
PyTeal

Oct 27, 2020

1	Overview	3
2	Install PyTeal	5
3	PyTeal Examples	7
4	Data Types and Constants	29
5	Arithmetic Operators	31
6	Byte Operators	33
7	Transaction Fields and Global Parameters	35
8	Cryptographic Primitives	39
9	Control Flow	41
10	State Access and Manipulation	45
11	PyTeal Package	51
12	Indices and tables	75
	Python Module Index	77
	Index	79

PyTeal is a Python language binding for [Algorand Smart Contracts \(ASC1s\)](#).

Algorand Smart Contracts are implemented using a new language that is stack-based, called [Transaction Execution Approval Language \(TEAL\)](#). This is a non-Turing complete language that allows branch forwards but prevents recursive logic to maximize safety and performance.

However, TEAL is essentially an assembly language. With PyTeal, developers can express smart contract logic purely using Python. PyTeal provides high level, functional programming style abstractions over TEAL and does type checking at construction time.

The [User Guide](#) describes many useful features in PyTeal, and the complete documentation for every expression and operation can be found in the [PyTeal Package API documentation](#).

PyTeal **hasn't been security audited**. Use it at your own risk.

CHAPTER 1

Overview

With PyTeal, developers can easily write **Algorand Smart Contracts (ASC1s)** in Python. PyTeal supports both stateless and statefull smart contracts.

Below is an example of writing a basic stateless smart contract that allows a specific receiver to withdraw funds from an account.

```
# This example is provided for informational purposes only and has not been audited,
↳ for security.

from pyteal import *

"""Basic Bank"""

def bank_for_account(receiver):
    """Only allow receiver to withdraw funds from this contract account.

    Args:
        receiver (str): Base 32 Algorand address of the receiver.
    """

    is_payment = Txn.type_enum() == TxnType.Payment
    is_single_tx = Global.group_size() == Int(1)
    is_correct_receiver = Txn.receiver() == Addr(receiver)
    no_close_out_addr = Txn.close_remainder_to() == Global.zero_address()
    no_rekey_addr = Txn.rekey_to() == Global.zero_address()
    acceptable_fee = Txn.fee() <= Int(1000)

    return And(
        is_payment,
        is_single_tx,
        is_correct_receiver,
        no_close_out_addr,
        no_rekey_addr,
        acceptable_fee
```

(continues on next page)

(continued from previous page)

```
)

if __name__ == "__main__":
    program = bank_for_account(
        → "ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBAYFZM")
    print(compileTeal(program, Mode.Signature))
```

As shown in this example, the logic of smart contract is expressed using PyTeal expressions constructed in Python. PyTeal overloads Python's arithmetic operators such as `<` and `==` (more overloaded operators can be found in *Arithmetic Operators*), allowing Python developers express smart contract logic more naturally.

Lastly, `compileTeal` is called to convert an PyTeal expression to a TEAL program, consisting of a sequence of TEAL opcodes. The output of the above example is:

```
#pragma version 2
txn TypeEnum
int pay
==
global GroupSize
int 1
==
&&
txn Receiver
addr ZZAF5ARA4MEC5PVDOP64JM5O5MQST63Q2KOY2FLYFLXXD3PFSNJJBAYFZM
==
&&
txn CloseRemainderTo
global ZeroAddress
==
&&
txn RekeyTo
global ZeroAddress
==
&&
txn Fee
int 1000
<=
&&
```

CHAPTER 2

Install PyTeal

The easiest way of installing PyTeal is using `pip` :

```
$ pip3 install pyteal
```

Alternatively, choose a [distribution file](#), and run

```
$ pip3 install [file name]
```


Here are some additional PyTeal example programs:

3.1 Signature Mode

3.1.1 Atomic Swap

Atomic Swap allows the transfer of Algos from a buyer to a seller in exchange for a good or service. This is done using a *Hashed Time Locked Contract*. In this scheme, the buyer funds a TEAL account with the sale price. The buyer also picks a secret value and encodes a secure hash of this value in the TEAL program. The TEAL program will transfer its balance to the seller if the seller is able to provide the secret value that corresponds to the hash in the program. When the seller renders the good or service to the buyer, the buyer discloses the secret from the program. The seller can immediately verify the secret and withdraw the payment.

```
# This example is provided for informational purposes only and has not been audited_
↪ for security.

from pyteal import *

"""Atomic Swap"""

alice = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
bob = Addr("7Z5PW02C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
secret = Bytes("base32", "23232323232323")
timeout = 3000

def htlc(tmpl_seller=alice,
         tmpl_buyer=bob,
         tmpl_fee=1000,
         tmpl_secret=secret,
         tmpl_hash_fn=Sha256,
         tmpl_timeout=timeout):
```

(continues on next page)

(continued from previous page)

```

    fee_cond = Txn.fee() < Int(templ_fee)
    safety_cond = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.close_remainder_to() == Global.zero_address(),
        Txn.rekey_to() == Global.zero_address(),
    )

    recv_cond = And(
        Txn.receiver() == templ_seller,
        templ_hash_fn(Arg(0)) == templ_secret
    )

    esc_cond = And(
        Txn.receiver() == templ_buyer,
        Txn.first_valid() > Int(templ_timeout)
    )

    return And(
        fee_cond,
        safety_cond,
        Or(recv_cond, esc_cond)
    )

if __name__ == "__main__":
    print(compileTeal(htlc(), Mode.Signature))

```

3.1.2 Split Payment

Split Payment splits payment between `templ_rcv1` and `templ_rcv2` on the ratio of `templ_ratn` / `templ_ratd`.

```

# This example is provided for informational purposes only and has not been audited,
↳ for security.

from pyteal import *

"""Split Payment"""

templ_fee = Int(1000)
templ_rcv1 = Addr("6ZHGH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUDHO57VTCMJOBGY")
templ_rcv2 = Addr("7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
templ_own = Addr("5MK5NGBRT5RL6IGUSYDIX5P7TNNZKRVXKT6FGVI6UVK6IZAWTYQGE4RZIQ")
templ_ratn = Int(1)
templ_ratd = Int(3)
templ_min_pay = Int(1000)
templ_timeout = Int(3000)

def split(templ_fee=templ_fee,
          templ_rcv1=templ_rcv1,
          templ_rcv2=templ_rcv2,
          templ_own=templ_own,
          templ_ratn=templ_ratn,
          templ_ratd=templ_ratd,
          templ_min_pay=templ_min_pay,
          templ_timeout=templ_timeout):

```

(continues on next page)

(continued from previous page)

```

split_core = And(
    Txn.type_enum() == TxnType.Payment,
    Txn.fee() < tmpl_fee,
    Txn.rekey_to() == Global.zero_address()
)

split_transfer = And(
    Gtxn[0].sender() == Gtxn[1].sender(),
    Txn.close_remainder_to() == Global.zero_address(),
    Gtxn[0].receiver() == tmpl_rcv1,
    Gtxn[1].receiver() == tmpl_rcv2,
    Gtxn[0].amount() == ((Gtxn[0].amount() + Gtxn[1].amount()) * tmpl_ratn) / 
    ↪tmpl_ratd,
    Gtxn[0].amount() == tmpl_min_pay
)

split_close = And(
    Txn.close_remainder_to() == tmpl_own,
    Txn.receiver() == Global.zero_address(),
    Txn.amount() == Int(0),
    Txn.first_valid() > tmpl_timeout
)

split_program = And(
    split_core,
    If(Global.group_size() == Int(2),
        split_transfer,
        split_close
    )
)

return split_program

if __name__ == "__main__":
    print(compileTeal(split(), Mode.Signature))

```

3.1.3 Periodic Payment

Periodic Payment allows some account to execute periodic withdrawal of funds. This PyTeal program creates an contract account that allows `tmpl_rcv` to withdraw `tmpl_amt` every `tmpl_period` rounds for `tmpl_dur` after every multiple of `tmpl_period`.

