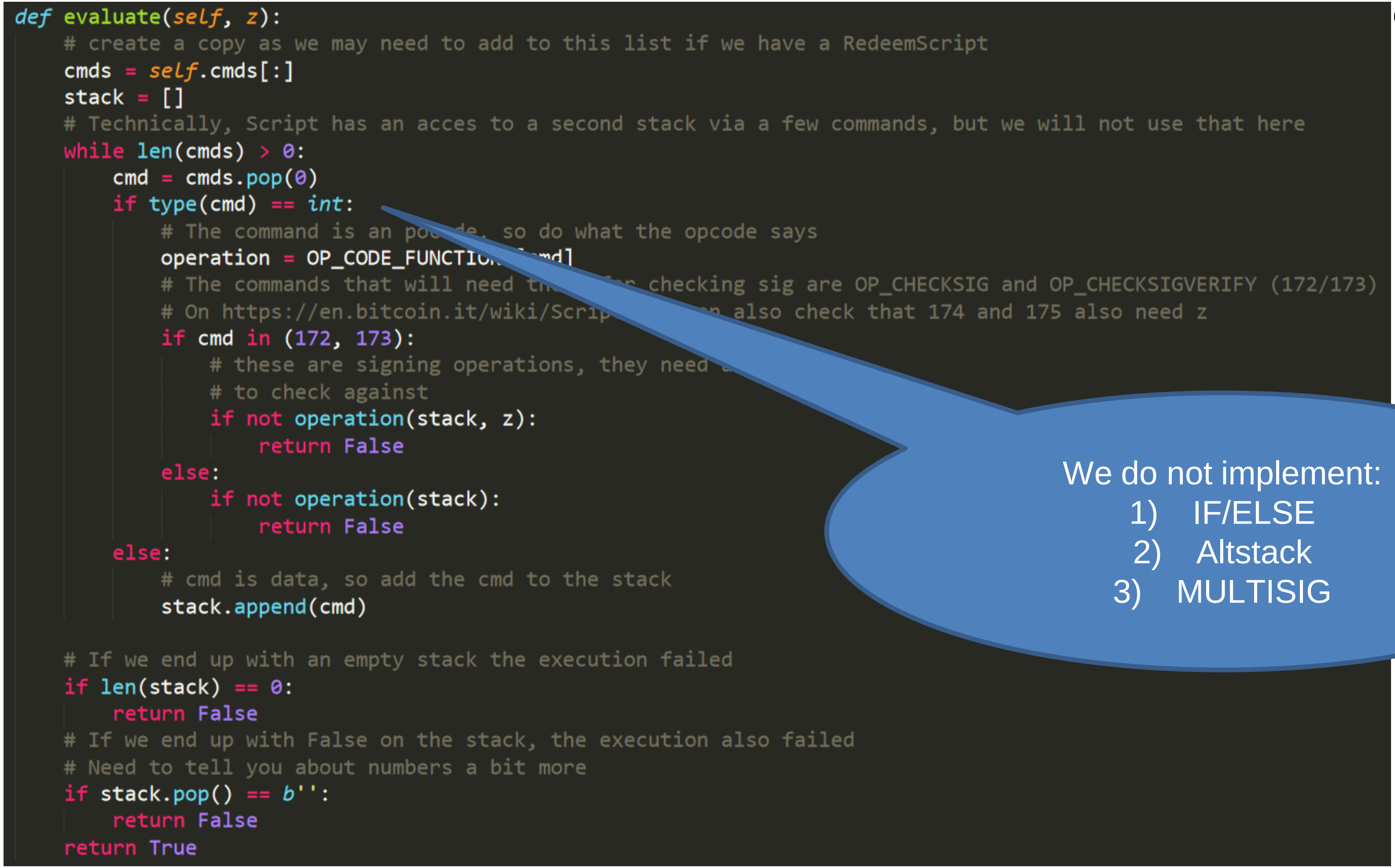


```python
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)
        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 stac          xecution also failed
    # Need to tell you about numbe          more
    if stack.pop() == b'':
        return False
    return True
```

Successfull execution!

Scripts

Evaluate



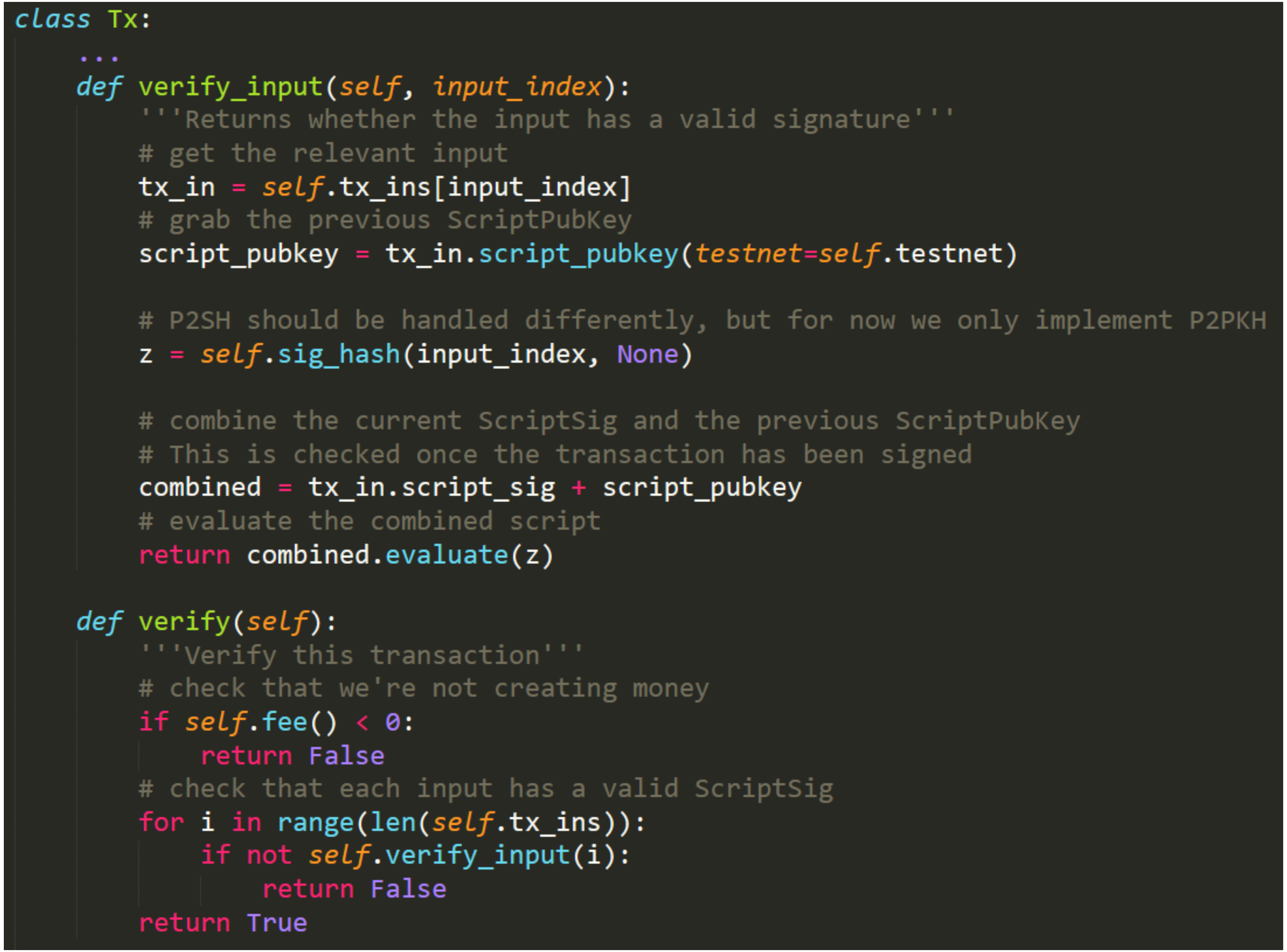
```
def evaluate(self, z):
    # create a copy as we may need to add to this list if we have a RedeemScript
    cmds = self.cmds[:]
    stack = []
    # Technically, Script has an acces to a second stack via a few commands, but we will not use that here
    while len(cmds) > 0:
        cmd = cmds.pop(0)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

We do not implement:

1) IF/ELSE
2) Altstack
3) MULTISIG

Scripts

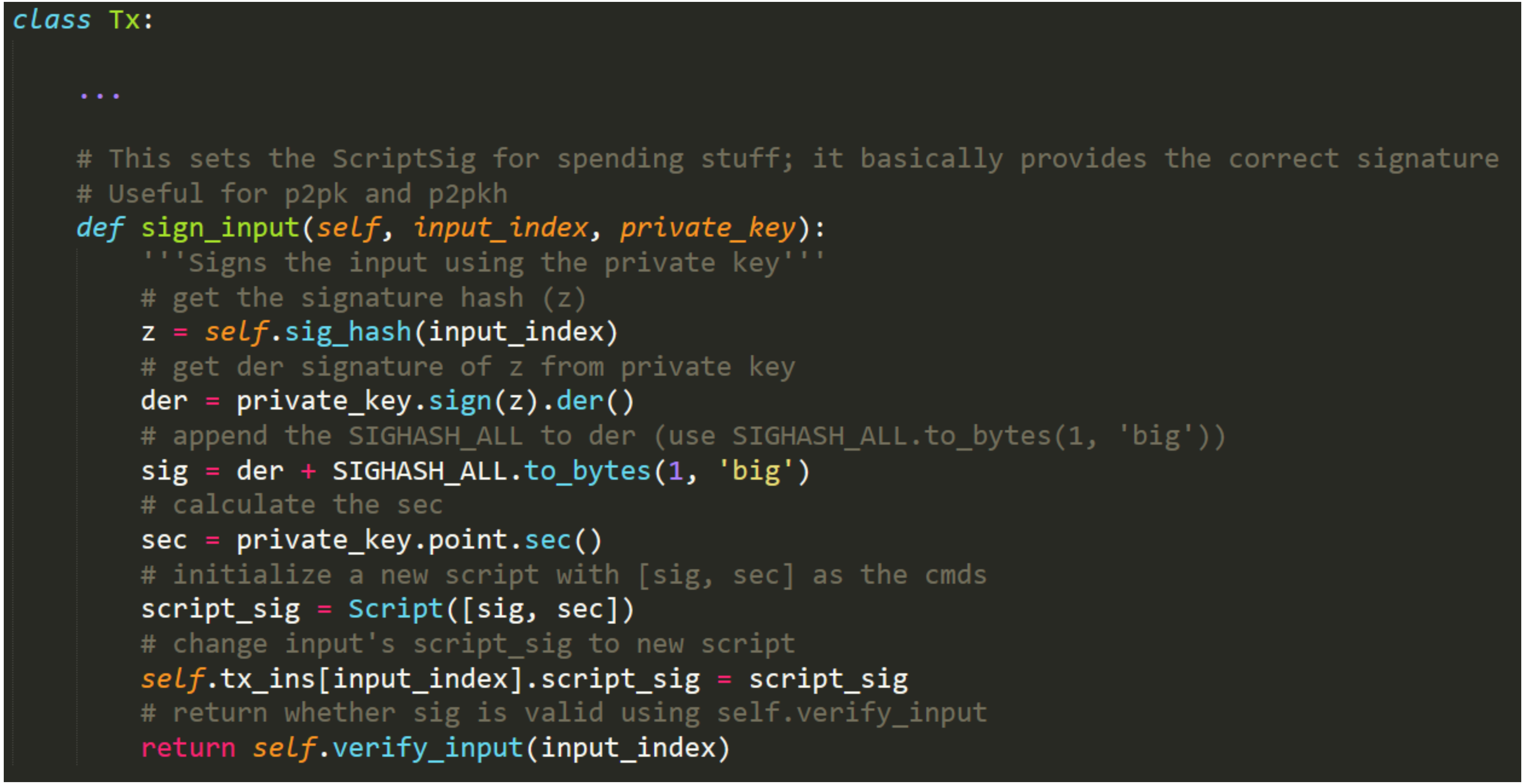
Verification



```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

Signing



```python
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)
```

Spending money

How do I spend my bitcoins?

- An address is a public key?
- What is an address used for?
- How do I find my bitcoins?
- Waaaaa!!!

Spending money

How do I spend my bitcoins?

- **An address is a public key? – only for P2PK**
- What is an address used for?
- How do I find my bitcoins?
- Waaaaa!!!

Spending money

How do I spend my bitcoins?

- An address is a public key? – only for P2PK
- **What is an address used for? – P2PKH, P2SH**
- How do I find my bitcoins?
- Waaaaa!!!

Spending money

How do I spend my bitcoins?

- An address is a public key? – only for P2PK
- What is an address used for?
- **How do I find my bitcoins? – You already know (full node)**
- Waaaaa!!!

Spending money

How do I spend my bitcoins?

- An address is a public key? – only for P2PK
- What is an address used for?
- How do I find my bitcoins? – You already know (full node)
- **Waaaaa!!! – OK, I'll make your life a bit easier!**

Spending money

~~How do I spend my bitcoins?~~

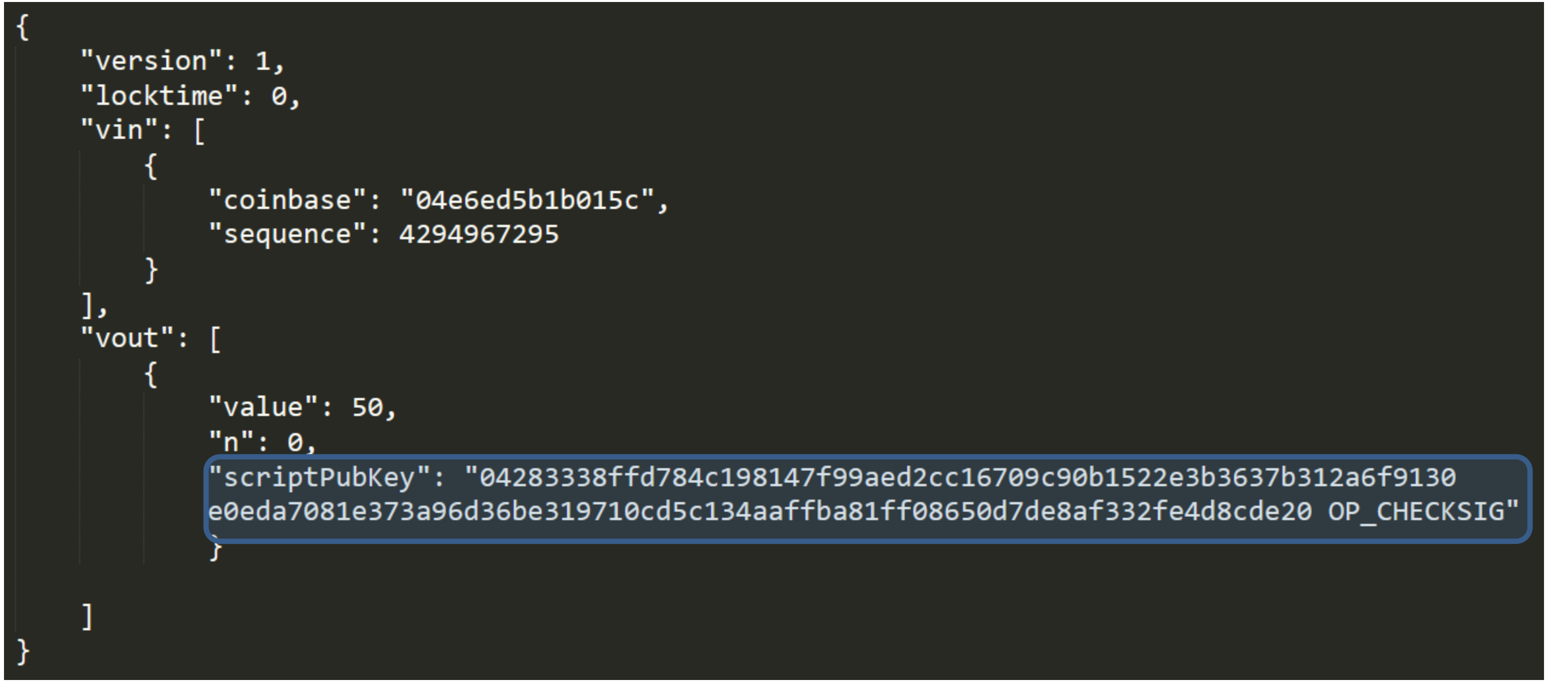
- ~~An address is a public key? – only for P2PK~~
- ~~What is an address used for?~~
- ~~How do I find my bitcoins? – You already know (full node)~~
- ~~**Waaaaa!!! – OK, I'll make your life a bit easier!**~~

I'll send you a script!!!

Which script?

