---

title: "Vyper ERC-721 Contract Walkthrough" description: Ryuya Nakamura's ERC-721 contract and how it works author: Ori Pomerantz lang: en tags: ["vyper", "erc-721", "python"] skill: beginner published: 2021-04-01

Introduction {#introduction}

The ERC-721 standard is used to hold the ownership of Non-Fungible Tokens (NFT). ERC-20 tokens behave as a commodity, because there is no difference between individual tokens. In contrast to that, ERC-721 tokens are designed for assets that are similar but not identical, such as different cat cartoons or titles to different pieces of real estate.

In this article we will analyze Ryuya Nakamura's ERC-721 contract. This contract is written in Vyper, a Python-like contract language designed to make it harder to write insecure code than it is in Solidity.

The Contract {#contract}

```python

# @dev Implementation of ERC-721 non-fungible token standard.

# @author Ryuya Nakamura (@nrryuya)

# Modified from: https://github.com/vyperlang/vyper/blob/de74722bf2d8718cca46902be165f9fe0e3641dd/examp

```

Comments in Vyper, as in Python, start with a hash (#) and continue to the end of the line. Comments that include `@<keyword>` are used by NatSpec to produce human-readable documentation.

```python from vyper.interfaces import ERC721

implements: ERC721 ```

The ERC-721 interface is built into the Vyper language. You can see the code definition here. The interface definition is written in Python, rather than Vyper, because interfaces are used not only within the blockchain, but also when sending the blockchain a transaction from an external client, which may be written in Python.

The first line imports the interface, and the second specifies that we are implementing it here.

#### The ERC721Receiver Interface {#receiver-interface}

```python

# Interface for the contract called by safeTransferFrom()

interface ERC721Receiver: def onERC721Received( ```

ERC-721 supports two types of transfer:

- `transferFrom`, which lets the sender specify any destination address and places the responsibility for the transfer on the sender. This means that you can transfer to an invalid address, in which case the NFT is lost for good.
- `safeTransferFrom`, which checks if the destination address is a contract. If so, the ERC-721 contract asks the receiving contract if it wants to receive the NFT.

To answer `safeTransferFrom` requests a receiving contract has to implement `ERC721Receiver`.

```
python _operator: address, _from: address,
```

The `_from` address is the current owner of the token. The `_operator` address is the one that requested the transfer (those two may not be the same, because of allowances).

```
python _tokenId: uint256,
```

ERC-721 token IDs are 256 bits. Typically they are created by hashing a description of whatever the token represents.

```
python _data: Bytes[1024]
```

The request can have up to 1024 bytes of user data.

```
python ) -> bytes32: view
```

To prevent cases in which a contract accidentally accepts a transfer the return value is not a boolean, but 256 bits with a specific value.

This function is a `view`, which means it can read the state of the blockchain, but not modify it.

Events {#events}

Events are emitted to inform users and servers outside of the blockchain of events. Note that the content of events is not available to contracts on the blockchain.

```python

# @dev Emits when ownership of any NFT changes by any mechanism. This event emits when NFTs are

# created (`from` == 0) and destroyed (`to` == 0). Exception: during contract creation, any

# number of NFTs may be created and assigned without emitting Transfer. At the time of any

# transfer, the approved address for that NFT (if any) is reset to none.

# @param _from Sender of NFT (if address is zero address it indicates token creation).
```

@param _to Receiver of NFT (if address is zero address it indicates token destruction).