After `tmpl_timeout`, all remaining funds in the escrow are available to `tmpl_rcv`.

```

# This example is provided for informational purposes only and has not been audited,
↪for security.

from pyteal import *

"""Periodic Payment"""

tmpl_fee = Int(1000)
tmpl_period = Int(50)
tmpl_dur = Int(5000)

```

(continues on next page)

(continued from previous page)

```

tmpl_lease = Bytes("base64", "023sdDE2")
tmpl_amt = Int(2000)
tmpl_rcv = Addr("6ZHGHH5Z5CTPCF5WCESXMGRSVK7QJETR63M3NY5FJCUYDHO57VTCMJOBGY")
tmpl_timeout = Int(30000)

def periodic_payment(tmpl_fee=tmpl_fee,
                    tmpl_period=tmpl_period,
                    tmpl_dur=tmpl_dur,
                    tmpl_lease=tmpl_lease,
                    tmpl_amt=tmpl_amt,
                    tmpl_rcv=tmpl_rcv,
                    tmpl_timeout=tmpl_timeout):

    periodic_pay_core = And(
        Txn.type_enum() == TxnType.Payment,
        Txn.fee() < tmpl_fee,
        Txn.first_valid() % tmpl_period == Int(0),
        Txn.last_valid() == tmpl_dur + Txn.first_valid(),
        Txn.lease() == tmpl_lease
    )

    periodic_pay_transfer = And(
        Txn.close_remainder_to() == Global.zero_address(),
        Txn.rekey_to() == Global.zero_address(),
        Txn.receiver() == tmpl_rcv,
        Txn.amount() == tmpl_amt
    )

    periodic_pay_close = And(
        Txn.close_remainder_to() == tmpl_rcv,
        Txn.rekey_to() == Global.zero_address(),
        Txn.receiver() == Global.zero_address(),
        Txn.first_valid() == tmpl_timeout,
        Txn.amount() == Int(0)
    )

    periodic_pay_escrow = periodic_pay_core.And(periodic_pay_transfer.Or(periodic_pay_
→close))

    return periodic_pay_escrow

if __name__ == "__main__":
    print(compileTeal(periodic_payment(), Mode.Signature))

```

3.2 Application Mode

3.2.1 Voting

Voting allows accounts to register and vote for arbitrary choices. Here a *choice* is any byte slice and anyone is allowed to register to vote.

This example has a configurable *registration period* defined by the global state `RegBegin` and `RegEnd` which restrict when accounts can register to vote. There is also a separate configurable *voting period* defined by the global state `VotingBegin` and `VotingEnd` which restrict when voting can take place.

An account must register in order to vote. Accounts cannot vote more than once, and if an account opts out of the application before the voting period has concluded, their vote is discarded. The results are visible in the global state of the application, and the winner is the candidate with the highest number of votes.

```
# This example is provided for informational purposes only and has not been audited,
↪ for security.

from pyteal import *

def approval_program():
    on_creation = Seq([
        App.globalPut(Bytes("Creator"), Txn.sender()),
        Assert(Txn.application_args.length() == Int(4)),
        App.globalPut(Bytes("RegBegin"), Btoi(Txn.application_args[0])),
        App.globalPut(Bytes("RegEnd"), Btoi(Txn.application_args[1])),
        App.globalPut(Bytes("VoteBegin"), Btoi(Txn.application_args[2])),
        App.globalPut(Bytes("VoteEnd"), Btoi(Txn.application_args[3])),
        Return(Int(1))
    ])

    is_creator = Txn.sender() == App.globalGet(Bytes("Creator"))

    get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))

    on_closeout = Seq([
        get_vote_of_sender,
        If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_vote_of_sender.
↪ hasValue()),
            App.globalPut(get_vote_of_sender.value(), App.globalGet(get_vote_of_
↪ sender.value()) - Int(1))
        ),
        Return(Int(1))
    ])

    on_register = Return(And(
        Global.round() >= App.globalGet(Bytes("RegBegin")),
        Global.round() <= App.globalGet(Bytes("RegEnd"))
    ))

    choice = Txn.application_args[1]
    choice_tally = App.globalGet(choice)
    on_vote = Seq([
        Assert(And(
            Global.round() >= App.globalGet(Bytes("VoteBegin")),
            Global.round() <= App.globalGet(Bytes("VoteEnd"))
        )),
        get_vote_of_sender,
        If(get_vote_of_sender.hasValue(),
            Return(Int(0))
        ),
        App.globalPut(choice, choice_tally + Int(1)),
        App.localPut(Int(0), Bytes("voted"), choice),
        Return(Int(1))
    ])

    program = Cond(
        [Txn.application_id() == Int(0), on_creation],
        [Txn.on_completion() == OnComplete.DeleteApplication, Return(is_creator)],
```

(continues on next page)

(continued from previous page)

```

        [Txn.on_completion() == OnComplete.UpdateApplication, Return(is_creator)],
        [Txn.on_completion() == OnComplete.CloseOut, on_closeout],
        [Txn.on_completion() == OnComplete.OptIn, on_register],
        [Txn.application_args[0] == Bytes("vote"), on_vote]
    )

    return program

def clear_state_program():
    get_vote_of_sender = App.localGetEx(Int(0), App.id(), Bytes("voted"))
    program = Seq([
        get_vote_of_sender,
        If(And(Global.round() <= App.globalGet(Bytes("VoteEnd")), get_vote_of_sender.
↪hasValue()),
            App.globalPut(get_vote_of_sender.value(), App.globalGet(get_vote_of_
↪sender.value()) - Int(1))
        ),
        Return(Int(1))
    ])

    return program

if __name__ == "__main__":
    with open('vote_approval.teal', 'w') as f:
        compiled = compileTeal(approval_program(), Mode.Application)
        f.write(compiled)

    with open('vote_clear_state.teal', 'w') as f:
        compiled = compileTeal(clear_state_program(), Mode.Application)
        f.write(compiled)

```

A reference script that deploys the voting application is below:

```

# based off https://github.com/algorand/docs/blob/
↪cdf11d48a4b1168752e6ccaf77c8b9e8e599713a/examples/smart_contracts/v2/python/
↪stateful_smart_contracts.py

import base64
import datetime

from algosdk.future import transaction
from algosdk import account, mnemonic
from algosdk.v2client import algod
from pyteal import compileTeal, Mode
from vote import approval_program, clear_state_program

# user declared account mnemonics
creator_mnemonic = "Your 25-word mnemonic goes here"
user_mnemonic = "A second distinct 25-word mnemonic goes here"

# user declared algod connection parameters. Node must have EnableDeveloperAPI set to
↪true in its config
algod_address = "http://localhost:4001"
algod_token = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"

# helper function to compile program source
def compile_program(client, source_code):

```

(continues on next page)