Addresses

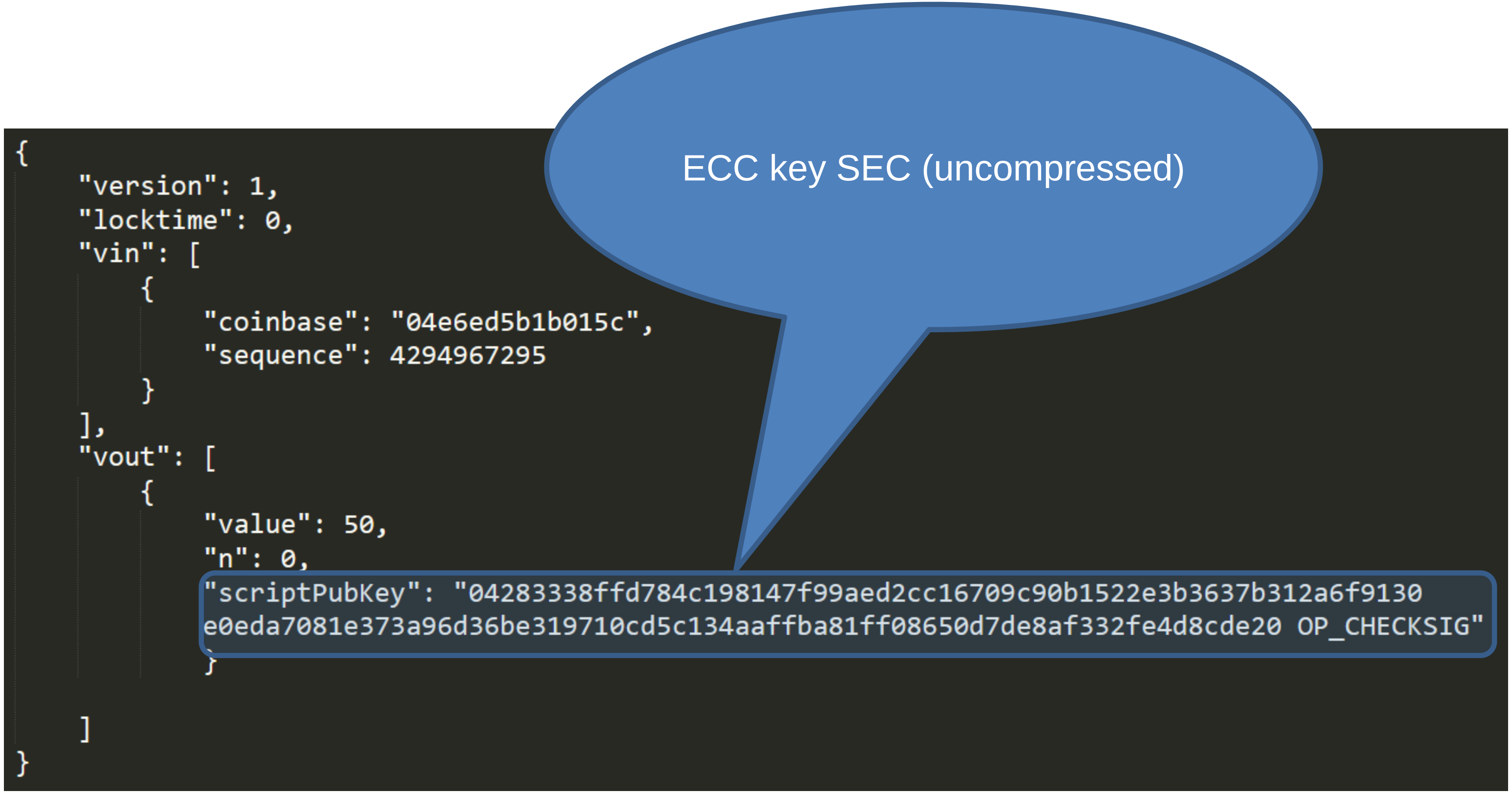
P2PK



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

    ]
}
```

Addresses

P2PK

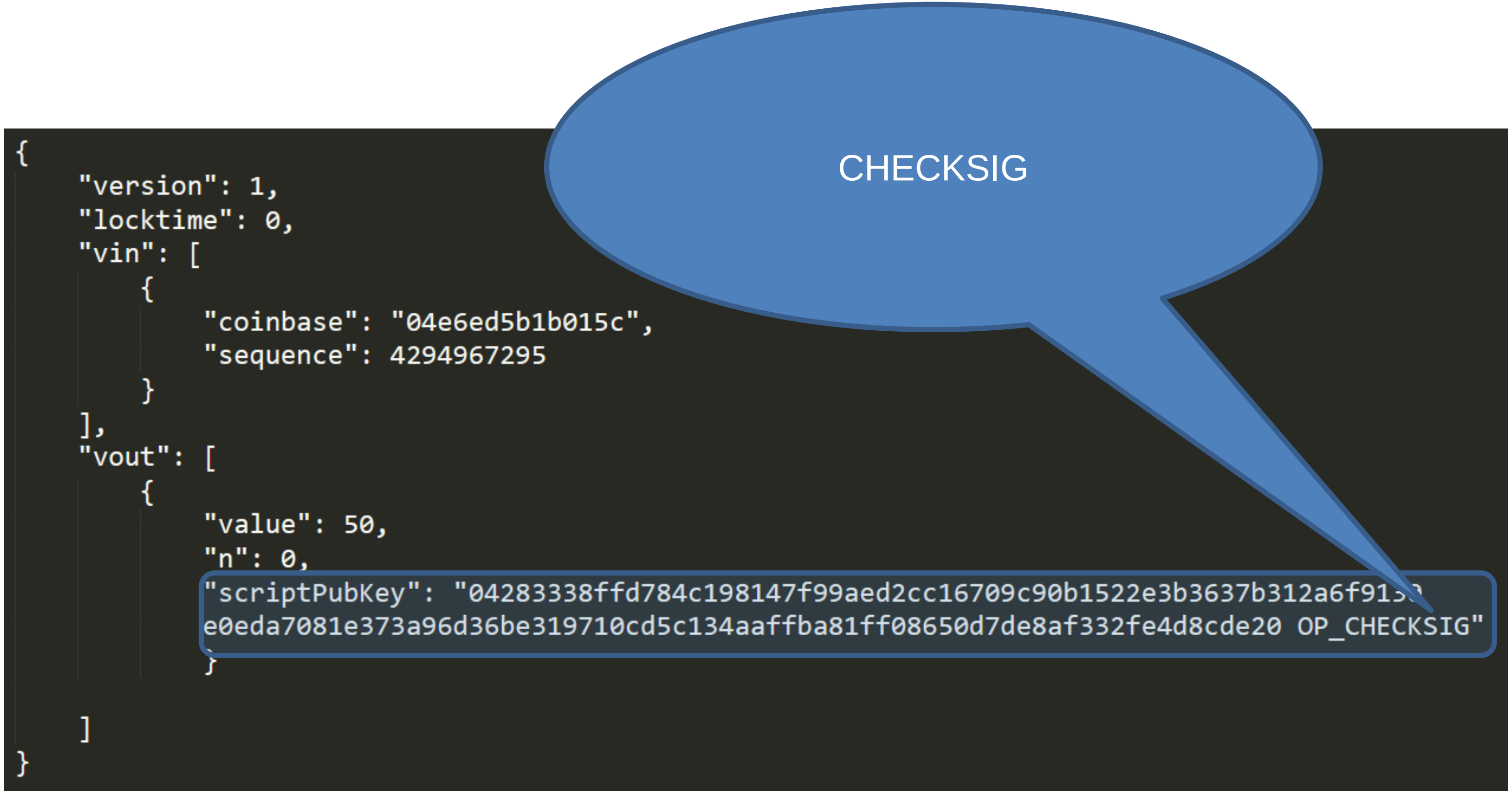


ECC key SEC (uncompressed)

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

    ]
}
```

Addresses

P2PK

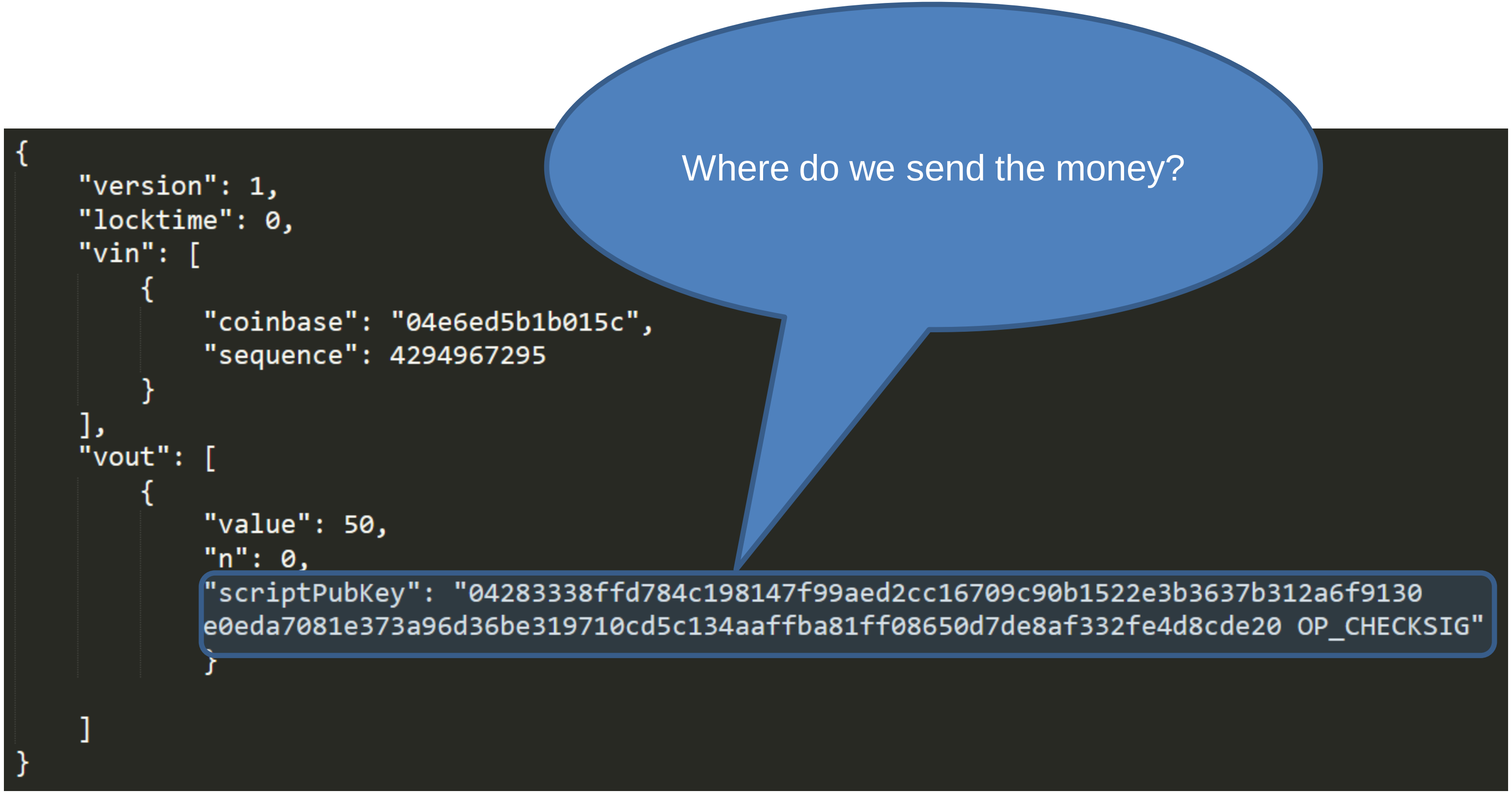


CHECKSIG

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

    ]
}
```

Addresses

P2PK

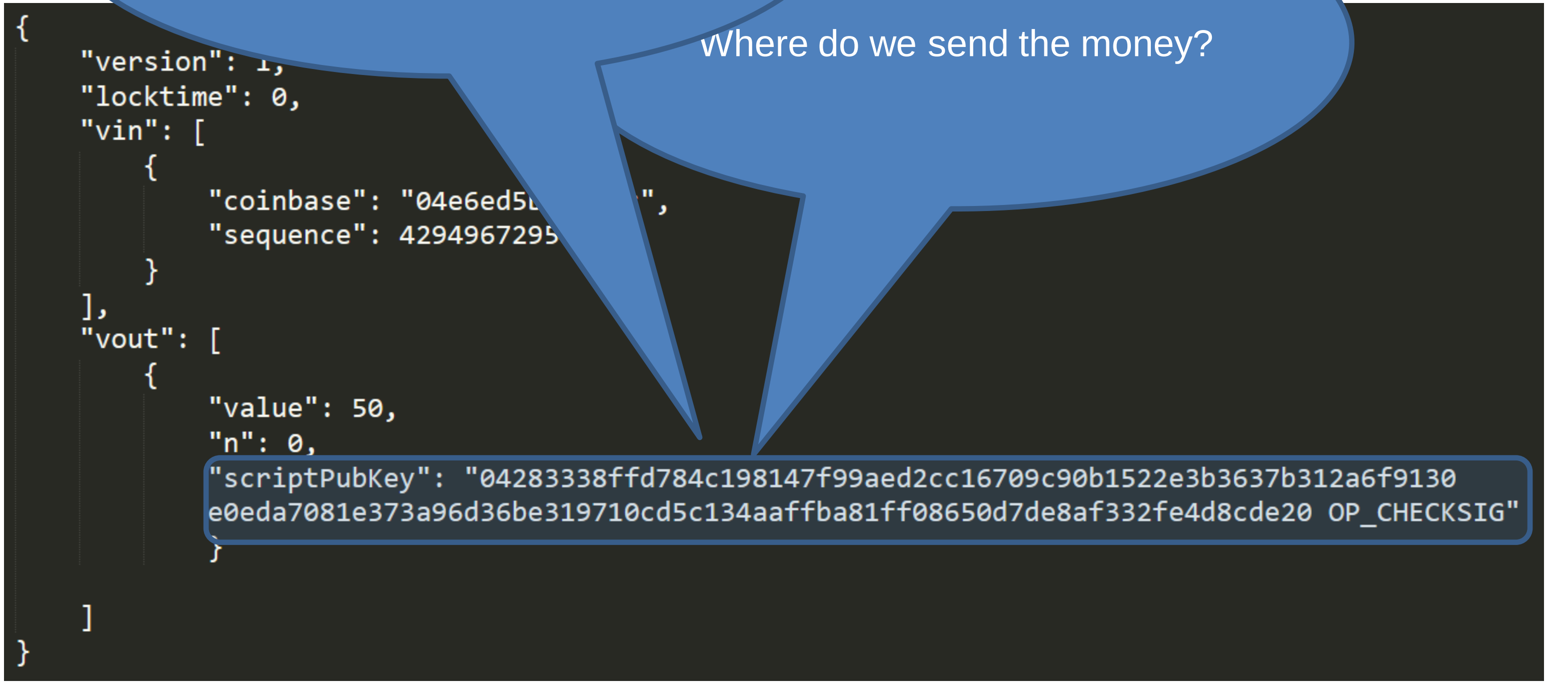


```
{
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "coinbase": "04e6ed5b1b015c",
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 50,
            "n": 0,
            "scriptPubKey": "04283338ffd784c198147f99aed2cc16709c90b1522e3b3637b312a6f9130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

    ]
}
```

Where do we send the money?

Addresses

P2PK



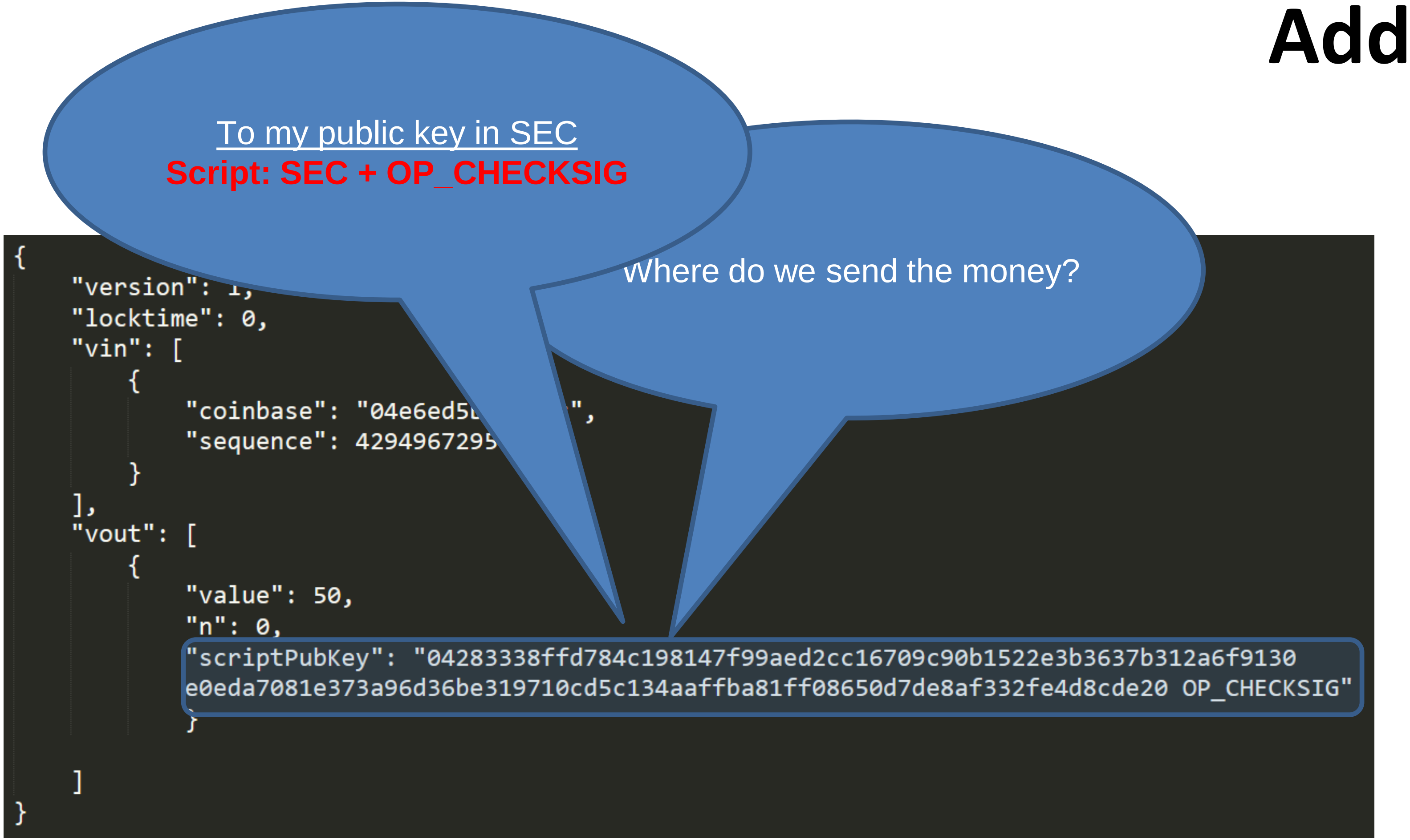
```
{
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "coinbase": "04e6ed5b",
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 50,
            "n": 0,
            "scriptPubKey": "04283338ffd784c198147f99aed2cc16709c90b1522e3b3637b312a6f9130
e0eda7081e373a96d36be319710cd5c134aaffba81ff08650d7de8af332fe4d8cde20 OP_CHECKSIG"
            }