@param _tokenId The NFT that got transferred.

event Transfer: sender: indexed(address) receiver: indexed(address) tokenId: indexed(uint256) ```

This is similar to the ERC-20 Transfer event, except that we report a`tokenId` instead of an amount. Nobody owns address zero, so by convention we use it to report creation and destruction of tokens.

```python

## @dev This emits when the approved address for an NFT is changed or reaffirmed. The zero address indicates there is no approved address. When a Transfer event emits, this also indicates that the approved address for that NFT (if any) is reset to none.

## @param _owner Owner of NFT.

## @param _approved Address that we are approving.

## @param _tokenId NFT which we are approving.

event Approval: owner: indexed(address) approved: indexed(address) tokenId: indexed(uint256) ```

An ERC-721 approval is similar to an ERC-20 allowance. A specific address is allowed to transfer a specific token. This gives a mechanism for contracts to respond when they accept a token. Contracts cannot listen for events, so if you just transfer the token to them they don't "know" about it. This way the owner first submits an approval and then sends a request to the contract: "I approved for you to transfer token X, please do ...".

This is a design choice to make the ERC-721 standard similar to the ERC-20 standard. Because ERC-721 tokens are not fungible, a contract can also identify that it got a specific token by looking at the token's ownership.

```python

## @dev This emits when an operator is enabled or disabled for an owner. The operator can manage

## all NFTs of the owner.

## @param _owner Owner of NFT.

## @param _operator Address to which we are setting operator rights.

## @param _approved Status of operator rights(true if operator rights are given and false if revoked).

event ApprovalForAll: owner: indexed(address) operator: indexed(address) approved: bool ```

It is sometimes useful to have an *operator* that can manage all of an account's tokens of a specific type (those that are managed by a specific contract), similar to a power of attorney. For example, I might want to give such a power to a contract that checks if I haven't contacted it for six months, and if so distributes my assets to my heirs (if one of them asks for it, contracts can't do anything without being called by a transaction). In ERC-20 we can just give a high allowance to an inheritance contract, but that doesn't work for ERC-721 because the tokens are not fungible. This is the equivalent.

The `approved` value tells us whether the event is for an approval, or the withdrawal of an approval.

**State Variables {#state-vars}**

These variables contain the current state of the tokens: which ones are available and who owns them. Most of these are`HashMap` objects, [unidirectional mappings that exist between two types](

```python

## @dev Mapping from NFT ID to the address that owns it.

idToOwner: HashMap[uint256, address]

## @dev Mapping from NFT ID to approved address.

idToApprovals: HashMap[uint256, address] ```

User and contract identities in Ethereum are represented by 160-bit addresses. These two variables map from token IDs to their owners and those approved to transfer them (at a maximum of one for each). In Ethereum, uninitialized data is always zero, so if there is no owner or approved transferor the value for that token is zero.

```python

## @dev Mapping from owner address to count of his tokens.

ownerToNFTokenCount: HashMap[address, uint256] ```

This variable holds the count of tokens for each owner. There is no mapping from owners to tokens, so the only way to identify the tokens that a specific owner owns is to look back in the blockchain's event history and see the appropriate `Transfer` events. We can use this variable to know when we have all the NFTs and don't need to look even further in time.

Note that this algorithm only works for user interfaces and external servers. Code running on the blockchain itself cannot read past events.

```python

# @dev Mapping from owner address to mapping of operator addresses.

ownerToOperators: HashMap[address, HashMap[address, bool]] ```

An account may have more than a single operator. A simple `HashMap` is insufficient to keep track of them, because each key leads to a single value. Instead, you can use `HashMap[address, bool]` as the value. By default the value for each address is `False`, which means it is not an operator. You can set values to `True` as needed.