(continued from previous page)

```

compile_response = client.compile(source_code)
return base64.b64decode(compile_response['result'])

# helper function that converts a mnemonic passphrase into a private signing key
def get_private_key_from_mnemonic(mn):
    private_key = mnemonic.to_private_key(mn)
    return private_key

# helper function that waits for a given txid to be confirmed by the network
def wait_for_confirmation(client, txid):
    last_round = client.status().get('last-round')
    txinfo = client.pending_transaction_info(txid)
    while not (txinfo.get('confirmed-round') and txinfo.get('confirmed-round') > 0):
        print("Waiting for confirmation...")
        last_round += 1
        client.status_after_block(last_round)
        txinfo = client.pending_transaction_info(txid)
    print("Transaction {} confirmed in round {}".format(txid, txinfo.get('confirmed-
↳round')))
    return txinfo

def wait_for_round(client, round):
    last_round = client.status().get('last-round')
    print(f"Waiting for round {round}")
    while last_round < round:
        last_round += 1
        client.status_after_block(last_round)
        print(f"Round {last_round}")

# create new application
def create_app(client, private_key, approval_program, clear_program, global_schema,
↳local_schema, app_args):
    # define sender as creator
    sender = account.address_from_private_key(private_key)

    # declare on_complete as NoOp
    on_complete = transaction.OnComplete.NoOpOC.real

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationCreateTxn(sender, params, on_complete, \
                                           approval_program, clear_program, \
                                           global_schema, local_schema, app_args)

    # sign transaction
    signed_txn = txn.sign(private_key)
    tx_id = signed_txn.transaction.get_txid()

    # send transaction
    client.send_transactions([signed_txn])

    # await confirmation

```

(continues on next page)

(continued from previous page)

```

wait_for_confirmation(client, tx_id)

# display results
transaction_response = client.pending_transaction_info(tx_id)
app_id = transaction_response['application-index']
print("Created new app-id:", app_id)

return app_id

# opt-in to application
def opt_in_app(client, private_key, index):
    # declare sender
    sender = account.address_from_private_key(private_key)
    print("OptIn from account: ", sender)

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationOptInTxn(sender, params, index)

    # sign transaction
    signed_txn = txn.sign(private_key)
    tx_id = signed_txn.transaction.get_txid()

    # send transaction
    client.send_transactions([signed_txn])

    # await confirmation
    wait_for_confirmation(client, tx_id)

    # display results
    transaction_response = client.pending_transaction_info(tx_id)
    print("OptIn to app-id:", transaction_response['txn']['txn']['apid'])

# call application
def call_app(client, private_key, index, app_args):
    # declare sender
    sender = account.address_from_private_key(private_key)
    print("Call from account:", sender)

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationNoOpTxn(sender, params, index, app_args)

    # sign transaction
    signed_txn = txn.sign(private_key)
    tx_id = signed_txn.transaction.get_txid()

```

(continues on next page)

(continued from previous page)

```

# send transaction
client.send_transactions([signed_txn])

# await confirmation
wait_for_confirmation(client, tx_id)

def format_state(state):
    formatted = {}
    for item in state:
        key = item['key']
        value = item['value']
        formatted_key = base64.b64decode(key).decode('utf-8')
        if value['type'] == 1:
            # byte string
            if formatted_key == 'voted':
                formatted_value = base64.b64decode(value['bytes']).decode('utf-8')
            else:
                formatted_value = value['bytes']
            formatted[formatted_key] = formatted_value
        else:
            # integer
            formatted[formatted_key] = value['uint']
    return formatted

# read user local state
def read_local_state(client, addr, app_id):
    results = client.account_info(addr)
    for local_state in results['apps-local-state']:
        if local_state['id'] == app_id:
            if 'key-value' not in local_state:
                return {}
            return format_state(local_state['key-value'])
    return {}

# read app global state
def read_global_state(client, addr, app_id):
    results = client.account_info(addr)
    apps_created = results['created-apps']
    for app in apps_created:
        if app['id'] == app_id:
            return format_state(app['params']['global-state'])
    return {}

# delete application
def delete_app(client, private_key, index):
    # declare sender
    sender = account.address_from_private_key(private_key)

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationDeleteTxn(sender, params, index)

```

(continues on next page)

(continued from previous page)

```

    # sign transaction
    signed_txn = txn.sign(private_key)
    tx_id = signed_txn.transaction.get_txid()

    # send transaction
    client.send_transactions([signed_txn])

    # await confirmation
    wait_for_confirmation(client, tx_id)

    # display results
    transaction_response = client.pending_transaction_info(tx_id)
    print("Deleted app-id:", transaction_response['txn']['txn']['apid'])

# close out from application
def close_out_app(client, private_key, index):
    # declare sender
    sender = account.address_from_private_key(private_key)

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationCloseOutTxn(sender, params, index)

    # sign transaction
    signed_txn = txn.sign(private_key)
    tx_id = signed_txn.transaction.get_txid()

    # send transaction
    client.send_transactions([signed_txn])

    # await confirmation
    wait_for_confirmation(client, tx_id)

    # display results
    transaction_response = client.pending_transaction_info(tx_id)
    print("Closed out from app-id: ", transaction_response['txn']['txn']['apid'])

# clear application
def clear_app(client, private_key, index):
    # declare sender
    sender = account.address_from_private_key(private_key)

    # get node suggested parameters
    params = client.suggested_params()
    # comment out the next two (2) lines to use suggested fees
    params.flat_fee = True
    params.fee = 1000

    # create unsigned transaction
    txn = transaction.ApplicationClearStateTxn(sender, params, index)

    # sign transaction

```

(continues on next page)

(continued from previous page)

```

signed_txn = txn.sign(private_key)
tx_id = signed_txn.transaction.get_txid()

# send transaction
client.send_transactions([signed_txn])

# await confirmation
wait_for_confirmation(client, tx_id)

# display results
transaction_response = client.pending_transaction_info(tx_id)
print("Cleared app-id:", transaction_response['txn']['txn']['apid'])

# convert 64 bit integer i to byte string
def intToBytes(i):
    lower8 = (1 << 8) - 1
    charList = [
        (i >> (8*7)) & lower8,
        (i >> (8*6)) & lower8,
        (i >> (8*5)) & lower8,
        (i >> (8*4)) & lower8,
        (i >> (8*3)) & lower8,
        (i >> (8*2)) & lower8,
        (i >> (8*1)) & lower8,
        i & lower8
    ]
    string = ''.join(chr(c) for c in charList)
    return string.encode('latin1')

def main():
    # initialize an algodClient
    algod_client = algod.AlgodClient(algod_token, algod_address)

    # define private keys
    creator_private_key = get_private_key_from_mnemonic(creator_mnemonic)
    user_private_key = get_private_key_from_mnemonic(user_mnemonic)

    # declare application state storage (immutable)
    local_ints = 0
    local_bytes = 1
    global_ints = 24 # 4 for setup + 20 for choices. Use a larger number for more_
    ↪ choices.
    global_bytes = 1
    global_schema = transaction.StateSchema(global_ints, global_bytes)
    local_schema = transaction.StateSchema(local_ints, local_bytes)

    # get PyTeal approval program
    approval_program_ast = approval_program()
    # compile program to TEAL assembly
    approval_program_teal = compileTeal(approval_program_ast, Mode.Application)
    # compile program to binary
    approval_program_compiled = compile_program(algod_client, approval_program_teal)

    # get PyTeal clear state program
    clear_state_program_ast = clear_state_program()
    # compile program to TEAL assembly
    clear_state_program_teal = compileTeal(clear_state_program_ast, Mode.Application)