    ]
}
```

To my public key in SEC

Where do we send the money?

Addresses

P2PK



To my public key in SEC

Script: SEC + OP_CHECKSIG

Where do we send the money?

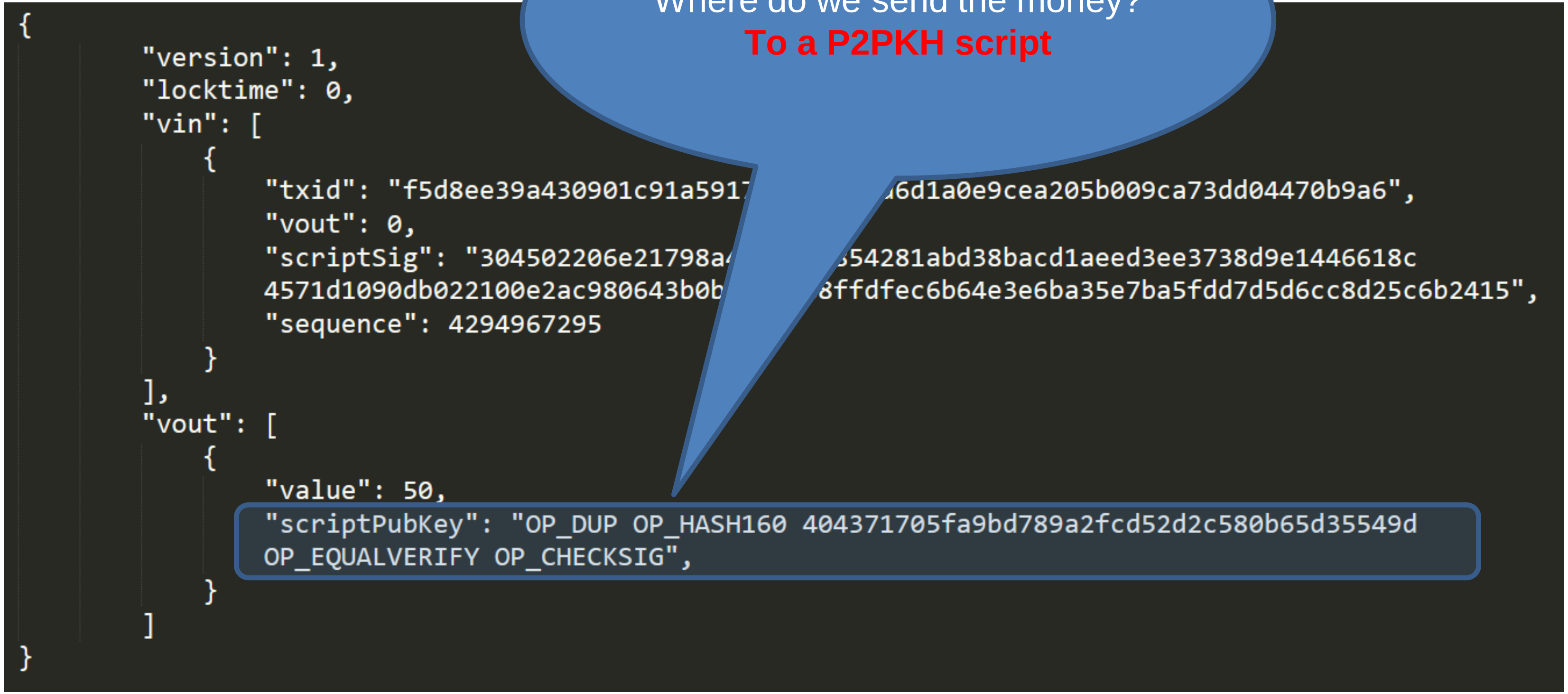
```
{
    "version": 1,
    "locktime": 0,
    "vin": [
        {
            "coinbase": "04e6ed5b        ",
            "sequence": 4294967295
        }
    ],
    "vout": [
        {
            "value": 50,
            "n": 0,
            "scriptPubKey": "04283338ffd784c198147f99aed2cc16709c90b1522e3b3637b312a6f9130
e0eda7081e373a96d36be319710cd5c134aaffba81ff08650d7de8af332fe4d8cde20 OP_CHECKSIG"
        }

    ]
}
```

Addresses

P2PKH

Where do we send the money?

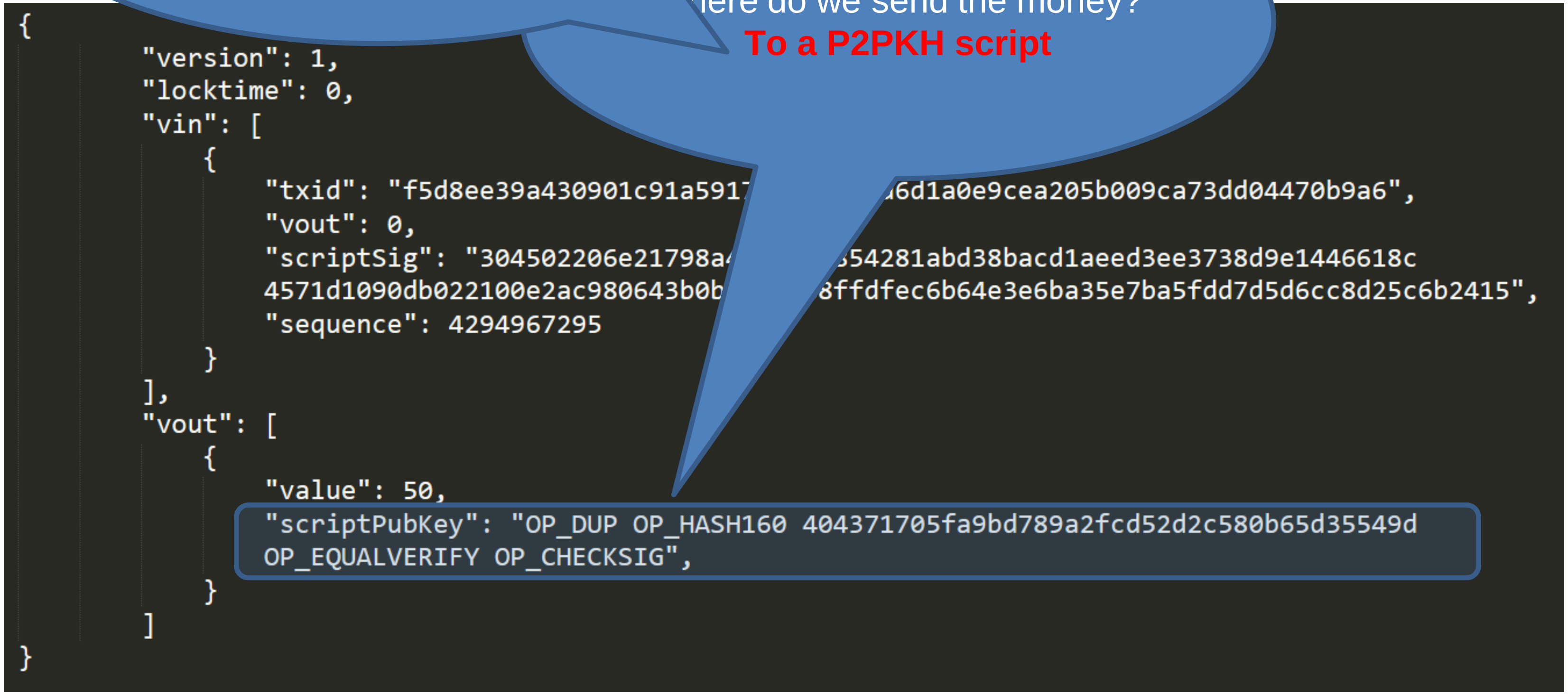
To a P2PKH script



```
{
        "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",
            }
        ]
}
```

Addresses

P2PKH