```python

# @dev Address of minter, who can mint a token

minter: address ```

New tokens have to be created somehow. In this contract there is a single entity that is allowed to do so, the `minter`. This is likely to be sufficient for a game, for example. For other purposes, it might be necessary to create a more complicated business logic.

```python

# @dev Mapping of interface id to bool about whether or not it's supported

supportedInterfaces: HashMap[bytes32, bool]

# @dev ERC165 interface ID of ERC165

ERC165_INTERFACE_ID: constant(bytes32) = 0x0000000000000000000000000000000000000000000000000000000001ffc9a7

# @dev ERC165 interface ID of ERC721

ERC721_INTERFACE_ID: constant(bytes32) = 0x0000000000000000000000000000000000000000000000000000000080ac58cd ```

ERC-165 specifies a mechanism for a contract to disclose how applications can communicate with it, to which ERCs it conforms. In this case, the contract conforms to ERC-165 and ERC-721.

### Functions {#functions}

These are the functions that actually implement ERC-721.

#### Constructor {#constructor}

`python @external def __init__():`

In Vyper, as in Python, the constructor function is called `__init__`.

`python """ @dev Contract constructor. """`

In Python, and in Vyper, you can also create a comment by specifying a multi-line string (which starts and ends with `"""`), and not using it in any way. These comments can also include NatSpec.

`python self.supportedInterfaces[ERC165_INTERFACE_ID] = True self.supportedInterfaces[ERC721_INTERFACE_ID] = True self.minter = msg.sender`

To access state variables you use `self.&lt;variable name&gt;` (again, same as in Python).

#### View Functions {#views}

These are functions that do not modify the state of the blockchain, and therefore can be executed for free if they are called externally. If the view functions are called by a contract they still have to be executed on every node and therefore cost gas.

`python @view @external`

These keywords prior to a function definition that start with an at sign (`@`) are called *decorations*. They specify the circumstances in which a function can be called.

- `@view` specifies that this function is a view.
- `@external` specifies that this particular function can be called by transactions and by other contracts.

`python def supportsInterface(_interfaceID: bytes32) -> bool:`

In contrast to Python, Vyper is a static typed language. You can't declare a variable, or a function parameter, without identifying the data type. In this case the input parameter is `bytes32`, a 256-bit value (256 bits is the native word size of the Ethereum Virtual Machine). The output is a boolean value. By convention, the names of function parameters start with an underscore (`_`).

`python """ @dev Interface identification is specified in ERC-165. @param _interfaceID Id of the interface """ return self.supportedInterfaces[_interfaceID]`

Return the value from the `self.supportedInterfaces` HashMap, which is set in the constructor (`__init__`).

```python

### VIEW FUNCTIONS

```

These are the view functions that make information about the tokens available to users and other contracts.

`python @view @external def balanceOf(_owner: address) -> uint256: """ @dev Returns the number of NFTs owned by `_owner`. Throws if `_owner` is the zero address. NFTs assigned to the zero address are considered invalid. @param _owner Address for whom to query the balance. """ assert _owner != ZERO_ADDRESS`

This line asserts that `_owner` is not zero. If it is, there is an error and the operation is reverted.

```python return self.ownerToNFTokenCount[_owner]

@view @external def ownerOf(_tokenId: uint256) -> address: """ @dev Returns the address of the owner of the NFT. Throws if `_tokenId` is not a valid NFT. @param _tokenId The identifier for an NFT. """ owner: address = self.idToOwner[_tokenId] # Throws if `_tokenId` is not a valid NFT assert owner != ZERO_ADDRESS return owner ```

In the Ethereum Virtual Machine (evm) any storage that does not have a value stored in it is zero. If there is no token at `_tokenId` then the value of `self.idToOwner[_tokenId]` is zero. In that case the function reverts.

```
python @view @external def getApproved(_tokenId: uint256) -> address: """ @dev Get the approved address for a single NFT. Throws if `_tokenId` is not a valid NFT. @param _tokenId ID of the NFT to query the approval of. """ # Throws if `_tokenId` is not a valid NFT assert self.idToOwner[_tokenId] != ZERO_ADDRESS return self.idToApprovals[_tokenId]
```

Note that `getApproved` *can* return zero. If the token is valid it returns `self.idToApprovals[_tokenId]`. If there is no approver that value is zero.

```
python @view @external def isApprovedForAll(_owner: address, _operator: address) -> bool: """ @dev Checks if `_operator` is an approved operator for `_owner`. @param _owner The address that owns the NFTs. @param _operator The address that acts on behalf of the owner. """ return (self.ownerToOperators[_owner])[_operator]
```

This function checks if `_operator` is allowed to manage all of `_owner`'s tokens in this contract. Because there can be multiple operators, this is a two level HashMap.