```

(continues on next page)

(continued from previous page)

```

    # compile program to binary
    clear_state_program_compiled = compile_program(algod_client, clear_state_program_
    ↪teal)

    # configure registration and voting period
    status = algod_client.status()
    regBegin = status['last-round'] + 10
    regEnd = regBegin + 10
    voteBegin = regEnd + 1
    voteEnd = voteBegin + 10

    print(f"Registration rounds: {regBegin} to {regEnd}")
    print(f"Vote rounds: {voteBegin} to {voteEnd}")

    # create list of bytes for app args
    app_args = [
        intToBytes(regBegin),
        intToBytes(regEnd),
        intToBytes(voteBegin),
        intToBytes(voteEnd)
    ]

    # create new application
    app_id = create_app(algod_client, creator_private_key, approval_program_compiled,
    ↪clear_state_program_compiled, global_schema, local_schema, app_args)

    # read global state of application
    print("Global state:", read_global_state(algod_client, account.address_from_
    ↪private_key(creator_private_key), app_id))

    # wait for registration period to start
    wait_for_round(algod_client, regBegin)

    # opt-in to application
    opt_in_app(algod_client, user_private_key, app_id)

    wait_for_round(algod_client, voteBegin)

    # call application without arguments
    call_app(algod_client, user_private_key, app_id, [b'vote', b'choiceA'])

    # read local state of application from user account
    print("Local state:", read_local_state(algod_client, account.address_from_private_
    ↪key(user_private_key), app_id))

    # wait for registration period to start
    wait_for_round(algod_client, voteEnd)

    # read global state of application
    global_state = read_global_state(algod_client, account.address_from_private_
    ↪key(creator_private_key), app_id)
    print("Global state:", global_state)

    max_votes = 0
    max_votes_choice = None
    for key, value in global_state.items():
        if key not in ('RegBegin', 'RegEnd', 'VoteBegin', 'VoteEnd', 'Creator') and
    ↪isinstance(value, int):

```

(continues on next page)

(continued from previous page)

```

        if value > max_votes:
            max_votes = value
            max_votes_choice = key

    print("The winner is:", max_votes_choice)

    # delete application
    delete_app(algod_client, creator_private_key, app_id)

    # clear application from user account
    clear_app(algod_client, user_private_key, app_id)

if __name__ == "__main__":
    main()

```

Example output for deployment would be:

```

Registration rounds: 592 to 602
Vote rounds: 603 to 613
Waiting for confirmation...
Transaction KXJHR6J4QSCAH036L77DPJ53CLZBCCSPSBAOGTGQDRA7WECDXUEA confirmed in round_
↳584.
Created new app-id: 29
Global state: {'RegEnd': 602, 'VoteBegin': 603, 'VoteEnd': 613, 'Creator':
↳'49y8gDrKSnm77cgRyFzYdlkw18SDVnKhhOis6NVVH8U=', 'RegBegin': 592}
Waiting for round 592
Round 585
Round 586
Round 587
Round 588
Round 589
Round 590
Round 591
Round 592
OptIn from account: FVQEFNOSD25TDBTTTIU2I5KW5DHR6PADYMZESTOCQ2O3ME4OWXEI7OHVRY
Waiting for confirmation...
Transaction YWXOAREFSUYID6QLWQHANTXK3NR2XOVTIQYKMD27F3VXJKP7CMYQ confirmed in round_
↳595.
OptIn to app-id: 29
Waiting for round 603
Round 596
Round 597
Round 598
Round 599
Round 600
Round 601
Round 602
Round 603
Call from account: FVQEFNOSD25TDBTTTIU2I5KW5DHR6PADYMZESTOCQ2O3ME4OWXEI7OHVRY
Waiting for confirmation...
Transaction WNV4DTPEMVGUXNRZHMWNSCUU7AQJOCFTBKJT6NV2KN6THT4QGKNQ confirmed in round_
↳606.
Local state: {'voted': 'choiceA'}
Waiting for round 613
Round 607
Round 608
Round 609

```

(continues on next page)

(continued from previous page)

```

Round 610
Round 611
Round 612
Round 613
Global state: {'RegBegin': 592, 'RegEnd': 602, 'VoteBegin': 603, 'VoteEnd': 613,
↳ 'choiceA': 1, 'Creator': '49y8gDrKSnm77cgRyFzYdlkw18SDVnKhhoiS6NVVH8U='}
The winner is: choiceA
Waiting for confirmation...
Transaction 535KBWJ7RQX4ISV763IUUICQWI6VERYBJ7J6X7HPMAMFNKJPSNPQ confirmed in round_
↳ 616.
Deleted app-id: 29
Waiting for confirmation...
Transaction Z56HDAJYARUC4PWGWQLCBA6TZYQOOLNOXY5XRM3IYUEEUCT5DRMA confirmed in round_
↳ 618.
Cleared app-id: 29

```

3.2.2 Asset

Asset is an implementation of a custom asset type using smart contracts. While Algorand has *ASAs*, in some blockchains the only way to create a custom asset is through smart contracts.

At creation, the creator specifies the total supply of the asset. Initially this supply is placed in a reserve and the creator is made an admin. Any admin can move funds from the reserve into the balance of any account that has opted into the application using the *mint* argument. Additionally, any admin can move funds from any account's balance into the reserve using the *burn* argument.

Accounts are free to transfer funds in their balance to any other account that has opted into the application. When an account opts out of the application, their balance is added to the reserve.

```

# This example is provided for informational purposes only and has not been audited_
↳ for security.

from pyteal import *

def approval_program():
    on_creation = Seq([
        Assert(Txn.application_args.length() == Int(1)),
        App.globalPut(Bytes("total supply"), Btoi(Txn.application_args[0])),
        App.globalPut(Bytes("reserve"), Btoi(Txn.application_args[0])),
        App.localPut(Int(0), Bytes("admin"), Int(1)),
        App.localPut(Int(0), Bytes("balance"), Int(0)),
        Return(Int(1))
    ])

    is_admin = App.localGet(Int(0), Bytes("admin"))

    on_closeout = Seq([
        App.globalPut(
            Bytes("reserve"),
            App.globalGet(Bytes("reserve")) + App.localGet(Int(0), Bytes("balance"))
        ),
        Return(Int(1))
    ])

    register = Seq([

```

(continues on next page)