```
{
        "version": 1,
        "locktime": 0,
        "vin": [
            {
                "txid": "f5d8ee39a430901c91a5917          6d1a0e9cea205b009ca73dd04470b9a6",
                "vout": 0,
                "scriptSig": "304502206e21798a          54281abd38bacd1aeed3ee3738d9e1446618c
                4571d1090db022100e2ac980643b0b          8ffdfec6b64e3e6ba35e7ba5fdd7d5d6cc8d25c6b2415",
                "sequence": 4294967295
            }
        ],
        "vout": [
            {
                "value": 50,
                "scriptPubKey": "OP_DUP OP_HASH160 404371705fa9bd789a2fcd52d2c580b65d35549d
                OP_EQUALVERIFY OP_CHECKSIG",
            }
        ]
}
```

How do we specify this??

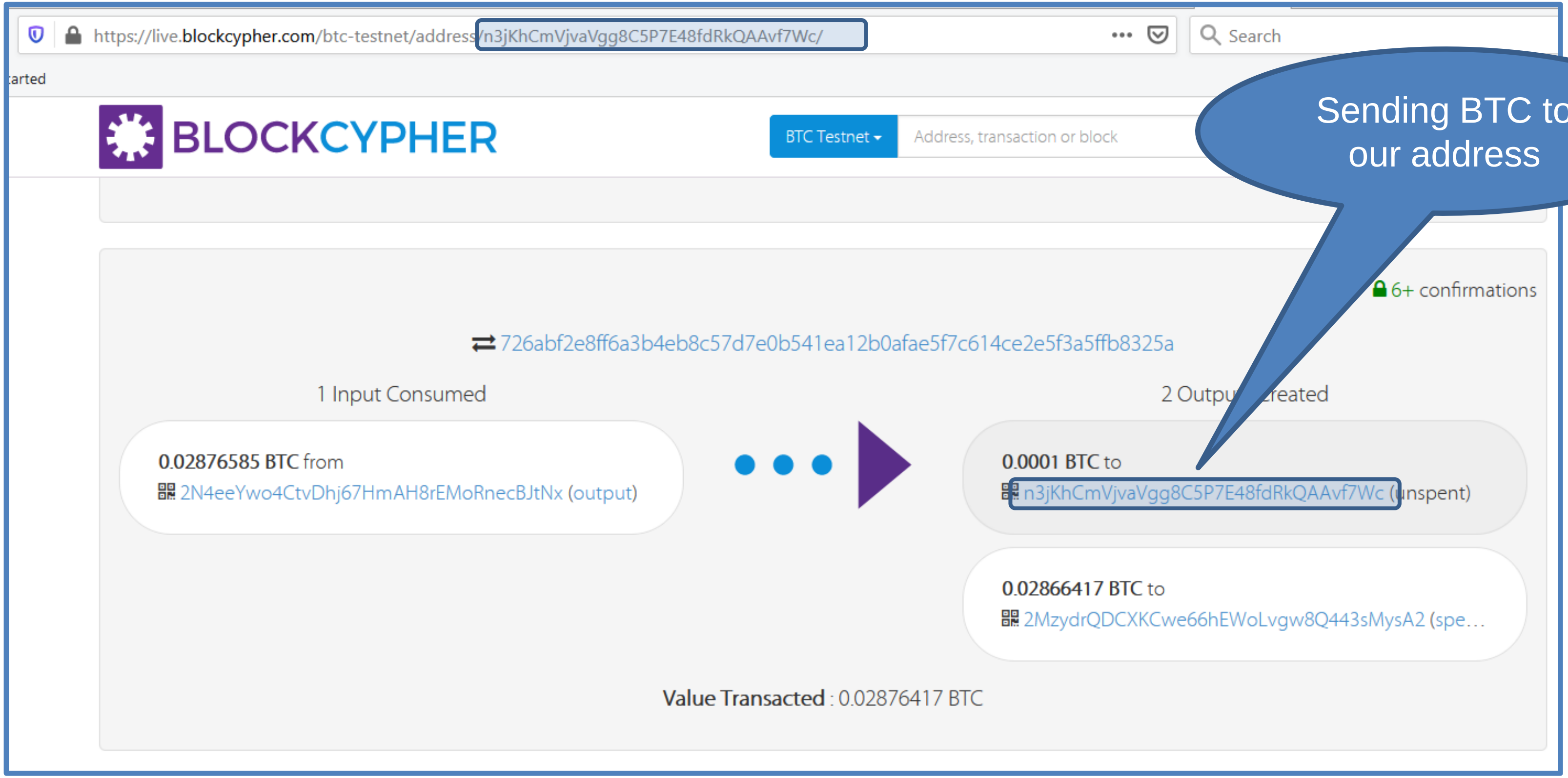
With my address!

here do we send the money?

To a P2PKH script

Addresses

P2PKH



https://live.blockcypher.com/btc-testnet/address/n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc/

6+ confirmations

726abf2e8ff6a3b4eb8c57d7e0b541ea12b0afae5f7c614ce2e5f3a5ffb8325a

1 Input Consumed

0.02876585 BTC from
2N4eeYwo4CtvDhj67HmAH8rEMoRnecBJtNx (output)

2 Outpu reated

0.0001 BTC to
n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc (unspent)

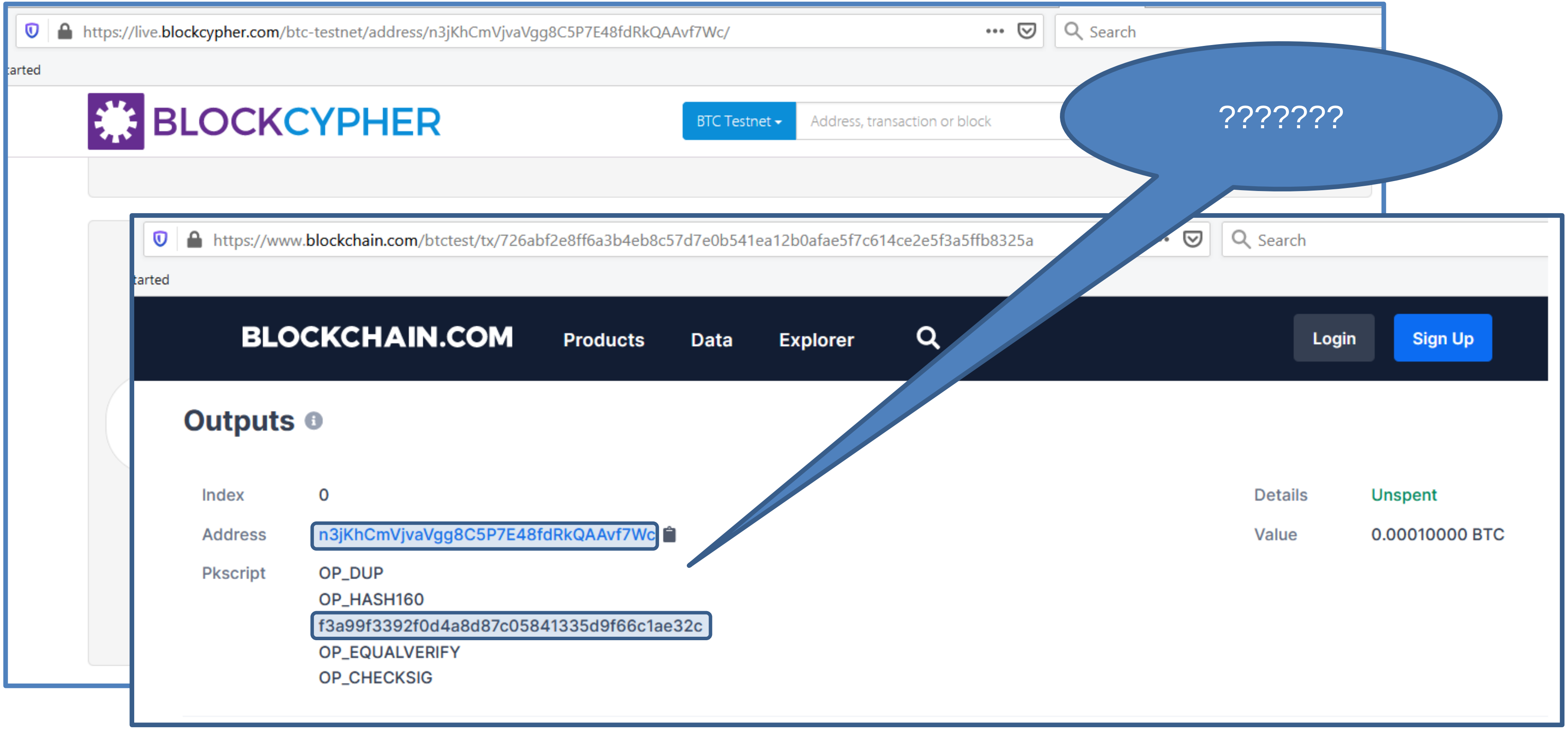
0.02866417 BTC to
2MzydrQDCXKCwe66hEWoLvgw8Q443sMysA2 (spe…

Value Transacted : 0.02876417 BTC

Sending BTC to our address

Addresses

P2PKH



https://live.blockcypher.com/btc-testnet/address/n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc/
Search
tarted
BLOCKCYPHER
BTC Testnet ▾
Address, transaction or block
https://www.blockchain.com/btctest/tx/726abf2e8ff6a3b4eb8c57d7e0b541ea12b0afae5f7c614ce2e5f3a5ffb8325a
Search
tarted
BLOCKCHAIN.COM
Products
Data
Explorer
Login
Sign Up

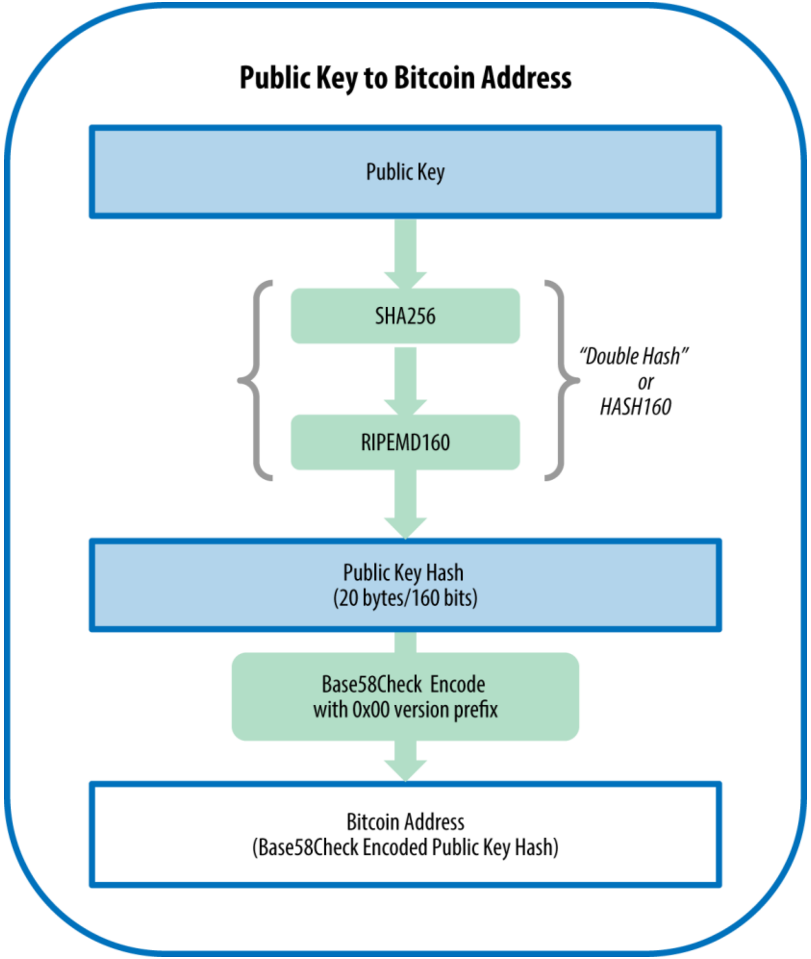
???????

Outputs

Index	0	Details	Unspent
Address	n3jKhCmVjvaVgg8C5P7E48fdRkQAAvf7Wc	Value	0.00010000 BTC
Pkscript	OP_DUP OP_HASH160 f3a99f3392f0d4a8d87c05841335d9f66c1ae32c OP_EQUALVERIFY OP_CHECKSIG		

A Bitcoin address

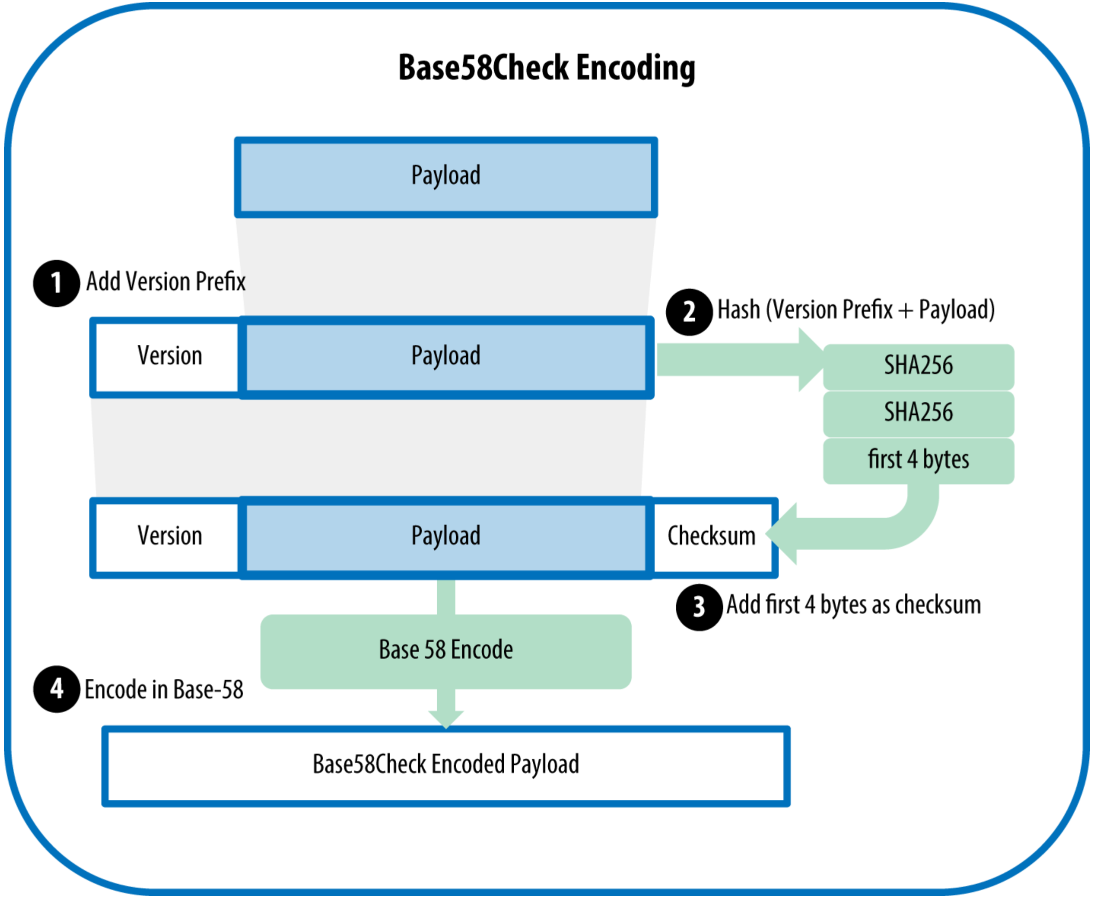
Is not a public key


Public Key to Bitcoin Address
Public Key
SHA256
RIPEMD160
"Double Hash" or HASH160
Public Key Hash (20 bytes/160 bits)
Base58Check Encode with 0x00 version prefix
Bitcoin Address (Base58Check Encoded Public Key Hash)

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

A Bitcoin address

Is not a public key



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

Addresses in BitCoin

For P2PKH!!!

P a public key in SEC format

How do we get the Bitcoin address?

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

Return encode_base58(res):

- Produces a sequence starting with 1 (mainnet), or m/n (testnet)
- This prefix tells me what type of address this is!
- https://en.bitcoin.it/wiki/List_of_address_prefixes

Addresses in BitCoin

For P2PKH!!!

An address with the prefix *0x00* or *0x6f* (1,m,n in base58):

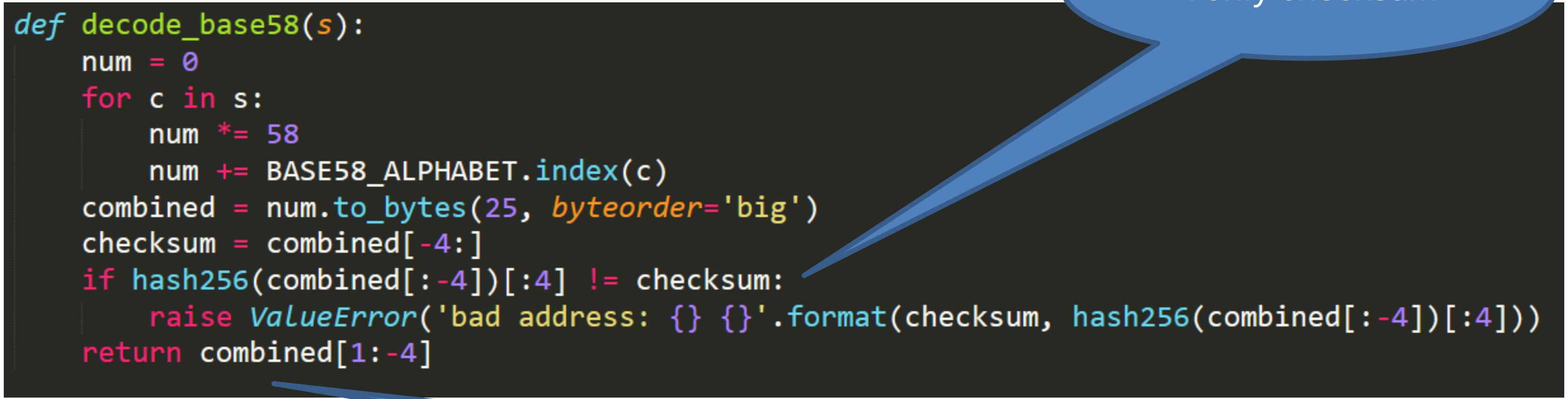
- Tells me: pay me to a P2PKH
- OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG

To which <hash>?

- D my address P2PKH
- Convert D from base 58 to bytes
- Remove the prefix
- Remove the checksum
- The result is my <hash>

Addresses in BitCoin

For P2PKH!!!



```
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]
```

Verify checksum

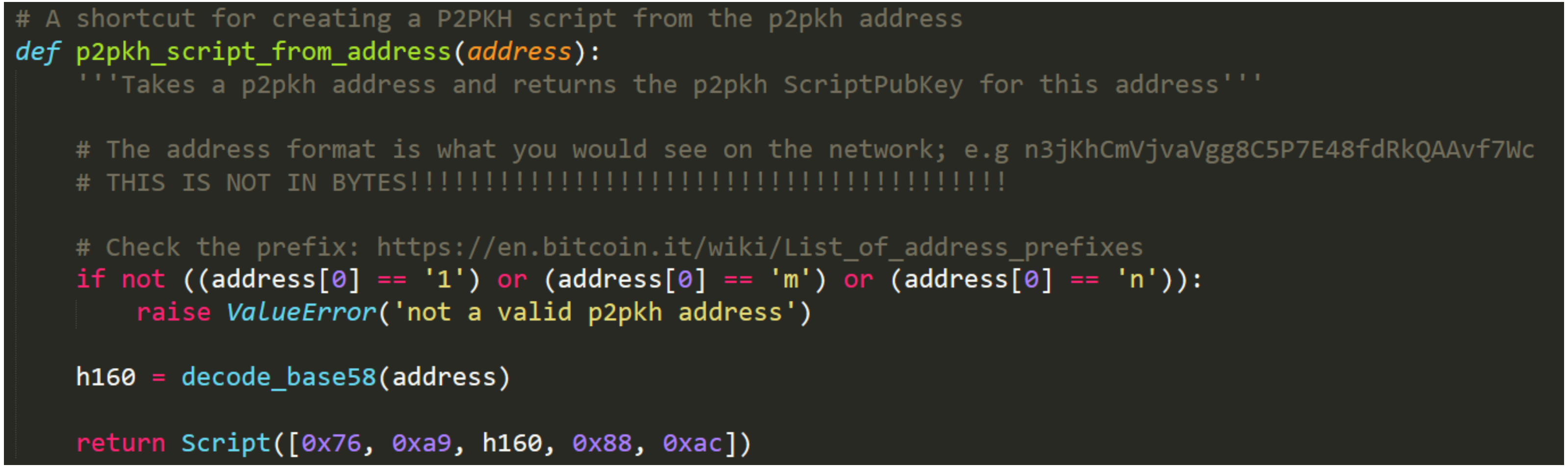
Remove prefix and checksum

Addresses in BitCoin

For P2PKH!!!

An address with the prefix *0x00* or *0x6f* (1,m,n in base58):

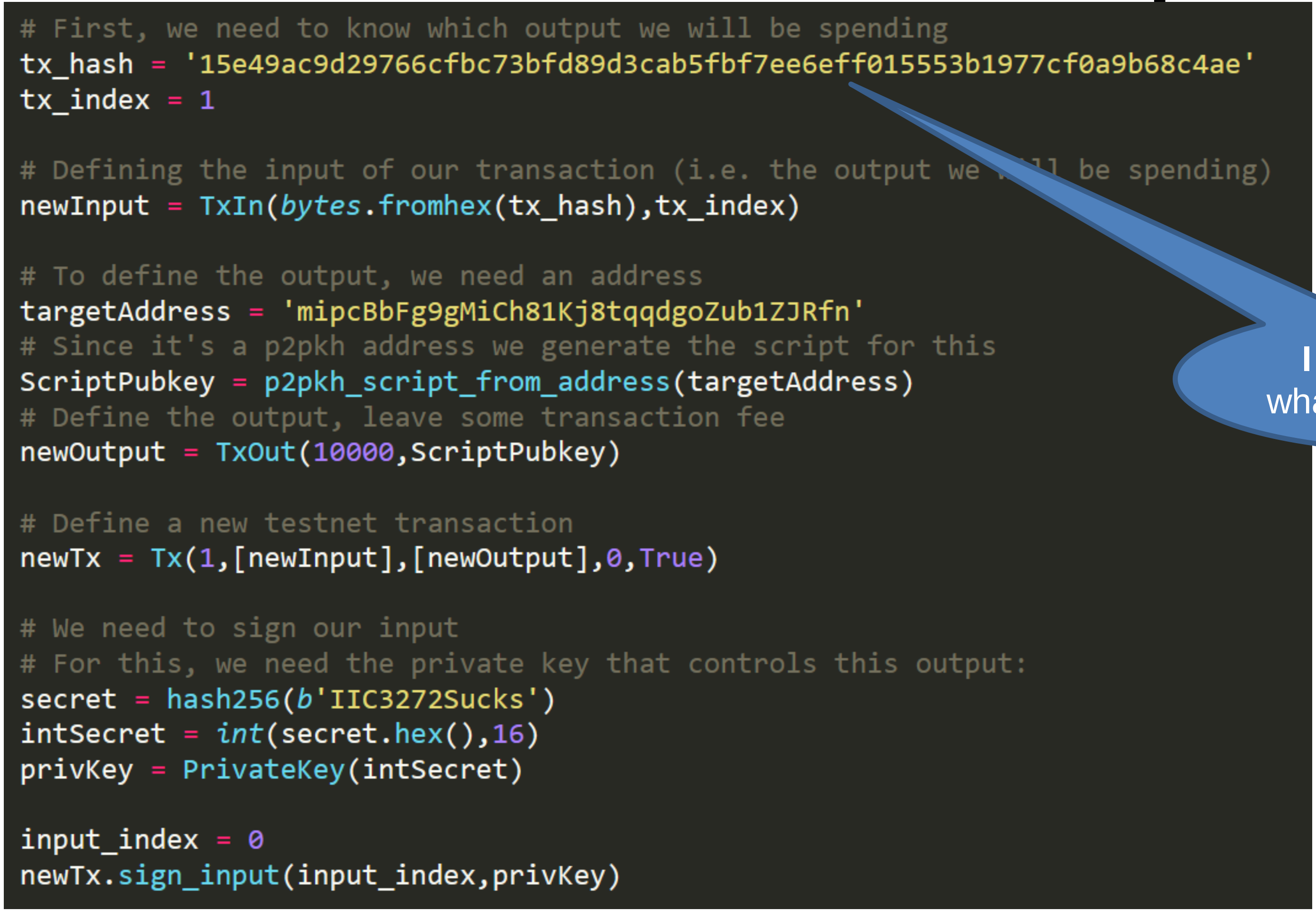
- Tells me: pay me to a P2PKH
- OP_DUP OP_HASH160 <hash> OP_EQUALVERIFY OP_CHECKSIG
- This will be the scriptPubKey to which we pay!!!