Transfer Helper Functions {#transfer-helpers}

These functions implement operations that are part of transferring or managing tokens.

```python

### TRANSFER FUNCTION HELPERS

@view @internal ```

This decoration, `@internal`, means that the function is only accessible from other functions within the same contract. By convention, these function names also start with an underscore (`_`).

```
python def _isApprovedOrOwner(_spender: address, _tokenId: uint256) -> bool: """ @dev Returns whether the given spender can transfer a given token ID @param spender address of the spender to query @param tokenId uint256 ID of the token to be transferred @return bool whether the msg.sender is approved for the given token ID, is an operator of the owner, or is the owner of the token """ owner: address = self.idToOwner[_tokenId] spenderIsOwner: bool = owner == _spender spenderIsApproved: bool = _spender == self.idToApprovals[_tokenId] spenderIsApprovedForAll: bool = (self.ownerToOperators[owner])[_spender] return (spenderIsOwner or spenderIsApproved) or spenderIsApprovedForAll
```

There are three ways in which an address can be allowed to transfer a token:

1. The address is the owner of the token
2. The address is approved to spend that token
3. The address is an operator for the owner of the token

The function above can be a view because it doesn't change the state. To reduce operating costs, any function that *can* be a view *should* be a view.

``python @internal def _addTokenTo(_to: address, _tokenId: uint256): """ @dev Add a NFT to a given address Throws if `_tokenId` is owned by someone. """ # Throws if `_tokenId` is owned by someone assert self.idToOwner[_tokenId] == ZERO_ADDRESS # Change the owner self.idToOwner[_tokenId] = _to # Change count tracking self.ownerToNFTokenCount[_to] += 1

@internal def _removeTokenFrom(_from: address, _tokenId: uint256): """ @dev Remove a NFT from a given address Throws if `_from` is not the current owner. """ # Throws if `_from` is not the current owner assert self.idToOwner[_tokenId] == _from # Change the owner self.idToOwner[_tokenId] = ZERO_ADDRESS # Change count tracking self.ownerToNFTokenCount[_from] -= 1 ```

When there's a problem with a transfer we revert the call.

```
python @internal def _clearApproval(_owner: address, _tokenId: uint256): """ @dev Clear an approval of a given address Throws if `_owner` is not the current owner. """ # Throws if `_owner` is not the current owner assert self.idToOwner[_tokenId] == _owner if self.idToApprovals[_tokenId] != ZERO_ADDRESS: # Reset approvals self.idToApprovals[_tokenId] = ZERO_ADDRESS
```

Only change the value if necessary. State variables live in storage. Writing to storage is one of the most expensive operations the EVM (Ethereum Virtual Machine) does (in terms of gas). Therefore, it is a good idea to minimize it, even writing the existing value has a high cost.

```
python @internal def _transferFrom(_from: address, _to: address, _tokenId: uint256, _sender: address): """ @dev Execute transfer of a NFT. Throws unless `msg.sender` is the current owner, an authorized operator, or the approved address for this NFT. (NOTE: `msg.sender` not allowed in private function so pass `_sender`.) Throws if `_to` is the zero address. Throws if `_from` is not the current owner. Throws if `_tokenId` is not a valid NFT. """
```

We have this internal function because there are two ways to transfer tokens (regular and safe), but we want only a single location in the code where we do it to make auditing easier.