(continued from previous page)

```

    App.localPut(Int(0), Bytes("balance"), Int(0)),
    Return(Int(1))
})

# configure the admin status of the account Txn.accounts[0]
# sender must be admin
new_admin_status = Btoi(Txn.application_args[1])
set_admin = Seq([
    Assert(And(is_admin, Txn.application_args.length() == Int(2))),
    App.localPut(Int(1), Bytes("admin"), new_admin_status),
    Return(Int(1))
])

# NOTE: The above set_admin code is carefully constructed. If instead we used the
↪following code:
# Seq([
#     Assert(Txn.application_args.length() == Int(2)),
#     App.localPut(Int(1), Bytes("admin"), new_admin_status),
#     Return(is_admin)
# ])
# It would be vulnerable to the following attack: a sender passes in their own
↪address as
# Txn.accounts[0], so then the line App.localPut(Int(1), Bytes("admin"), new_
↪admin_status)
# changes the sender's admin status, meaning the final Return(is_admin) can
↪return anything the
# sender wants. This allows anyone to become an admin!

# move assets from the reserve to Txn.accounts[0]
# sender must be admin
mint_amount = Btoi(Txn.application_args[1])
mint = Seq([
    Assert(Txn.application_args.length() == Int(2)),
    Assert(mint_amount <= App.globalGet(Bytes("reserve"))),
    App.globalPut(Bytes("reserve"), App.globalGet(Bytes("reserve")) - mint_
↪amount),
    App.localPut(Int(1), Bytes("balance"), App.localGet(Int(1), Bytes("balance"))
↪+ mint_amount),
    Return(is_admin)
])

# transfer assets from the sender to Txn.accounts[0]
transfer_amount = Btoi(Txn.application_args[1])
transfer = Seq([
    Assert(Txn.application_args.length() == Int(2)),
    Assert(transfer_amount <= App.localGet(Int(0), Bytes("balance"))),
    App.localPut(Int(0), Bytes("balance"), App.localGet(Int(0), Bytes("balance"))
↪- transfer_amount),
    App.localPut(Int(1), Bytes("balance"), App.localGet(Int(1), Bytes("balance"))
↪+ transfer_amount),
    Return(Int(1))
])

program = Cond(
    [Txn.application_id() == Int(0), on_creation],
    [Txn.on_completion() == OnComplete.DeleteApplication, Return(is_admin)],
    [Txn.on_completion() == OnComplete.UpdateApplication, Return(is_admin)],
    [Txn.on_completion() == OnComplete.CloseOut, on_closeout],

```

(continues on next page)

(continued from previous page)

```

        [Txn.on_completion() == OnComplete.OptIn, register],
        [Txn.application_args[0] == Bytes("set admin"), set_admin],
        [Txn.application_args[0] == Bytes("mint"), mint],
        [Txn.application_args[0] == Bytes("transfer"), transfer]
    )

    return program

def clear_state_program():
    program = Seq([
        App.globalPut(
            Bytes("reserve"),
            App.globalGet(Bytes("reserve")) + App.localGet(Int(0), Bytes("balance"))
        ),
        Return(Int(1))
    ])

    return program

if __name__ == "__main__":
    with open('asset_approval.teal', 'w') as f:
        compiled = compileTeal(approval_program(), Mode.Application)
        f.write(compiled)

    with open('asset_clear_state.teal', 'w') as f:
        compiled = compileTeal(clear_state_program(), Mode.Application)
        f.write(compiled)

```

3.2.3 Security Token

Security Token is an extension of the *Asset* example with more features and restrictions. There are two types of admins, *contract admins* and *transfer admins*.

Contract admins can delete the smart contract if the entire supply is in the reserve. They can promote accounts to transfer or contract admins. They can also *mint* and *burn* funds.

Transfer admins can impose maximum balance limitations on accounts, temporarily lock accounts, assign accounts to transfer groups, and impose transaction restrictions between transaction groups.

Both contract and transfer admins can pause trading of funds and freeze individual accounts.

Accounts can only transfer funds if trading is not paused, both the sender and receive accounts are not frozen or temporarily locked, transfer group restrictions are not in place between them, and the receiver's account does not have a maximum balance restriction that would be invalidated.

```

# This example is provided for informational purposes only and has not been audited_
↪ for security.

from pyteal import *

def approval_program():
    on_creation = Seq([
        Assert(Txn.application_args.length() == Int(1)),
        App.globalPut(Bytes("total supply"), Btoi(Txn.application_args[0])),
        App.globalPut(Bytes("reserve"), Btoi(Txn.application_args[0])),
        App.globalPut(Bytes("paused"), Int(0)),

```

(continues on next page)

(continued from previous page)

```

    App.localPut(Int(0), Bytes("contract admin"), Int(1)),
    App.localPut(Int(0), Bytes("transfer admin"), Int(1)),
    App.localPut(Int(0), Bytes("balance"), Int(0)),
    Return(Int(1))
])

is_contract_admin = App.localGet(Int(0), Bytes("contract admin"))
is_transfer_admin = App.localGet(Int(0), Bytes("transfer admin"))
is_any_admin = is_contract_admin.Or(is_transfer_admin)

can_delete = And(
    is_contract_admin,
    App.globalGet(Bytes("total supply")) == App.globalGet(Bytes("reserve"))
)

on_closeout = Seq([
    App.globalPut(
        Bytes("reserve"),
        App.globalGet(Bytes("reserve")) + App.localGet(Int(0), Bytes("balance"))
    ),
    Return(Int(1))
])

register = Seq([
    App.localPut(Int(0), Bytes("balance"), Int(0)),
    Return(Int(1))
])

# pause all transfers
# sender must be any admin
new_pause_value = Btoi(Txn.application_args[1])
pause = Seq([
    Assert(Txn.application_args.length() == Int(2)),
    App.globalPut(Bytes("paused"), new_pause_value),
    Return(is_any_admin)
])

# configure the admin status of the account Txn.accounts[0]
# sender must be contract admin
new_admin_type = Txn.application_args[1]
new_admin_status = Btoi(Txn.application_args[2])
set_admin = Seq([
    Assert(And(
        is_contract_admin,
        Txn.application_args.length() == Int(3),
        Or(new_admin_type == Bytes("contract admin"), new_admin_type == Bytes(
            ↪ "transfer admin")),
        Txn.accounts.length() == Int(1)
    )),
    App.localPut(Int(1), new_admin_type, new_admin_status),
    Return(Int(1))
])

# NOTE: The above set_admin code is carefully constructed. If instead we used the
↪ following code:
# Seq([
#     Assert(And(
#         Txn.application_args.length() == Int(3),

```

(continues on next page)

(continued from previous page)

```

#         Or(new_admin_type == Bytes("contract admin"), new_admin_type == Bytes(
↪ "transfer admin")),
#         Txn.accounts.length() == Int(1)
#     )),
#     App.localPut(Int(1), new_admin_type, new_admin_status),
#     Return(is_contract_admin)
# ])
# It would be vulnerable to the following attack: a sender passes in their own
↪ address as
# Txn.accounts[0], so then the line App.localPut(Int(1), new_admin_type, new_
↪ admin_status)
# changes the sender's admin status, meaning the final Return(is_contract_admin)
↪ can return
# anything the sender wants. This allows anyone to become an admin!

# freeze Txn.accounts[0]
# sender must be any admin
new_freeze_value = Btoi(Txn.application_args[1])
freeze = Seq([
    Assert(And(
        Txn.application_args.length() == Int(2),
        Txn.accounts.length() == Int(1)
    )),
    App.localPut(Int(1), Bytes("frozen"), new_freeze_value),
    Return(is_any_admin)
])

# modify the max balance of Txn.accounts[0]
# if max_balance_value is 0, will delete the existing max balance limitation on
↪ the account
# sender must be transfer admin
max_balance_value = Btoi(Txn.application_args[1])
max_balance = Seq([
    Assert(And(
        Txn.application_args.length() == Int(2),
        Txn.accounts.length() == Int(1)
    )),
    If(max_balance_value == Int(0),
        App.localDel(Int(1), Bytes("max balance")),
        App.localPut(Int(1), Bytes("max balance"), max_balance_value)
    ),
    Return(is_transfer_admin)
])

# lock Txn.accounts[0] until a UNIX timestamp
# sender must be transfer admin
lock_until_value = Btoi(Txn.application_args[1])
lock_until = Seq([
    Assert(And(
        Txn.application_args.length() == Int(2),
        Txn.accounts.length() == Int(1)
    )),
    If(lock_until_value == Int(0),
        App.localDel(Int(1), Bytes("lock until")),
        App.localPut(Int(1), Bytes("lock until"), lock_until_value)
    ),
    Return(is_transfer_admin)
])