```python
# 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])
```

Spending money

P2PKH

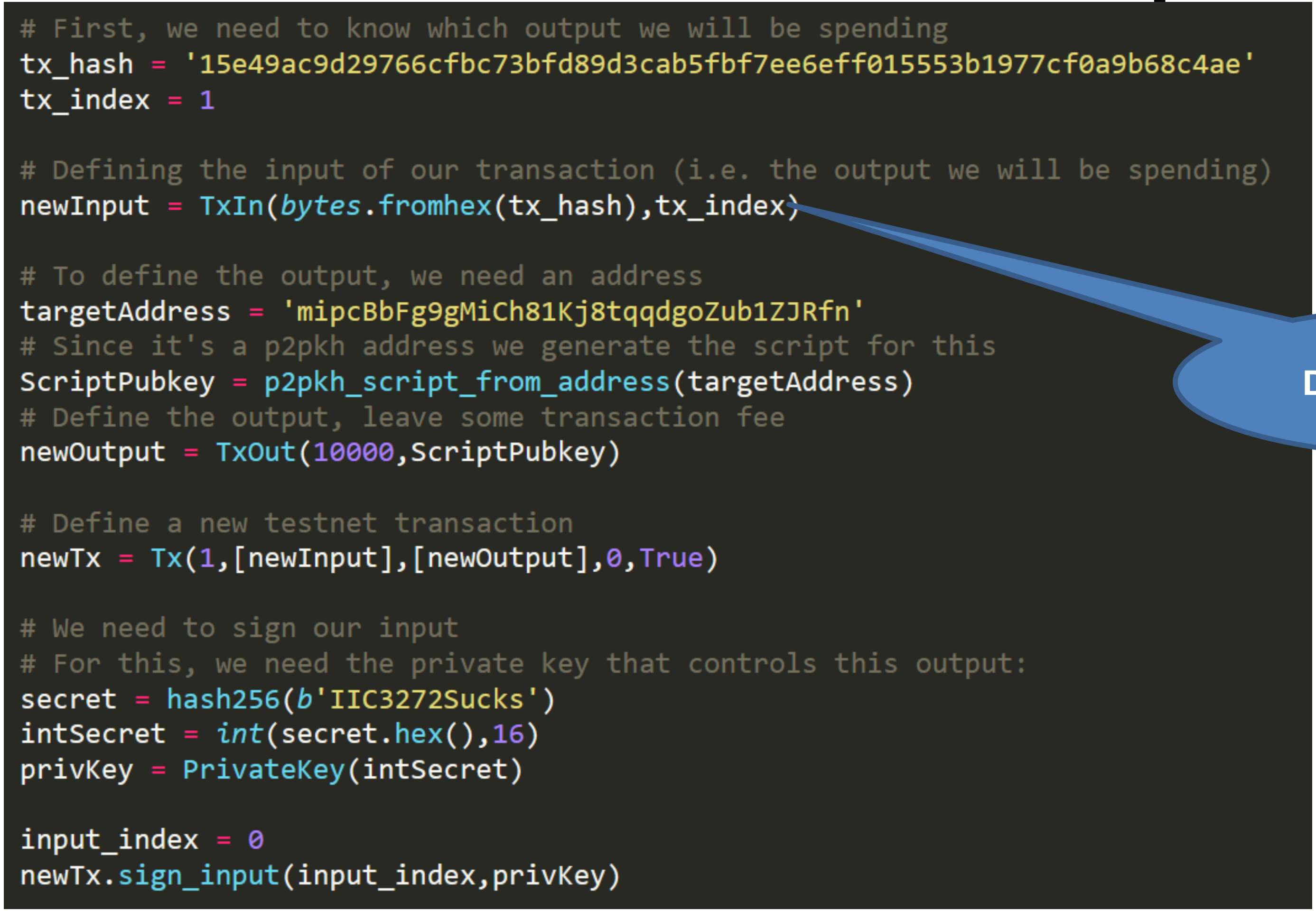


```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# 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)
```

I need to know
what am I spending!

Spending money

P2PKH

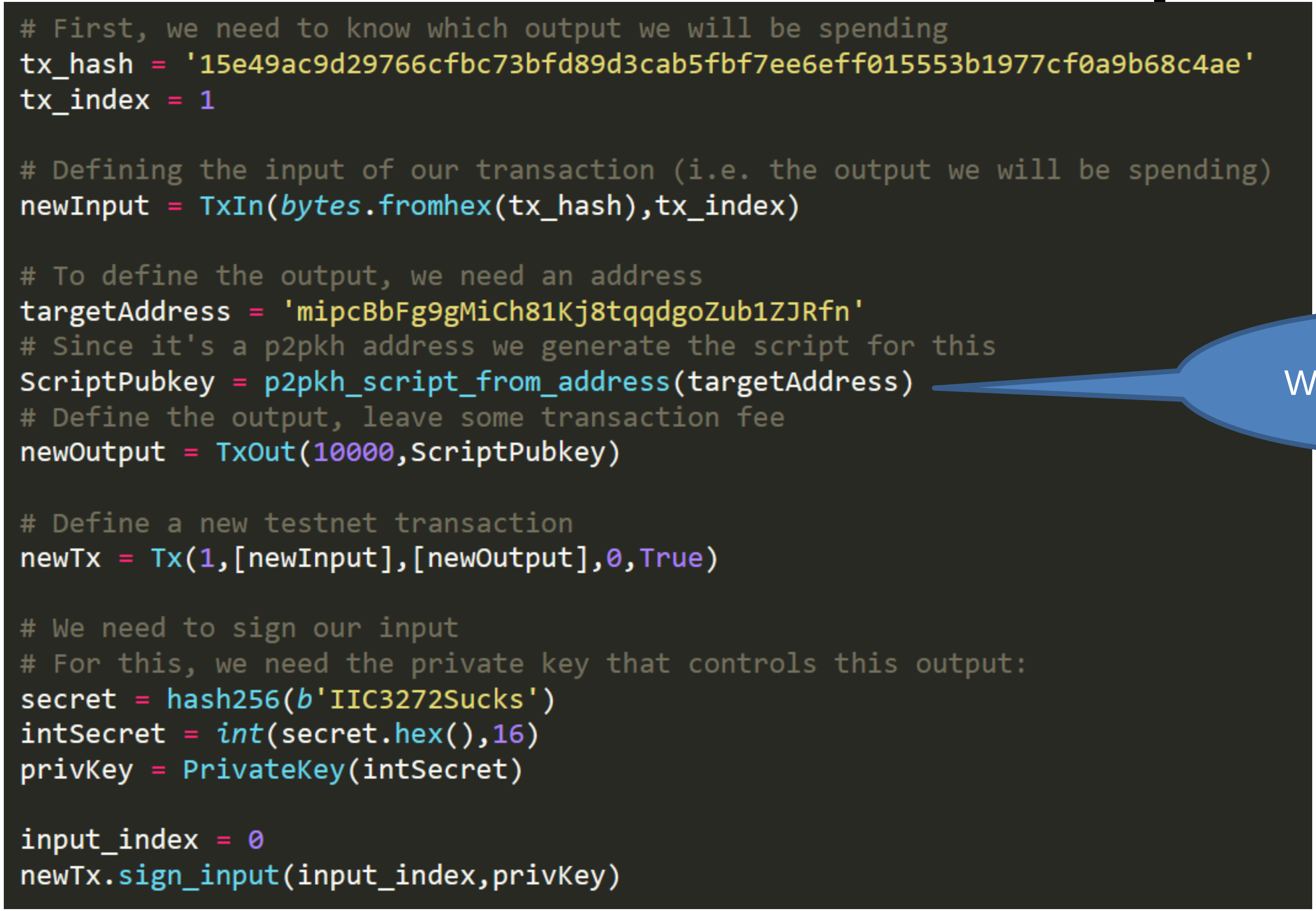


```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# 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)
```

Define this input!

Spending money

P2PKH

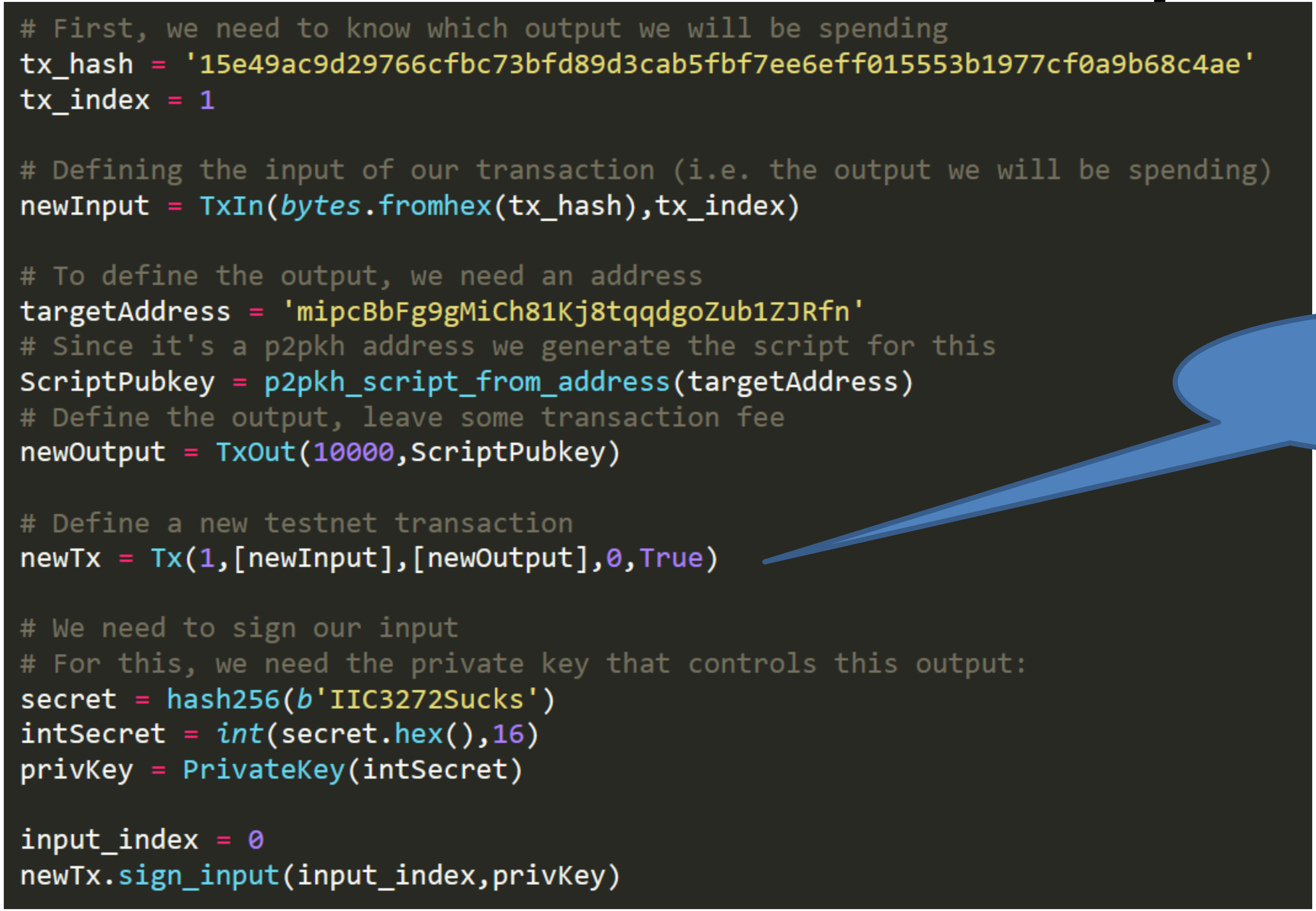


```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# 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)
```

Who am I paying to?

Spending money

P2PKH

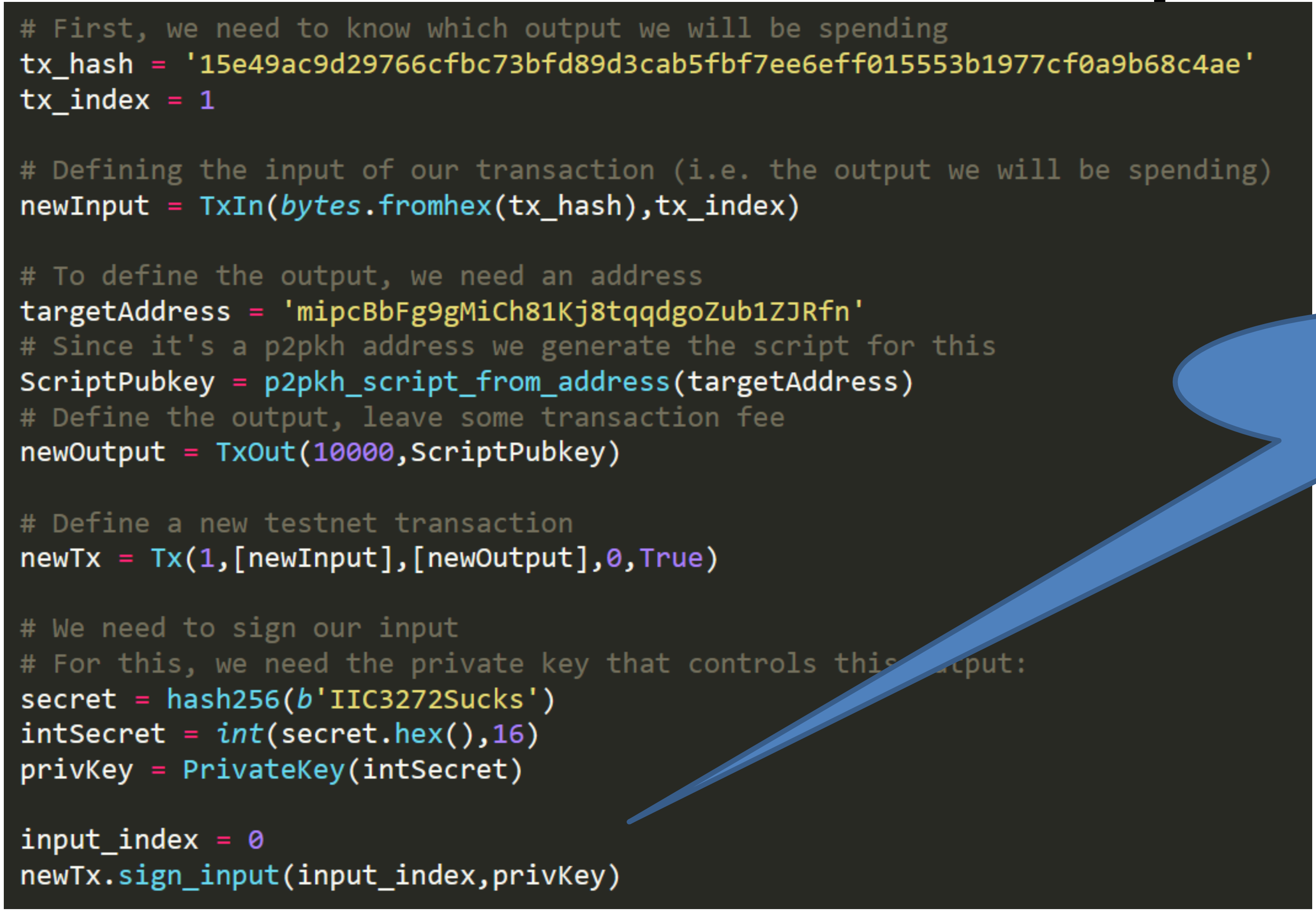


```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# 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)
```

I have my TX!

Spending money

P2PKH

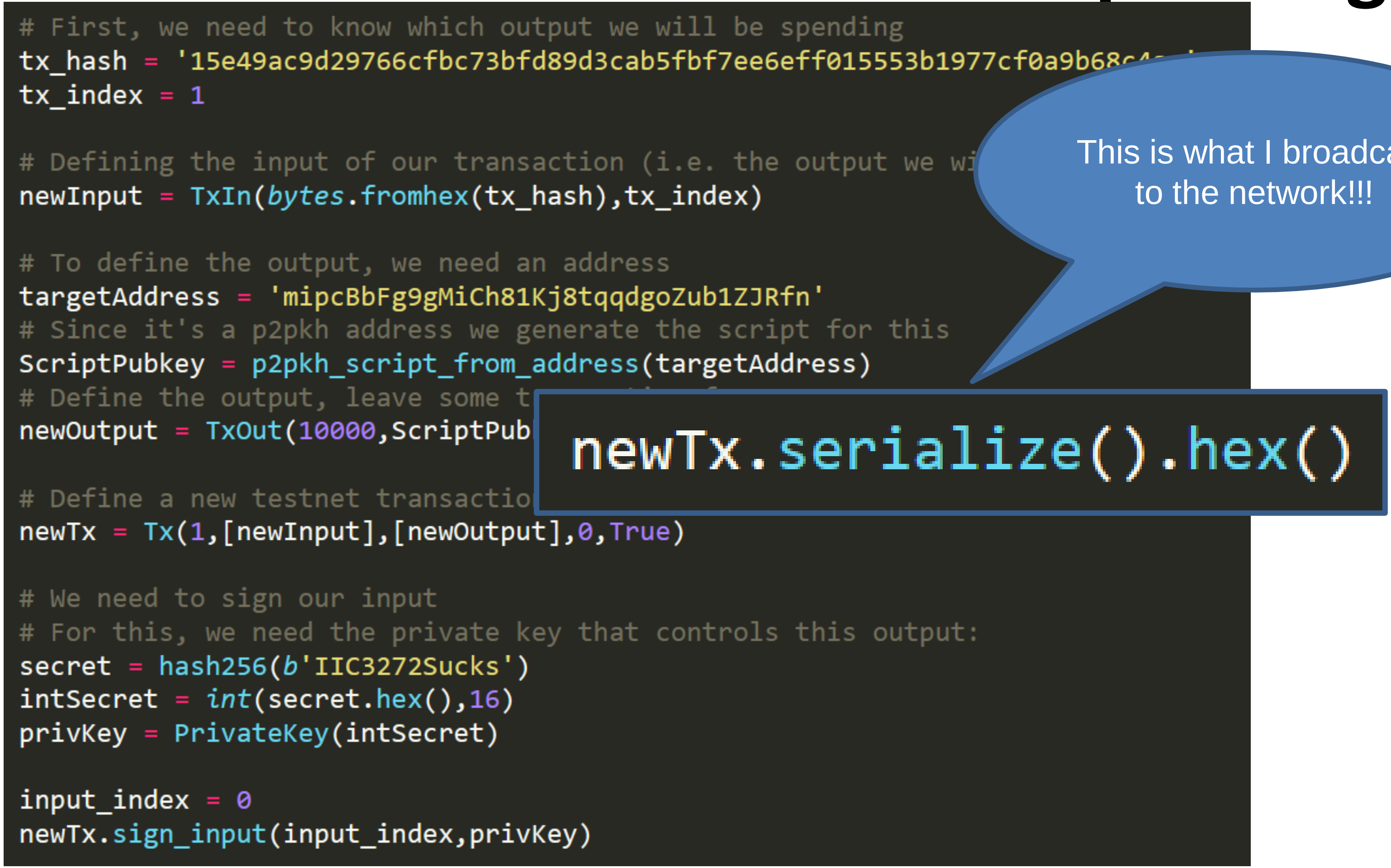