```
python # Check requirements assert self._isApprovedOrOwner(_sender, _tokenId) # Throws if `_to` is the zero address assert _to != ZERO_ADDRESS # Clear approval. Throws if `_from` is not the current owner self._clearApproval(_from, _tokenId) # Remove NFT. Throws if `_tokenId` is not a valid NFT self._removeTokenFrom(_from, _tokenId) # Add NFT self._addTokenTo(_to, _tokenId) # Log the transfer log Transfer(_from, _to, _tokenId)
```

To emit an event in Vyper you use a `log` statement (see here for more details).

Transfer Functions {#transfer-funs}

```python

### TRANSFER FUNCTIONS

@external def transferFrom(_from: address, _to: address, _tokenId: uint256): """ @dev Throws unless `msg.sender` is the current owner, an authorized operator, or the approved address for this NFT. Throws if `_from` is not the current owner. Throws if `_to` is the zero address. Throws if `_tokenId` is not a valid NFT. @notice The caller is responsible to confirm that `_to` is capable of receiving NFTs or else they maybe be permanently lost. @param _from The current owner of the NFT. @param _to The new owner. @param _tokenId The NFT to transfer. """ self._transferFrom(_from, _to, _tokenId, msg.sender) ```

This function lets you transfer to an arbitrary address. Unless the address is a user, or a contract that knows how to transfer tokens, any token you transfer will be stuck in that address and useless.

```
python @external def safeTransferFrom(_from: address, _to: address, _tokenId: uint256, _data: Bytes[1024]=b""): """ @dev Transfers the ownership of an NFT from one address to another address. Throws unless `msg.sender` is the current owner, an authorized operator, or the approved address for this NFT. Throws if `_from` is not the current owner. Throws if `_to` is the zero address. Throws if `_tokenId` is not a valid NFT. If `_to` is a smart contract, it calls `onERC721Received` on `_to` and throws if the return value is not `bytes4(keccak256("onERC721Received(address,address,uint256,bytes)"))`. NOTE: bytes4 is represented by bytes32 with padding @param _from The current owner of the NFT. @param _to The new owner. @param _tokenId The NFT to transfer. @param _data Additional data with no specified format, sent in call to `_to`. """ self._transferFrom(_from, _to, _tokenId, msg.sender)
```

It is OK to do the transfer first because if there's a problem we are going to revert anyway, so everything done in the call will be cancelled.

```
python if _to.is_contract: # check if `_to` is a contract address
```

First check to see if the address is a contract (if it has code). If not, assume it is a user address and the user will be able to use the token or transfer it. But don't let it lull you into a false sense of security. You can lose tokens, even with `safeTransferFrom`, if you transfer them to an address for which nobody knows the private key.

```
python returnValue: bytes32 = ERC721Receiver(_to).onERC721Received(msg.sender, _from, _tokenId, _data)
```

Call the target contract to see if it can receive ERC-721 tokens.

```
python # Throws if transfer destination is a contract which does not implement 'onERC721Received' assert returnValue == method_id("onERC721Received(address,address,uint256,bytes)", output_type=bytes32)
```

If the destination is a contract, but one that doesn't accept ERC-721 tokens (or that decided not to accept this particular transfer), revert.

```
python @external def approve(_approved: address, _tokenId: uint256): """ @dev Set or reaffirm the approved address for an NFT. The zero address indicates there is no approved address. Throws unless `msg.sender` is the current NFT owner, or an authorized operator of the current owner. Throws if `_tokenId` is not a valid NFT. (NOTE: This is not written the EIP) Throws if `_approved` is the current owner. (NOTE: This is not written the EIP) @param _approved Address to be approved for the given NFT ID. @param _tokenId ID of the token to be approved. """ owner: address = self.idToOwner[_tokenId] # Throws if `_tokenId` is not a valid NFT assert owner != ZERO_ADDRESS # Throws if `_approved` is the current owner assert _approved != owner
```

By convention if you want not to have an approver you appoint the zero address, not yourself.

```
python # Check requirements senderIsOwner: bool = self.idToOwner[_tokenId] == msg.sender senderIsApprovedForAll: bool = (self.ownerToOperators[owner])[msg.sender] assert (senderIsOwner or senderIsApprovedForAll)
```