```

(continues on next page)

(continued from previous page)

```

    ])

    set_transfer_group = Seq([
        Assert(And(
            Txn.application_args.length() == Int(3),
            Txn.accounts.length() == Int(1)
        )),
        App.localPut(Int(1), Bytes("transfer group"), Btoi(Txn.application_args[2]))
    ])

    def getRuleKey(sendGroup, receiveGroup):
        return Concat(Bytes("rule"), Itob(sendGroup), Itob(receiveGroup))

    lock_transfer_key = getRuleKey(Btoi(Txn.application_args[2]), Btoi(Txn.
↪application_args[3]))
    lock_transfer_until = Btoi(Txn.application_args[4])
    lock_transfer_group = Seq([
        Assert(Txn.application_args.length() == Int(5)),
        If(lock_transfer_until == Int(0),
            App.globalDel(lock_transfer_key),
            App.globalPut(lock_transfer_key, lock_transfer_until)
        )
    ])

    # sender must be transfer admin
    transfer_group = Seq([
        Assert(Txn.application_args.length() > Int(2)),
        Cond(
            [Txn.application_args[1] == Bytes("set"), set_transfer_group],
            [Txn.application_args[1] == Bytes("lock"), lock_transfer_group]
        ),
        Return(is_transfer_admin)
    ])

    # move assets from the reserve to Txn.accounts[0]
    # sender must be contract admin
    mint_amount = Btoi(Txn.application_args[1])
    mint = Seq([
        Assert(And(
            Txn.application_args.length() == Int(2),
            Txn.accounts.length() == Int(1),
            mint_amount <= App.globalGet(Bytes("reserve"))
        )),
        App.globalPut(Bytes("reserve"), App.globalGet(Bytes("reserve")) - mint_
↪amount),
        App.localPut(Int(1), Bytes("balance"), App.localGet(Int(1), Bytes("balance"))_
↪+ mint_amount),
        Return(is_contract_admin)
    ])

    # move assets from Txn.accounts[0] to the reserve
    # sender must be contract admin
    burn_amount = Btoi(Txn.application_args[1])
    burn = Seq([
        Assert(And(
            Txn.application_args.length() == Int(2),
            Txn.accounts.length() == Int(1),

```

(continues on next page)

(continued from previous page)

```

        burn_amount <= App.localGet(Int(1), Bytes("balance"))
    )),
    App.globalPut(Bytes("reserve"), App.globalGet(Bytes("reserve")) + burn_
↪ amount),
    App.localPut(Int(1), Bytes("balance"), App.localGet(Int(1), Bytes("balance"))_
↪ burn_amount),
    Return(is_contract_admin)
])

# transfer assets from the sender to Txn.accounts[0]
transfer_amount = Btoi(Txn.application_args[1])
receiver_max_balance = App.localGetEx(Int(1), App.id(), Bytes("max balance"))
transfer = Seq([
    Assert(And(
        Txn.application_args.length() == Int(2),
        Txn.accounts.length() == Int(1),
        transfer_amount <= App.localGet(Int(0), Bytes("balance"))
    )),
    receiver_max_balance,
    If(
        Or(
            App.globalGet(Bytes("paused")),
            App.localGet(Int(0), Bytes("frozen")),
            App.localGet(Int(1), Bytes("frozen")),
            App.localGet(Int(0), Bytes("lock until")) >= Global.latest_
↪ timestamp(),
            App.localGet(Int(1), Bytes("lock until")) >= Global.latest_
↪ timestamp(),
            App.globalGet(getRuleKey(App.localGet(Int(0), Bytes("transfer group
↪ ")), App.localGet(Int(1), Bytes("transfer group")))) >= Global.latest_timestamp(),
            And(
                receiver_max_balance.hasValue(),
                receiver_max_balance.value() < App.localGet(Int(1), Bytes("balance
↪ ")) + transfer_amount
            )
        ),
        Return(Int(0))
    ),
    App.localPut(Int(0), Bytes("balance"), App.localGet(Int(0), Bytes("balance"))_
↪ transfer_amount),
    App.localPut(Int(1), Bytes("balance"), App.localGet(Int(1), Bytes("balance"))_
↪ + transfer_amount),
    Return(Int(1))
])

program = Cond(
    [Txn.application_id() == Int(0), on_creation],
    [Txn.on_completion() == OnComplete.DeleteApplication, Return(can_delete)],
    [Txn.on_completion() == OnComplete.UpdateApplication, Return(is_contract_
↪ admin)],
    [Txn.on_completion() == OnComplete.CloseOut, on_closeout],
    [Txn.on_completion() == OnComplete.OptIn, register],
    [Txn.application_args[0] == Bytes("pause"), pause],
    [Txn.application_args[0] == Bytes("set admin"), set_admin],
    [Txn.application_args[0] == Bytes("freeze"), freeze],
    [Txn.application_args[0] == Bytes("max balance"), max_balance],
    [Txn.application_args[0] == Bytes("lock until"), lock_until],

```

(continues on next page)

(continued from previous page)

```
[Txn.application_args[0] == Bytes("transfer group"), transfer_group],
[Txn.application_args[0] == Bytes("mint"), mint],
[Txn.application_args[0] == Bytes("burn"), burn],
[Txn.application_args[0] == Bytes("transfer"), transfer],
)

return program

def clear_state_program():
    program = Seq([
        App.globalPut(
            Bytes("reserve"),
            App.globalGet(Bytes("reserve")) + App.localGet(Int(0), Bytes("balance"))
        ),
        Return(Int(1))
    ])

    return program

if __name__ == "__main__":
    with open('security_token_approval.teal', 'w') as f:
        compiled = compileTeal(approval_program(), Mode.Application)
        f.write(compiled)

    with open('security_token_clear_state.teal', 'w') as f:
        compiled = compileTeal(clear_state_program(), Mode.Application)
        f.write(compiled)
```

Data Types and Constants

A PyTeal expression has one of the following two data types:

- `TealType.uint64`, 64 bit unsigned integer
- `TealType.bytes`, a slice of bytes

For example, all the transaction arguments (e.g. `Arg(0)`) are of type `TealType.bytes`. The first valid round of current transaction (`Txn.first_valid()`) is typed `TealType.uint64`.

4.1 Integers

`Int(n)` creates a `TealType.uint64` constant, where $n \geq 0$ and $n < 2^{64}$.