```python
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4ae'
tx_index = 1

# 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)
```

I sign my inputs!

Spending money

P2PKH



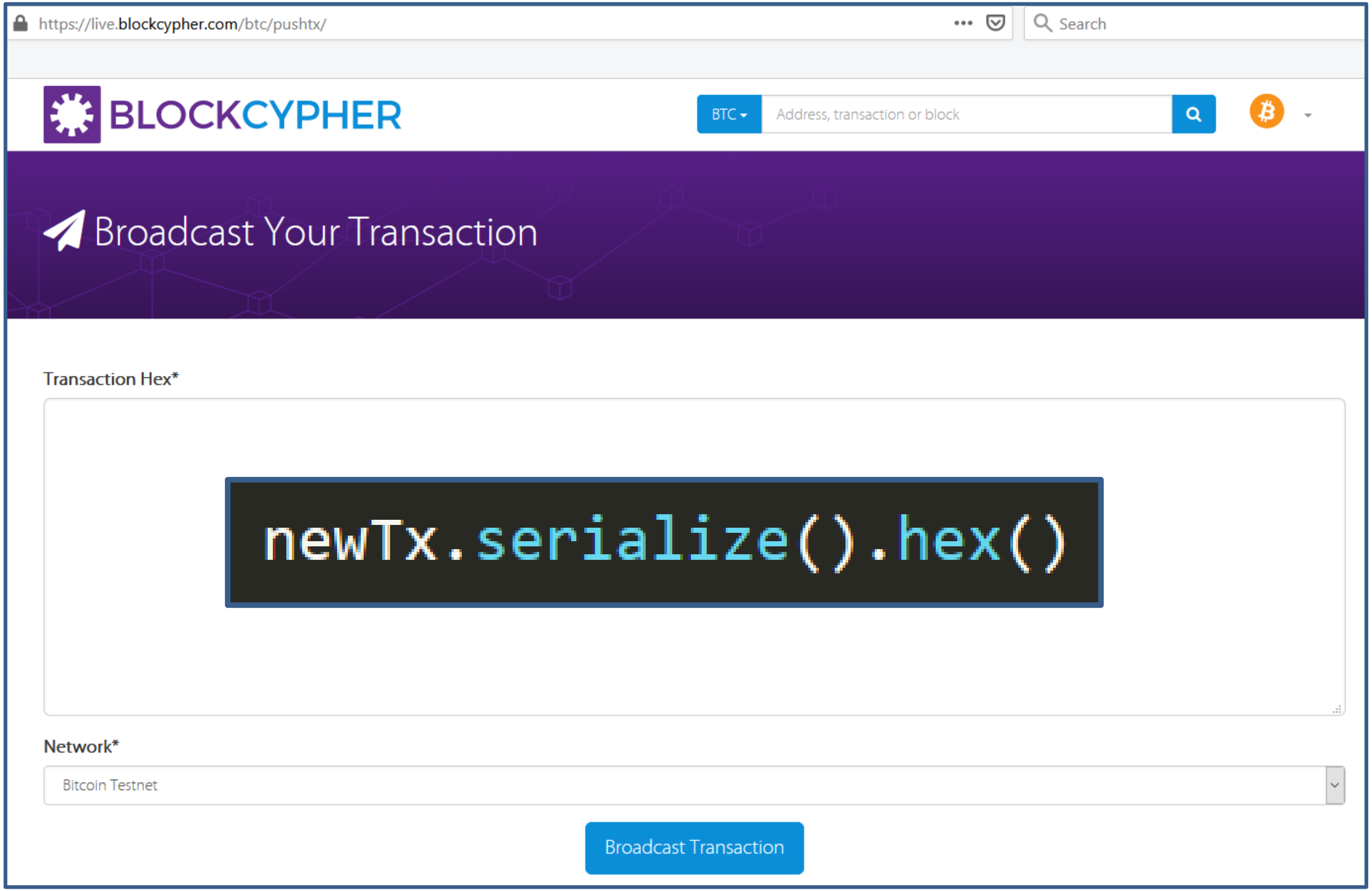
```
# First, we need to know which output we will be spending
tx_hash = '15e49ac9d29766cfbc73bfd89d3cab5fbf7ee6eff015553b1977cf0a9b68c4
tx_index = 1

# 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)
```

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

This is what I broadcast to the network!!!

Spending money

P2PKH



https://live.blockcypher.com/btc/pushtx/

Broadcast Your Transaction

Transaction Hex*

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

Network*

Bitcoin Testnet

Addresses

P2SH

An address with the prefix *0x05* or *0xce* (3,2 in base58):

- Tell me: pay me a P2SH
- OP_HASH160 <hash> OP_EQUAL

To which <hash>?

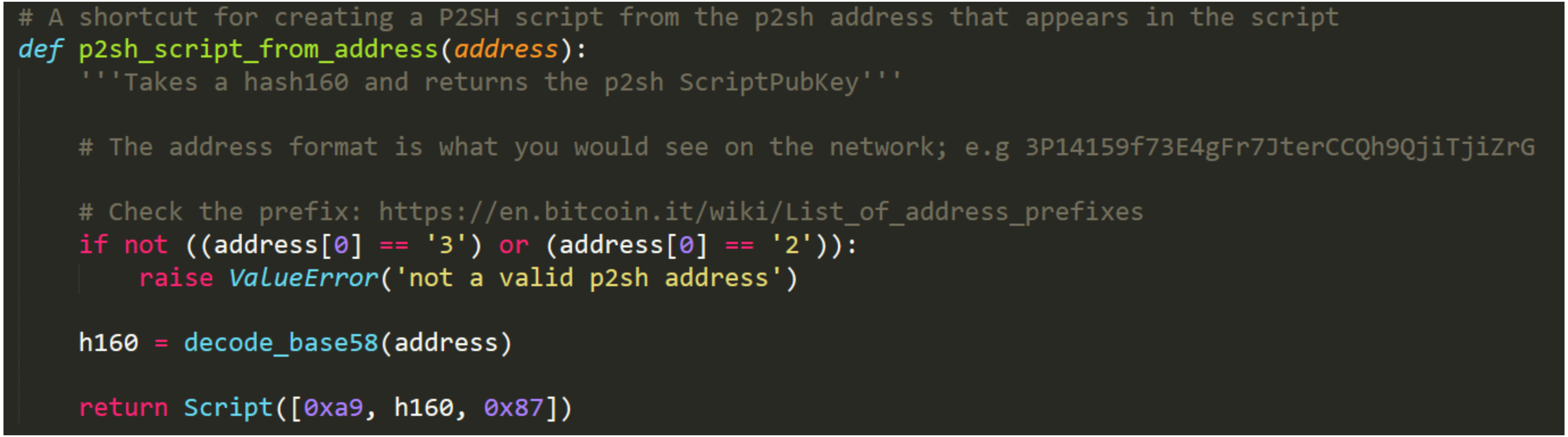
- D my P2SH address
- Convert D from base 58 to bytes
- Remove the prefix
- Remove the checksum
- The result is my <hash>

Addresses

P2SH

An address with the prefix *0x05* or *0xce* (3,2 in base58 mainnet/testnet):

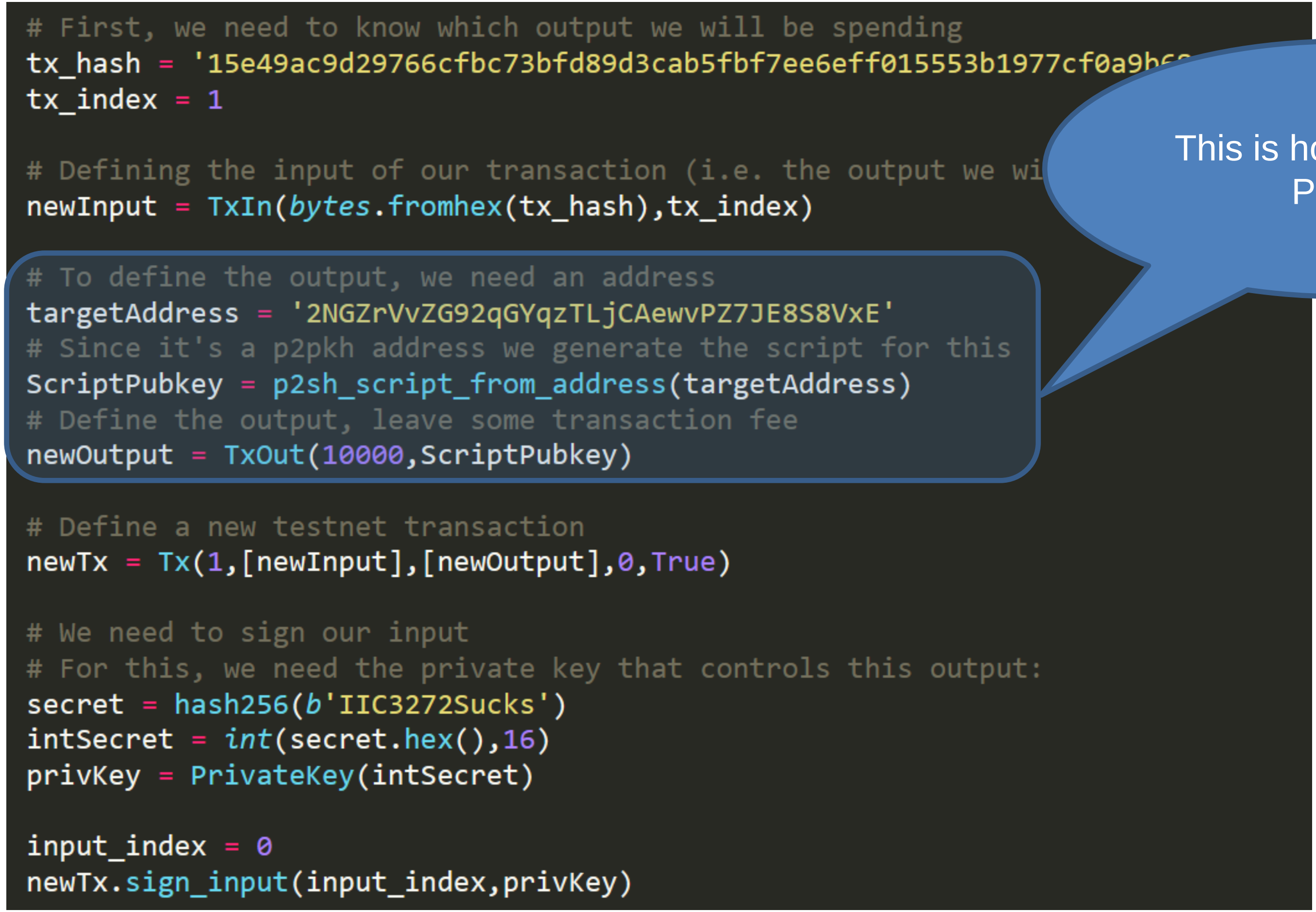
- Tell me: pay me a P2SH
- OP_HASH160 <hash> OP_EQUAL
- Pay to this script:



```python
# 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])
```

Spending money

P2SH



```python
# 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)
```

This is how i pay to a P2SH!

Addresses

P2SH

How to generate a P2SH address?

- From my redeem script (its serialization in bytes)
- **I need to have my redeem script!!!** (o be able to reconstruct it)

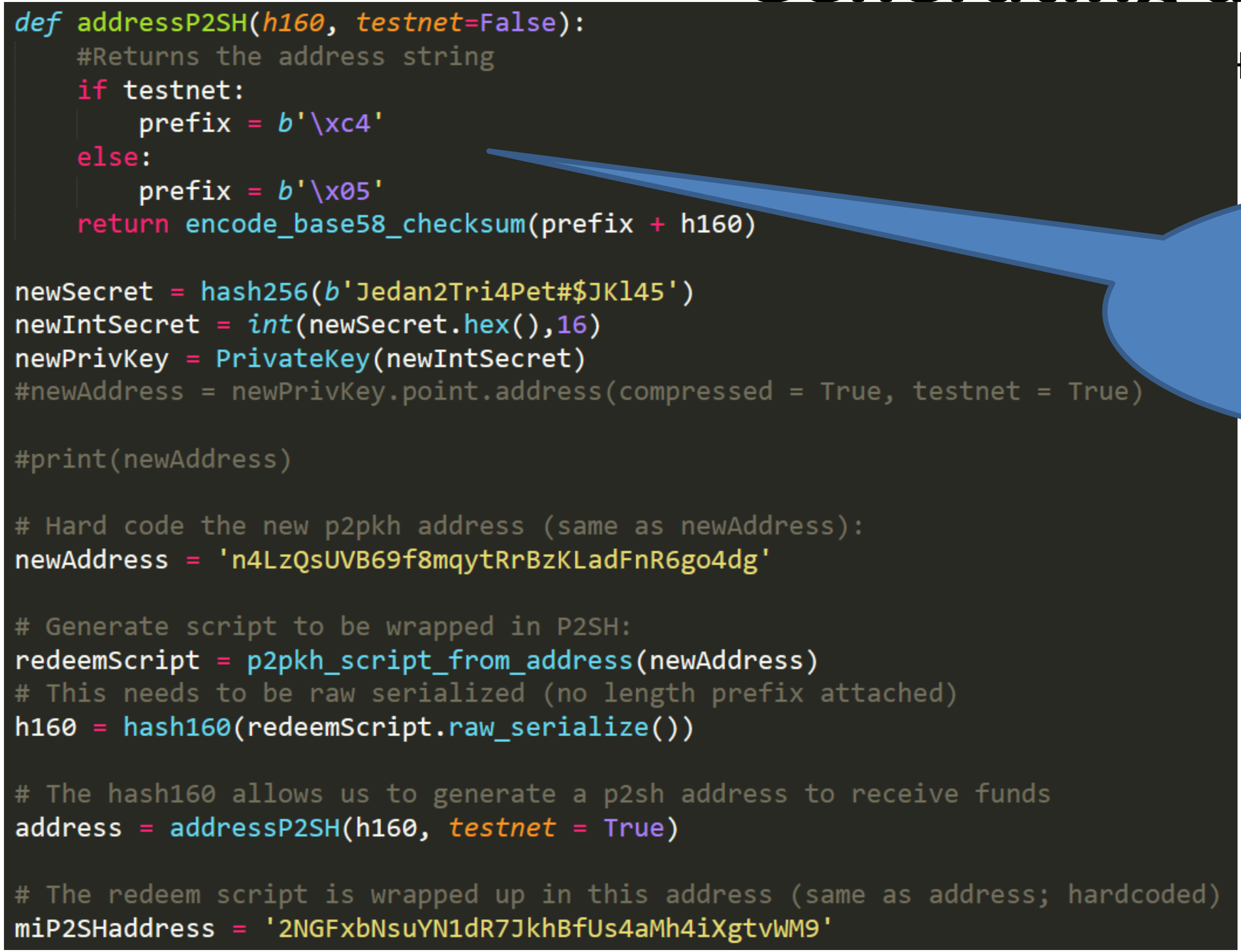
How to spend my P2SH output?

- With the redeem script
- And with the unlocking script for this redeem script
- The signature is different (scriptSig is rplaced by the redeem script)

Let’s see this in detail!

Generating a P2SH address

How to generate a P2SH address?

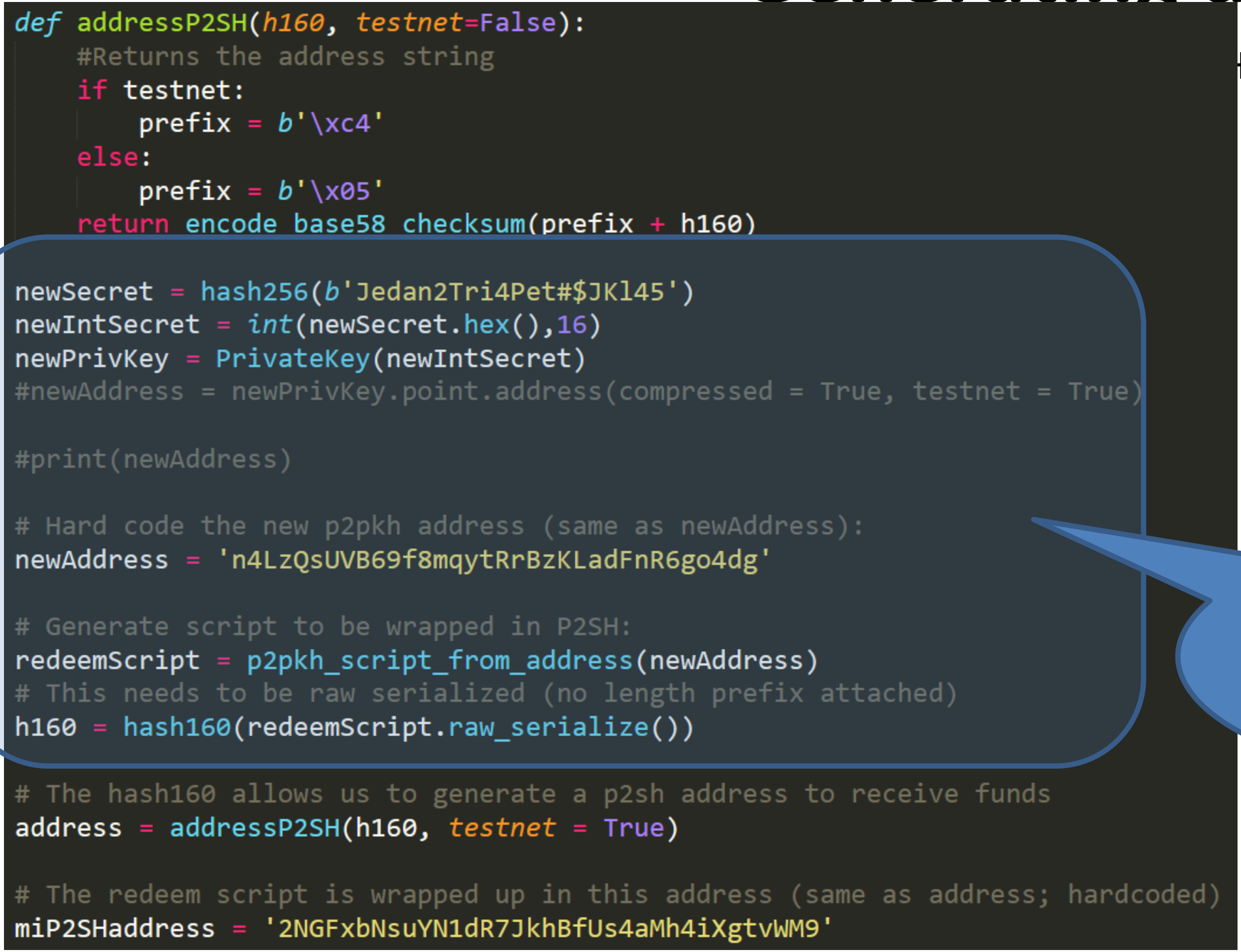