To set an approval you can either be the owner, or an operator authorized by the owner.

```python # Set the approval self.idToApprovals[_tokenId] = _approved log Approval(owner, _approved, _tokenId)

@external def setApprovalForAll(_operator: address, _approved: bool): """ @dev Enables or disables approval for a third party ("operator") to manage all of `msg.sender`'s assets. It also emits the ApprovalForAll event. Throws if `_operator` is the `msg.sender`. (NOTE: This is not written the EIP) @notice This works even if sender doesn't own any tokens at the time. @param _operator Address to add to the set of authorized operators. @param _approved True if the operators is approved, false to revoke approval. """ # Throws if `_operator` is the `msg.sender` assert _operator != msg.sender self.ownerToOperators[msg.sender][_operator] = _approved log ApprovalForAll(msg.sender, _operator, _approved) ```
```

Mint New Tokens and Destroy Existing Ones {#mint-burn}

The account that created the contract is the `minter`, the super user that is authorized to mint new NFTs. However, even it is not allowed to burn existing tokens. Only the owner, or an entity authorized by the owner, can do that.

```python

**MINT & BURN FUNCTIONS**

@external def mint(_to: address, _tokenId: uint256) -> bool: ```

This function always returns `True`, because if the operation fails it is reverted.

```
python """ @dev Function to mint tokens Throws if `msg.sender` is not the minter. Throws if `_to` is zero address. Throws if `_tokenId` is owned by someone. @param _to The address that will receive the minted tokens. @param _tokenId The token id to mint. @return A boolean that indicates if the operation was successful. """ # Throws if `msg.sender` is not the minter assert msg.sender == self.minter
```

Only the minter (the account that created the ERC-721 contract) can mint new tokens. This can be a problem in the future if we want to change the minter's identity. In a production contract you would probably want a function that allows the minter to transfer minter privileges to somebody else.

```
python # Throws if `_to` is zero address assert _to != ZERO_ADDRESS # Add NFT. Throws if `_tokenId` is owned by someone self._addTokenTo(_to, _tokenId) log Transfer(ZERO_ADDRESS, _to, _tokenId) return True
```

By convention, the minting of new tokens counts as a transfer from address zero.

```python

@external def burn(_tokenId: uint256): """ @dev Burns a specific ERC721 token. Throws unless `msg.sender` is the current owner, an authorized operator, or the approved address for this NFT. Throws if `_tokenId` is not a valid NFT. @param _tokenId uint256 id of the ERC721 token to be burned. """ # Check requirements assert self._isApprovedOrOwner(msg.sender, _tokenId) owner: address = self.idToOwner[_tokenId] # Throws if `_tokenId` is not a valid NFT assert owner != ZERO_ADDRESS self._clearApproval(owner, _tokenId) self._removeTokenFrom(owner, _tokenId) log Transfer(owner, ZERO_ADDRESS, _tokenId) ```
```

Anybody who is allowed to transfer a token is allowed to burn it. While a burn appears equivalent to transfer to the zero address, the zero address does not actually receives the token. This allows us to free up all the storage that was used for the token, which can reduce the gas cost of the transaction.

Using this Contract {#using-contract}

In contrast to Solidity, Vyper does not have inheritance. This is a deliberate design choice to make the code clearer and therefore easier to secure. So to create your own Vyper ERC-721 contract you take this contract and modify it to implement the business logic you want.

Conclusion {#conclusion}

For review, here are some of the most important ideas in this contract:

- To receive ERC-721 tokens with a safe transfer, contracts have to implement the `ERC721Receiver` interface.
- Even if you use safe transfer, tokens can still get stuck if you send them to an address whose private key is unknown.
- When there is a problem with an operation it is a good idea to `revert` the call, rather than just return a failure value.
- ERC-721 tokens exist when they have an owner.
- There are three ways to be authorized to transfer an NFT. You can be the owner, be approved for a specific token, or be an operator for all of the owner's tokens.
- Past events are visible only outside the blockchain. Code running inside the blockchain cannot view them.

Now go and implement secure Vyper contracts.
```