4.2 Bytes

A byte slice is a binary string. There are several ways to encode a byte slice in PyTeal:

4.2.1 UTF-8

Byte slices can be created from UTF-8 encoded strings. For example:

```
Bytes("hello world")
```

4.2.2 Base16

Byte slices can be created from a [RFC 4648#section-8](#) base16 encoded binary string, e.g. `"0xA21212EF"` or `"A21212EF"`. For example:

```
Bytes("base16", "0xA21212EF")
Bytes("base16", "A21212EF") # "0x" is optional
```

4.2.3 Base32

Byte slices can be created from a [RFC 4648#section-6](#) base32 encoded binary string **without padding**, e.g. "7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M".

```
Bytes("base32", "7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M")
```

4.2.4 Base64

Byte slices can be created from a [RFC 4648#section-4](#) base64 encoded binary string, e.g. "Zm9vYmE=".

```
Bytes("base64", "Zm9vYmE=")
```

4.3 Type Checking

All PyTeal expressions are type checked at construction time, for example, running the following code triggers a `TealTypeError`:

```
Int(0) < Arg(0)
```

Since `<` (overloaded Python operator, see [Arithmetic Operators](#) for more details) requires both operands of type `TealType.uint64`, while `Arg(0)` is of type `TealType.bytes`.

4.4 Conversion

Converting a value to its corresponding value in the other data type is supported by the following two operators:

- `Itob(n)`: generate a `TealType.bytes` value from a `TealType.uint64` value `n`
- `Btoi(b)`: generate a `TealType.uint64` value from a `TealType.bytes` value `b`

Arithmetic Operators

An arithmetic expression is an expression that results in a `TealType.uint64` value. In PyTeal, arithmetic expressions include integer arithmetics operators and boolean operators. We overloaded all integer arithmetics operator in Python.

Operator	Overloaded	Semantics	Example
<code>Lt(a, b)</code>	<code><</code>	<i>1 if a is less than b, 0 otherwise</i>	<code>Int(1) < Int(5)</code>
<code>Gt(a, b)</code>	<code>></code>	<i>1 if a is greater than b, 0 otherwise</i>	<code>Int(1) > Int(5)</code>
<code>Le(a, b)</code>	<code><=</code>	<i>1 if a is no greater than b, 0 otherwise</i>	<code>Int(1) <= Int(5)</code>
<code>Ge(a, b)</code>	<code>>=</code>	<i>1 if a is no less than b, 0 otherwise</i>	<code>Int(1) >= Int(5)</code>
<code>Add(a, b)</code>	<code>+</code>	<i>a + b, error (panic) if overflow</i>	<code>Int(1) + Int(5)</code>
<code>Minus(a, b)</code>	<code>-</code>	<i>a - b, error if underflow</i>	<code>Int(5) - Int(1)</code>
<code>Mul(a, b)</code>	<code>*</code>	<i>a * b, error if overflow</i>	<code>Int(2) * Int(3)</code>
<code>Div(a, b)</code>	<code>/</code>	<i>a / b, error if divided by zero</i>	<code>Int(3) / Int(2)</code>
<code>Mod(a, b)</code>	<code>%</code>	<i>a % b, modulo operation</i>	<code>Int(7) % Int(3)</code>
<code>Eq(a, b)</code>	<code>==</code>	<i>1 if a equals b, 0 otherwise</i>	<code>Int(7) == Int(7)</code>
<code>Neq(a, b)</code>	<code>!=</code>	<i>0 if a equals b, 1 otherwise</i>	<code>Int(7) != Int(7)</code>
<code>And(a, b)</code>		<i>1 if a > 0 && b > 0, 0 otherwise</i>	<code>And(Int(1), Int(1))</code>
<code>Or(a, b)</code>		<i>1 if a > 0 b > 0, 0 otherwise</i>	<code>Or(Int(1), Int(0))</code>
<code>Not(a)</code>		<i>1 if a equals 0, 0 otherwise</i>	<code>Not(Int(0))</code>
<code>BitwiseAnd(a, b)</code>	<code>&</code>	<i>a & b, bitwise and operation</i>	<code>Int(1) & Int(3)</code>
<code>BitwiseOr(a, b)</code>	<code> </code>	<i>a b, bitwise or operation</i>	<code>Int(2) Int(5)</code>
<code>BitwiseXor(a, b)</code>	<code>^</code>	<i>a ^ b, bitwise xor operation</i>	<code>Int(3) ^ Int(7)</code>
<code>BitwiseNot(a)</code>	<code>~</code>	<i>~a, bitwise complement operation</i>	<code>~Int(1)</code>

All these operators takes two `TealType.uint64` values. In addition, `Eq(a, b) (==)` and `Neq(a, b) (!=)` also work for byte slices. For example, `Arg(0) == Arg(1)` and `Arg(0) != Arg(1)` are valid PyTeal expressions.

Both `And` and `Or` also support more than 2 arguments when called as functions:

- `And(a, b, ...)`
- `Or(a, b, ...)`

The associativity and precedence of the overloaded Python arithmetic operators are the same as the [original python operators](#) . For example:

- `Int(1) + Int(2) + Int(3)` is equivalent to `Add(Add(Int(1), Int(2)), Int(3))`
- `Int(1) + Int(2) * Int(3)` is equivalent to `Add(Int(1), Mul(Int(2), Int(3)))`

TEAL byte slices are similar to strings and can be manipulated in much the same way.

6.1 Concat

Byte slices can be combined using the *Concat* expression. This expression takes at least two arguments and produces a new byte slice consisting of each argument, one after another. For example:

```
Concat(Bytes("a"), Bytes("b"), Bytes("c")) # equivalent to Bytes("abc")
```

6.2 Substring

Byte slices can be extracted from other byte slices using the *Substring* expression. This expression can extract part of a byte slicing given start and end indices. For example:

```
Substring(Bytes("algorand"), Int(0), Int(4)) # equivalent to Bytes("algo")
```

Transaction Fields and Global Parameters

PyTeal smart contracts can access properties of the current transaction and the state of the blockchain when they are running.

7.1 Transaction Fields

Information about the current transaction being evaluated can be obtained using the `Txn` object. Below are the PyTeal expressions that refer to transaction fields:

Operator	Type	Notes
<code>Txn.sender()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.fee()</code>	<code>TealType.uint64</code>	in microAlgos
<code>Txn.first_valid()</code>	<code>TealType.uint64</code>	round number
<code>Txn.last_valid()</code>	<code>TealType.uint64</code>	round number
<code>Txn.note()</code>	<code>TealType.bytes</code>	transaction note in bytes
<code>Txn.lease()</code>	<code>TealType.bytes</code>	transaction lease in bytes
<code>Txn.receiver()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.amount()</code>	<code>TealType.uint64</code>	in microAlgos
<code>Txn.close_remainder_to()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.vote_pk()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.selection_pk()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.vote_first()</code>	<code>TealType.uint64</code>	
<code>Txn.vote_last()</code>	<code>TealType.uint64</code>	
<code>Txn.vote_key_dilution()</code>	<code>TealType.uint64</code>	
<code>Txn.type()</code>	<code>TealType.bytes</code>	
<code>Txn.type_enum()</code>	<code>TealType.uint64</code>	see table below
<code>Txn.xfer_asset()</code>	<code>TealType.uint64</code>	asset ID
<code>Txn.asset_amount()</code>	<code>TealType.uint64</code>	value in Asset's units
<code>Txn.asset_sender()</code>	<code>TealType.bytes</code>	32 byte address, causes clawback of all value if sender
<code>Txn.asset_receiver()</code>	<code>TealType.bytes</code>	32 byte address