```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

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'
```

This generates an address from a h160

Generating a P2SH address

How to generate a P2SH address?

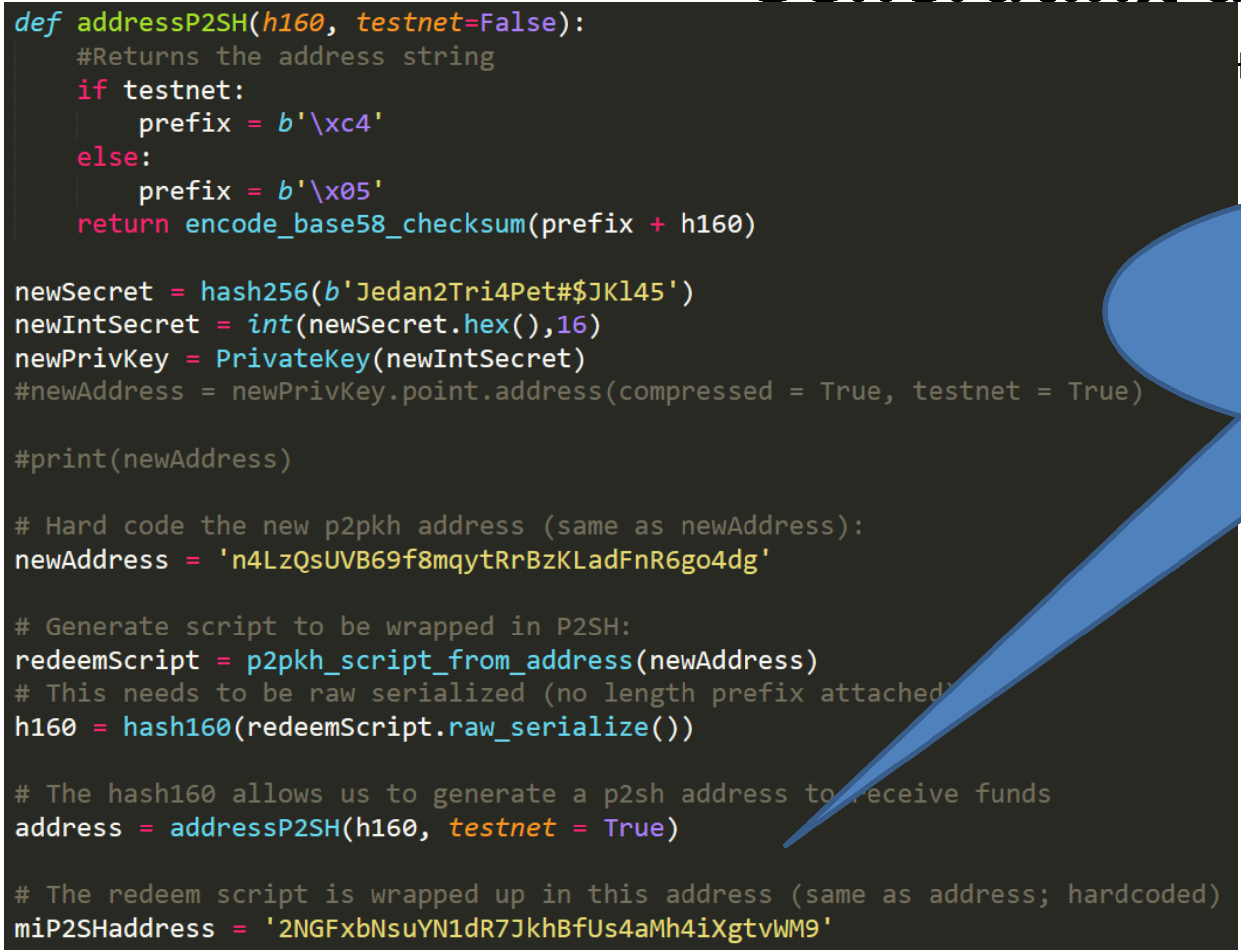


```
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

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'
```

First we generate the redeem script

Generating a P2SH address

How to generate a P2SH address?

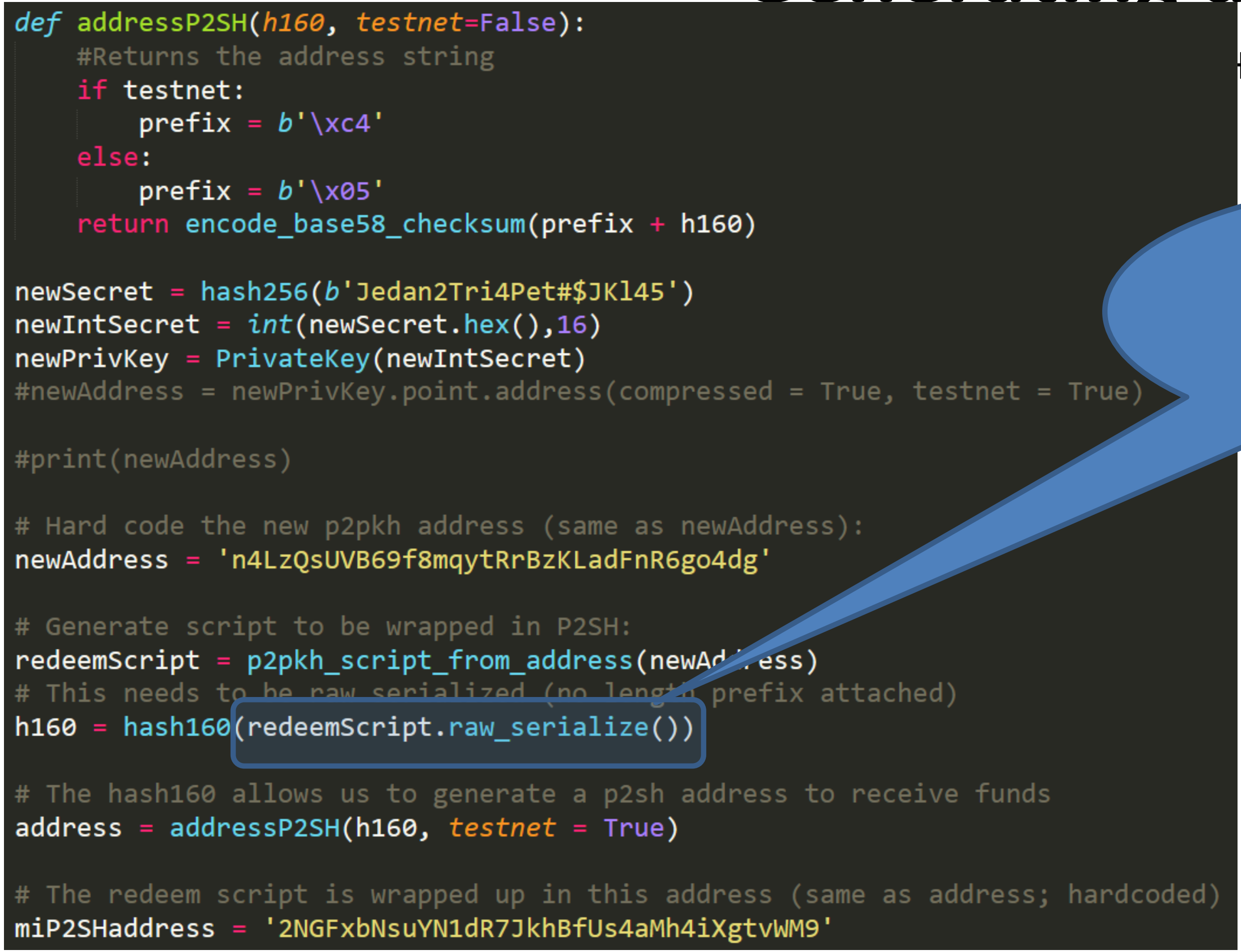


```python
def addressP2SH(h160, testnet=False):
    #Returns the address string
    if testnet:
        prefix = b'\xc4'
    else:
        prefix = b'\x05'
    return encode_base58_checksum(prefix + h160)

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'
```

With this we generate a P2SH address

Generating a P2SH address

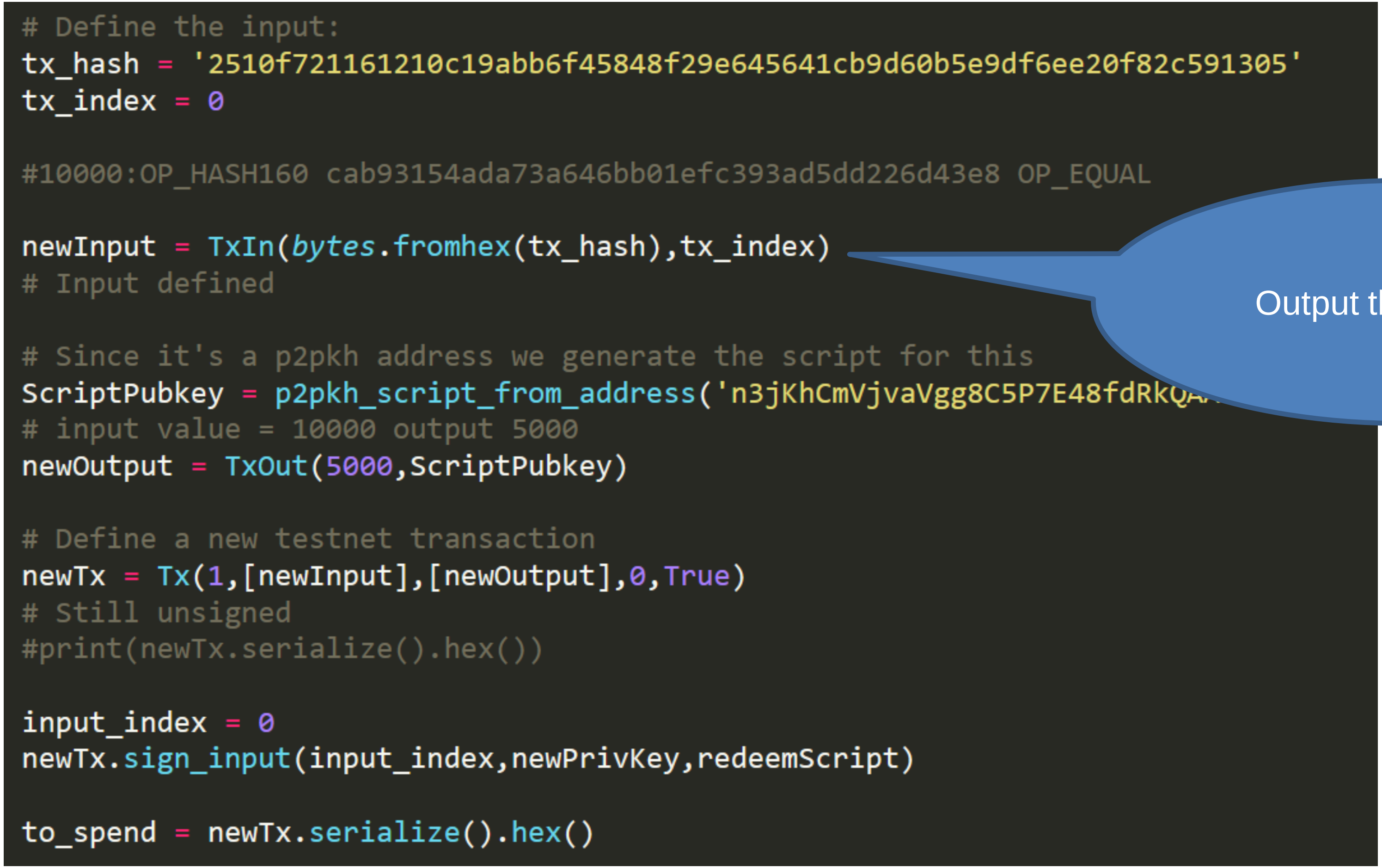
How to generate a P2SH address?



```
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'
```

IMPORTANT:
raw_serialize()
i.e. without len(script)

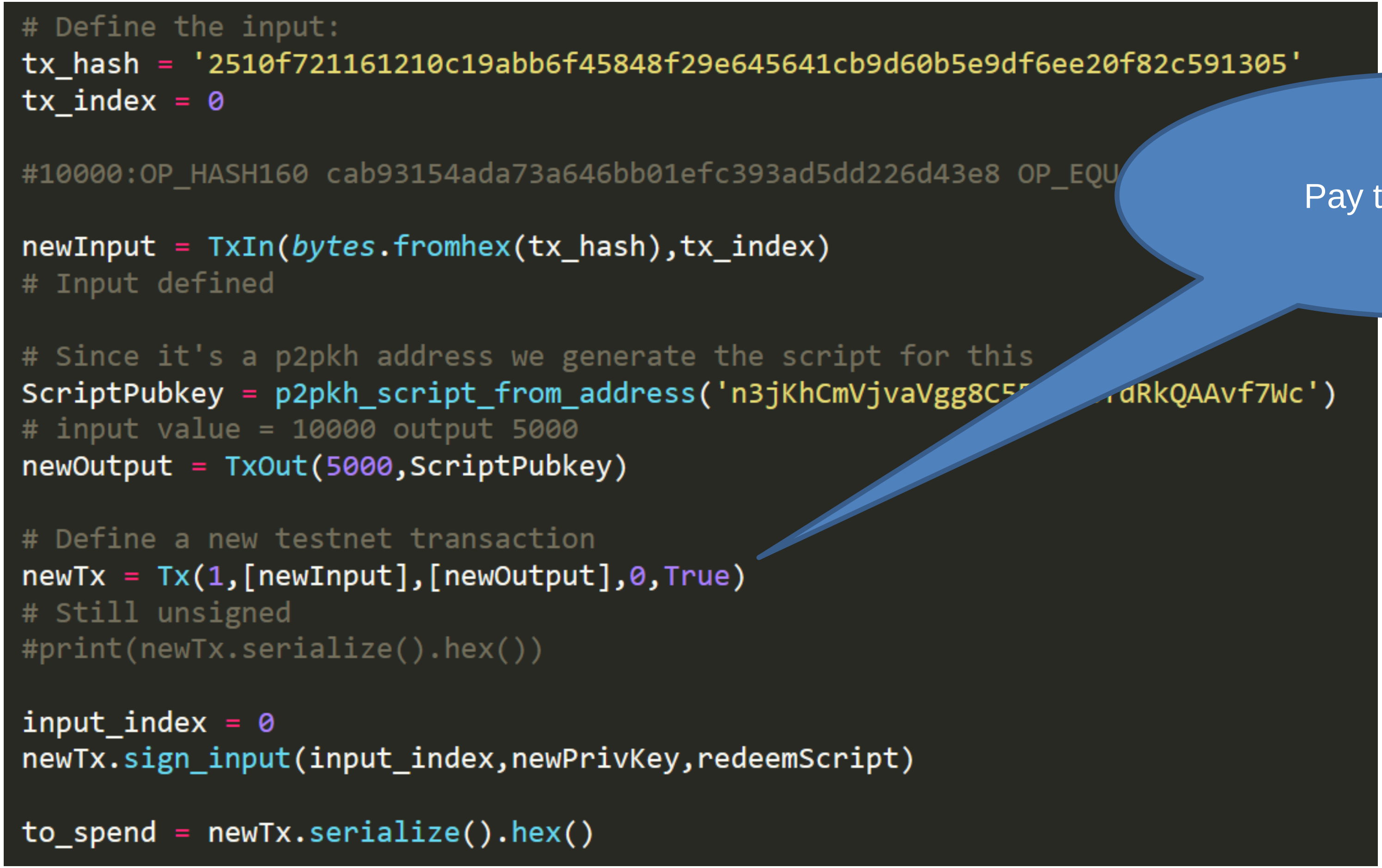
Spend a P2SH output



```
# 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()
```

Output that is spent

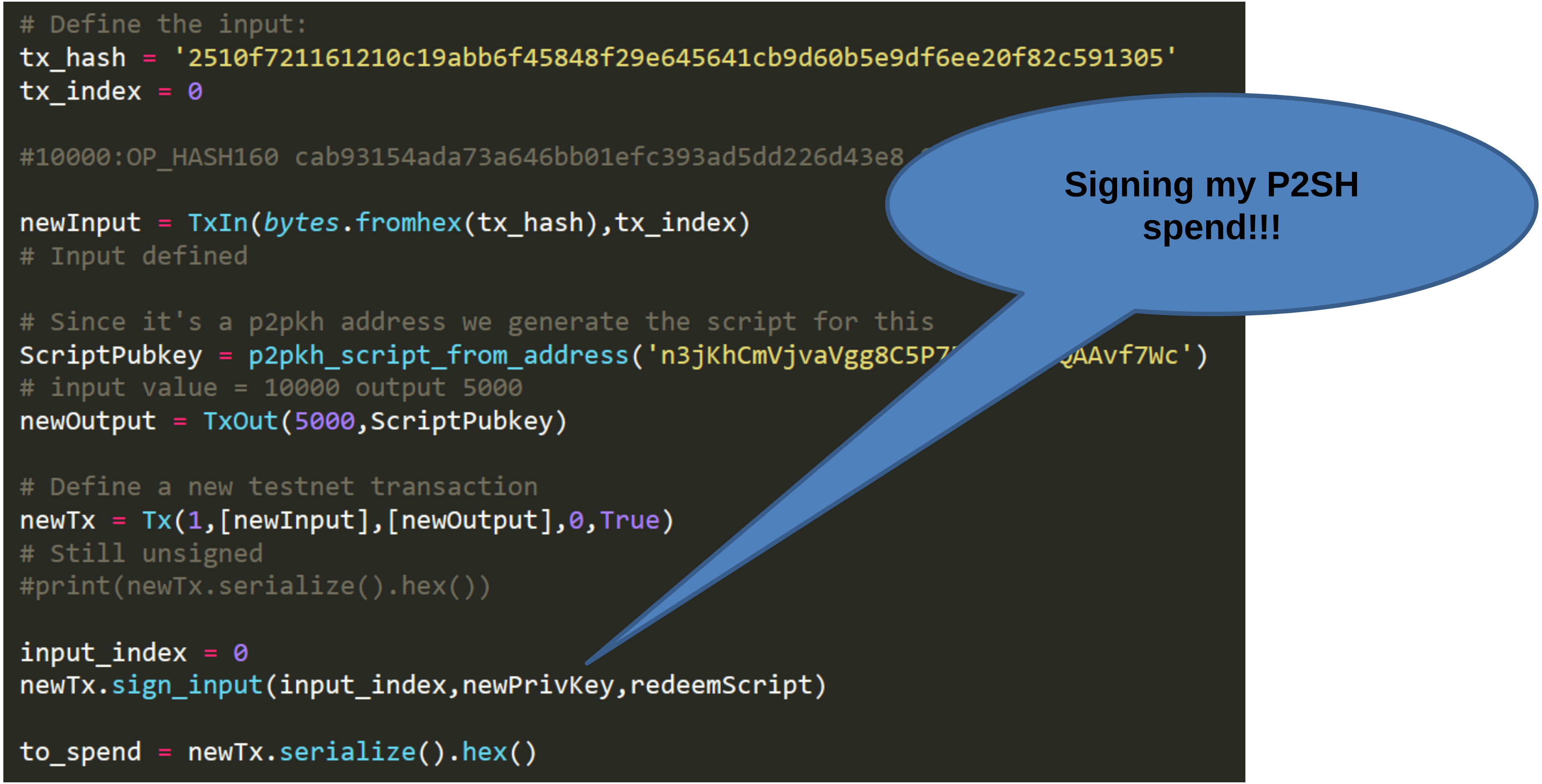
Spend a P2SH output



```
# 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()
```

Pay to a P2PKH

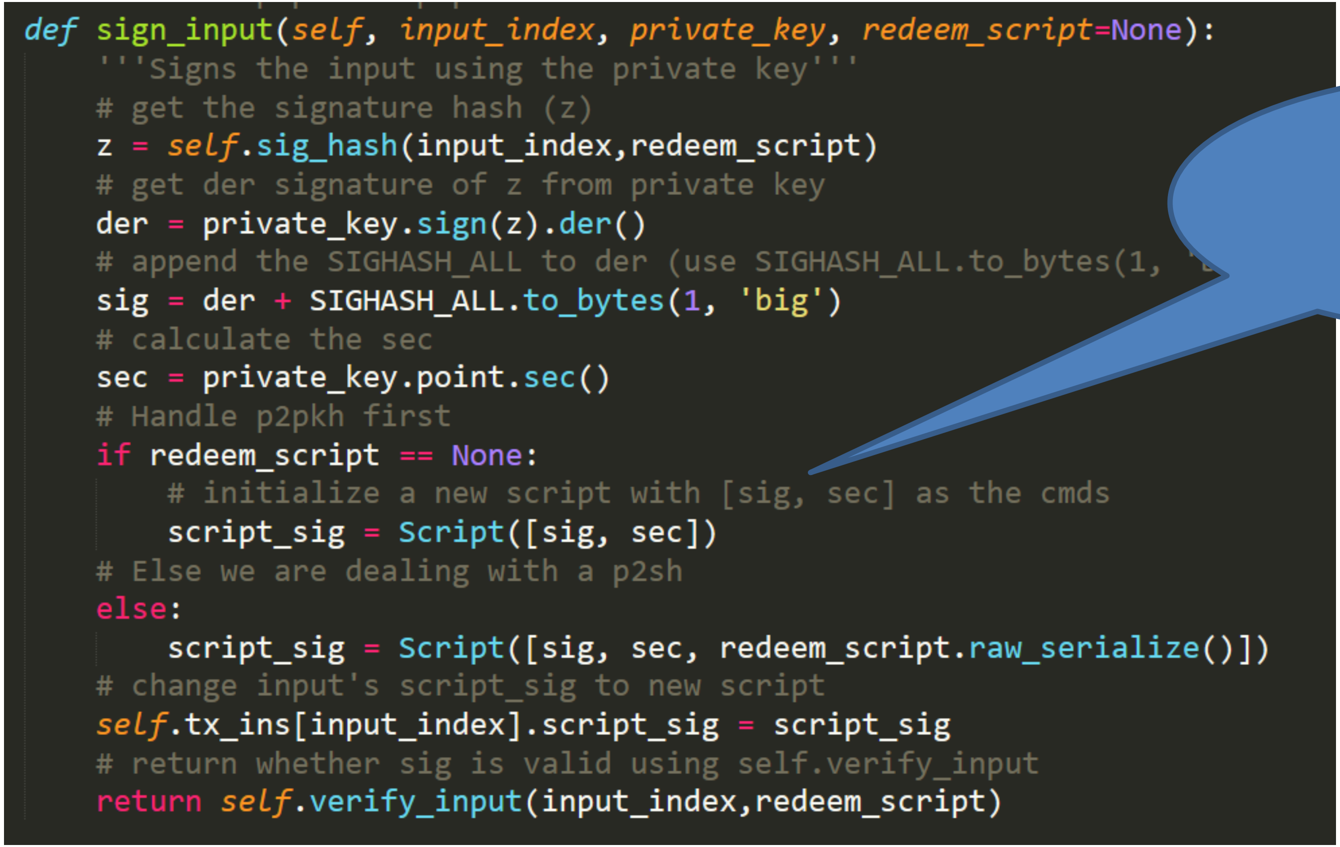
Spend a P2SH output



```
# 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()
```

Signing my P2SH spend!!!

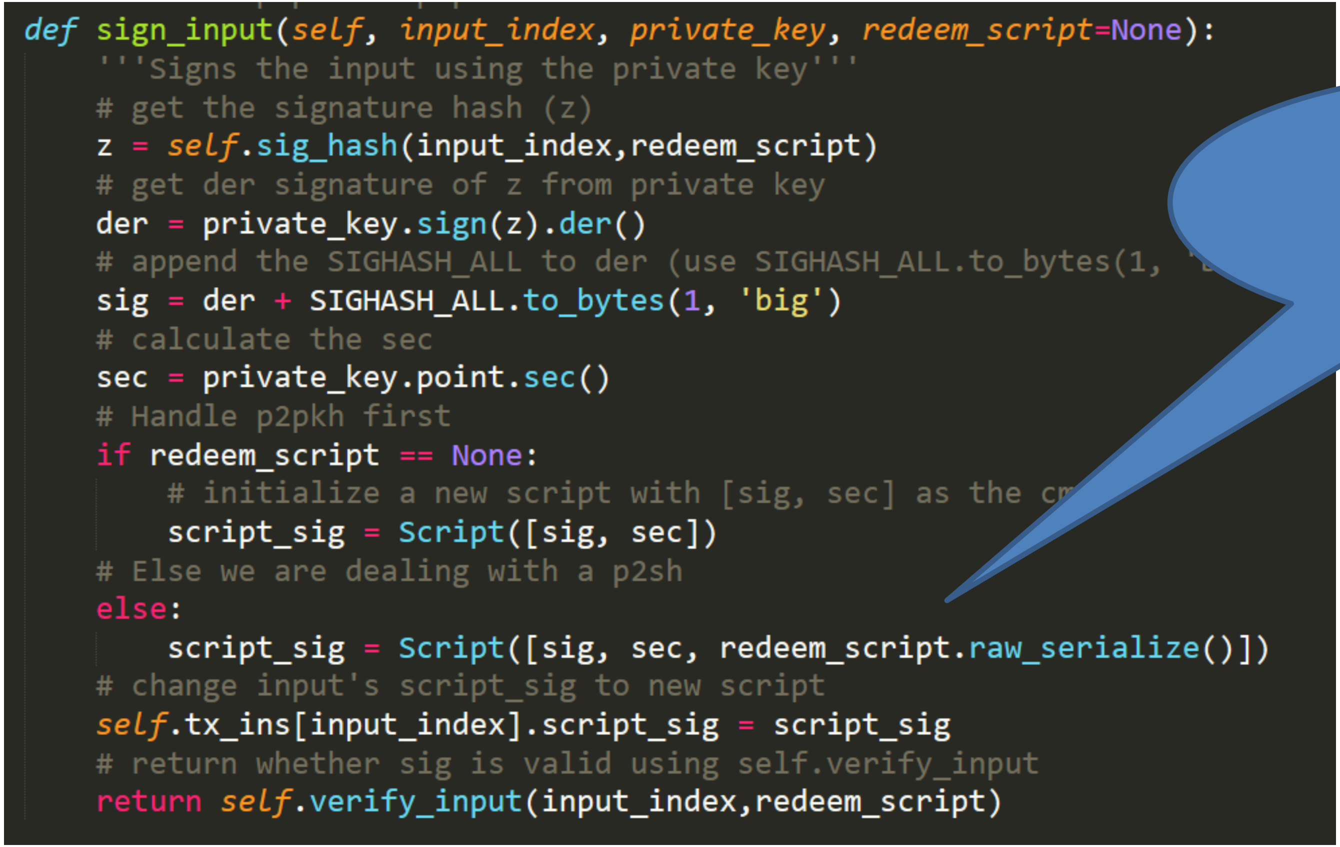
Spend a P2SH output



```
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, 'b
    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)
```

P2PKH

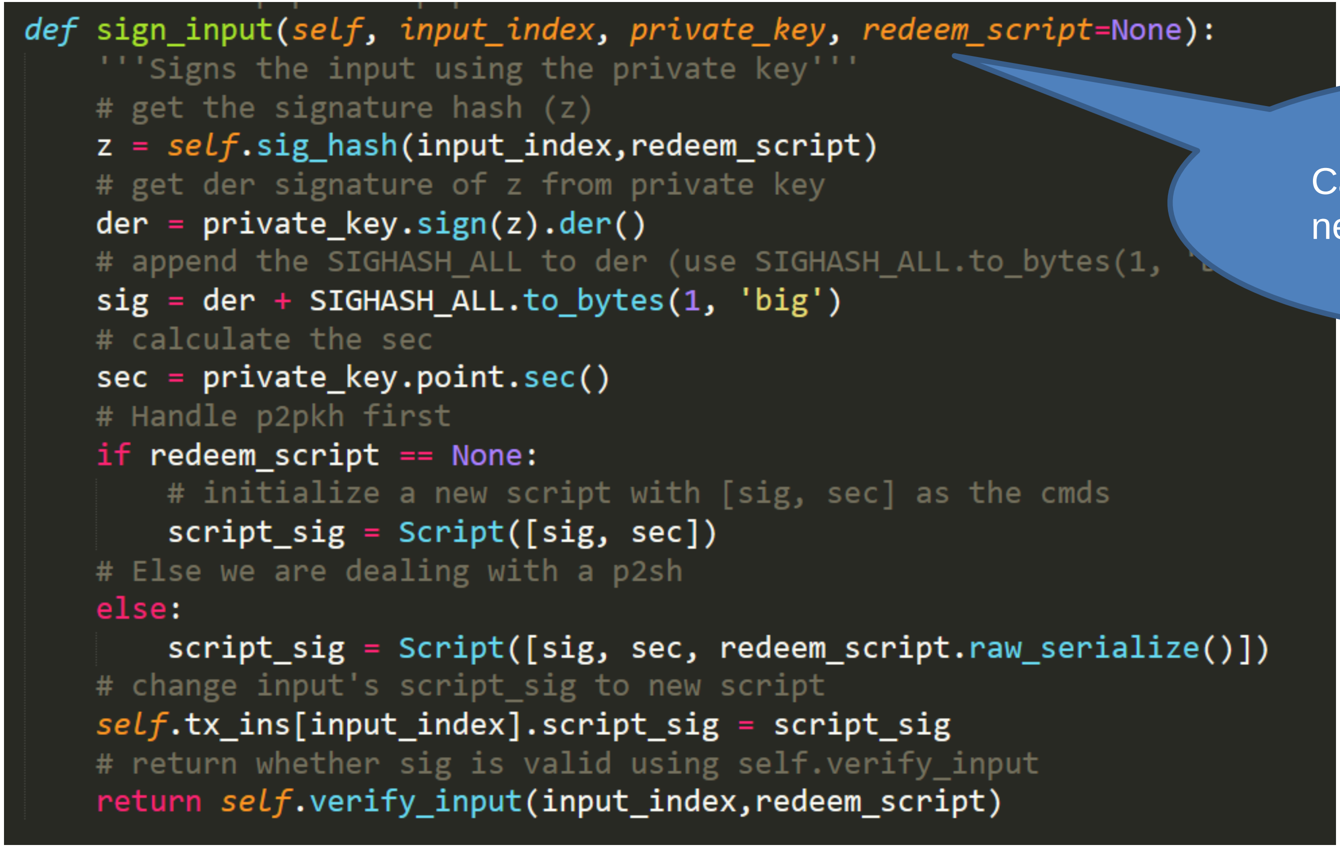
Spend a P2SH output



```python
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)
```

P2SH

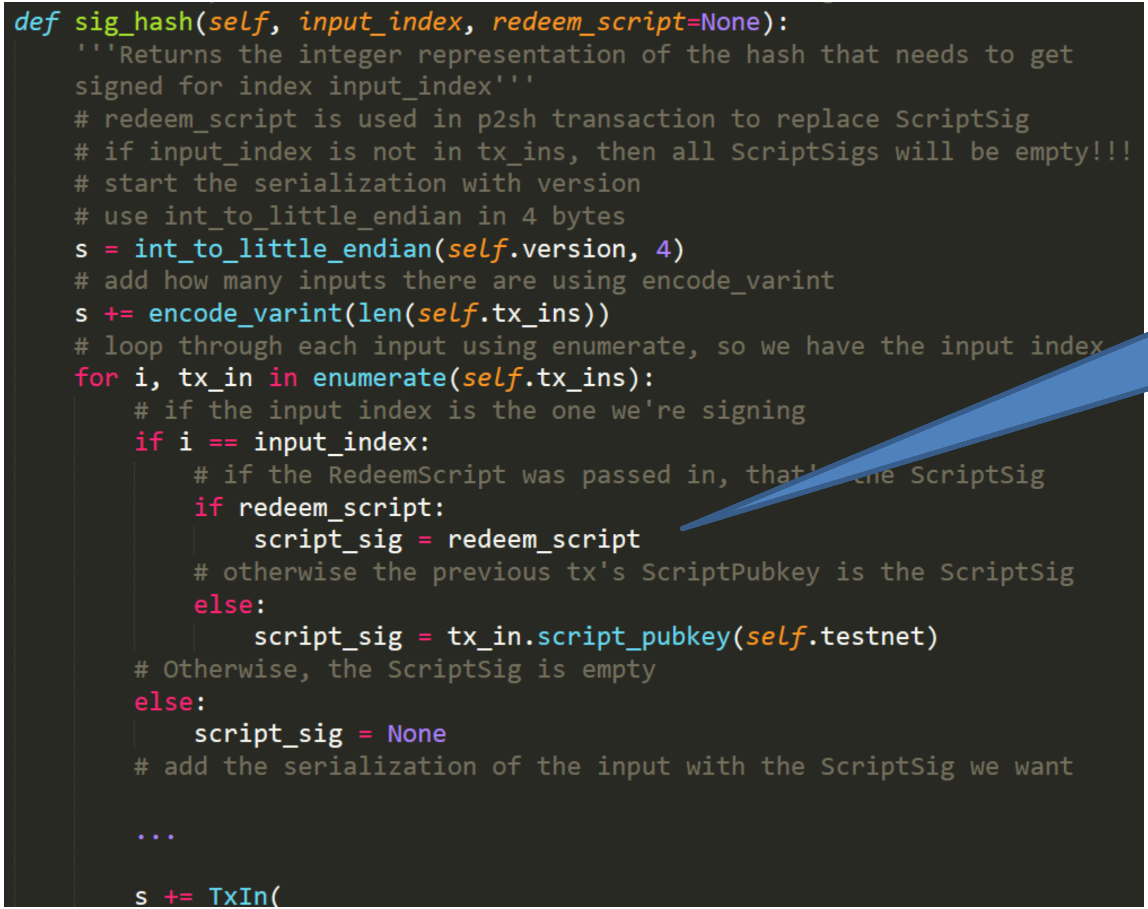
Spend a P2SH output



```python
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)
```

Careful: for P2SH I also need the redeem_script

Spend a P2SH output



```python
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(
```

For P2SH ScriptSig is replaced by redeem_script

Important

P2SH

The methods we have in tx.py allow us to:

- Create a P2SH spend

The methods we have in tx.py do **not** allow us:

- To validate a P2SH script (correctly)
- If you are bothered with this you can implement this in full!!!

References

- Jimmy Song, Programming Bitcoin, chapters 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