Continue

Table 1 – continued from previous page

Operator	Type	Notes
<code>Txn.asset_close_to()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.group_index()</code>	<code>TealType.uint64</code>	position of this transaction within a transaction group
<code>Txn.tx_id()</code>	<code>TealType.bytes</code>	the computed ID for this transaction, 32 bytes
<code>Txn.application_id()</code>	<code>TealType.uint64</code>	
<code>Txn.on_completion()</code>	<code>TealType.uint64</code>	
<code>Txn.approval_program()</code>	<code>TealType.bytes</code>	
<code>Txn.clear_state_program()</code>	<code>TealType.bytes</code>	
<code>Txn.rekey_to()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.config_asset()</code>	<code>TealType.uint64</code>	
<code>Txn.config_asset_total()</code>	<code>TealType.uint64</code>	
<code>Txn.config_asset_decimals()</code>	<code>TealType.uint64</code>	
<code>Txn.config_asset_default_frozen()</code>	<code>TealType.uint64</code>	
<code>Txn.config_asset_unit_name()</code>	<code>TealType.bytes</code>	
<code>Txn.config_asset_name()</code>	<code>TealType.bytes</code>	
<code>Txn.config_asset_url()</code>	<code>TealType.bytes</code>	
<code>Txn.config_asset_metadata_hash()</code>	<code>TealType.bytes</code>	
<code>Txn.config_asset_manager()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.config_asset_reserve()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.config_asset_freeze()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.config_asset_clawback()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.freeze_asset()</code>	<code>TealType.uint64</code>	
<code>Txn.freeze_asset_account()</code>	<code>TealType.bytes</code>	32 byte address
<code>Txn.freeze_asset_frozen()</code>	<code>TealType.uint64</code>	
<code>Txn.application_args()</code>	<code>TealType.bytes[]</code>	
<code>Txn.accounts()</code>	<code>TealType.bytes[]</code>	

The `Txn.type_enum()` values can be checked using the `TxnType` enum:

Value	Numerical Value	Type String	Description
<code>TxnType.Unknown</code>	0	unkown	unknown type, invalid
<code>TxnType.Payment</code>	1	pay	payment
<code>TxnType.KeyRegistration</code>	2	keyreg	key registration
<code>TxnType.AssetConfig</code>	3	acfg	asset config
<code>TxnType.AssetTransfer</code>	4	axfer	asset transfer
<code>TxnType.AssetFreeze</code>	5	afrz	asset freeze
<code>TxnType.ApplicationCall</code>	6	appl	application call

7.1.1 Atomic Tranfers

Atomic Transfers are irreducible batch transactions that allow groups of transactions to be submitted at one time. If any of the transactions fail, then all the transactions will fail. PyTeal allows programs to access information about the transactions in an atomic transfer group using the `Gtxn` object. This object acts like a list of `TxnObject`, meaning all of the above transaction fields on `Txn` are available on the elements of `Gtxn`. For example:

```
Gtxn[0].sender() # get the sender of the first transaction in the atomic transfer_
↪group
Gtxn[1].receiver() # get the receiver of the second transaction in the atomic_
↪transfer group
```

`Gtxn` is zero-indexed and the maximum size of an atomic transfer group is 16.

7.2 Global Parameters

Information about the current state of the blockchain can be obtained using the following *Global* expressions:

Operator	Type	Notes
<code>Global.min_txn_fee()</code>	<code>TealType.uint64</code>	in microAlgos
<code>Global.min_balance()</code>	<code>TealType.uint64</code>	in mircoAlgos
<code>Global.max_txn_life()</code>	<code>TealType.uint64</code>	number of rounds
<code>Global.zero_address()</code>	<code>TealType.bytes</code>	32 byte address of all zero bytes
<code>Global.group_size()</code>	<code>TealType.uint64</code>	number of txns in this atomic transaction group, At least 1
<code>Global.logic_sig_version()</code>	<code>TealType.uint64</code>	the maximum supported TEAL version
<code>Global.round()</code>	<code>TealType.uint64</code>	the current round number
<code>Global.latest_timestamp()</code>	<code>TealType.uint64</code>	the latest confirmed block UNIX timestamp
<code>Global.current_application_id()</code>	<code>TealType.uint64</code>	the ID of the current application executing

Cryptographic Primitives

Algorand Smart Contracts support 4 cryptographic primitives, including 3 cryptographic hash functions and 1 digital signature verification. Each of these cryptographic primitives is associated with a cost, which is a number indicating its relative performance overhead comparing with simple teal operations such as addition and subtraction. All TEAL opcodes except crypto primitives have cost *1*. Below is how you express cryptographic primitives in PyTeal:

Operator	Cost	Description
Sha256(e)	7	<i>SHA-256</i> hash function, produces 32 bytes
Keccak256(e)	26	<i>Keccak-256</i> hash function, produces 32 bytes
Sha512_256(e)	9	<i>SHA512-256</i> hash function, produces 32 bytes
Ed25519Verify(d, s, p)	1900	1 if <i>s</i> is the signature of <i>d</i> signed by <i>p</i> (PK), else 0

These cryptographic primitives cover the most used ones in blockchains and cryptocurrencies. For example, Bitcoin uses *SHA-256* for creating Bitcoin addresses; Algorand uses *ed25519* signature scheme for authorization and uses *SHA512-256* hash function for creating contract account addresses from TEAL bytecode.

PyTeal provides several control flow expressions to chain together multiple expressions and to conditionally evaluate expressions.

9.1 Exiting the Program: Return

The `Return` expression causes the program to exit immediately. It takes a single argument, which is the success value that the program should have. For example, `Return(Int(0))` causes the program to immediately fail, and `Return(Int(1))` causes the program to immediately succeed. Since the presence of a `Return` statement causes the program to exit, no operations after it will be executed.

9.2 Chaining Expressions: Seq

The `Seq` expression can be used to create a sequence of multiple expressions. It takes a single argument, which is a list of expressions to include in the sequence. For example:

```
Seq([
    App.globalPut(Bytes("creator"), Txn.sender()),
    Return(Int(1))
])
```

A `Seq` expression will take on the value of its last expression. Additionally, all expressions in a `Seq` expression, except the last one, must not return anything (e.g. evaluate to `TealType.none`). This restriction is in place because intermediate values must not add things to the TEAL stack. As a result, the following is an invalid sequence:

Listing 1: Invalid Seq expression

```
Seq([
    Txn.sender(),
    Return(Int(1))
])
```

If you must include an operation that returns a value in the earlier part of a sequence, you can wrap the value in a *Pop* expression to discard it. For example,

```
Seq([
    Pop(Txn.sender()),
    Return(Int(1))
])
```

9.3 Simple Branching: *If*

In an *If* expression,

```
If(test-expr, then-expr, else-expr)
```

the *test-expr* is always evaluated and needs to be typed `TealType.uint64`. If it results in a value greater than 0, then the *then-expr* is evaluated. Otherwise, *else-expr* is evaluated. Note that *then-expr* and *else-expr* must evaluate to the same type (e.g. both `TealType.uint64`).

You may also invoke an *If* expression without an *else-expr*:

```
If(test-expr, then-expr)
```

In this case, *then-expr* must be typed `TealType.none`.

9.4 Checking Conditions: *Assert*

The *Assert* expression can be used to ensure that conditions are met before continuing the program. The syntax for *Assert* is:

```
Assert(test-expr)
```

If *test-expr* is always evaluated and must be typed `TealType.uint64`. If *test-expr* results in a value greater than 0, the program continues. Otherwise, the program immediately exits and indicates that it encountered an error.

Example:

```
Assert(Txn.type_enum() == TxnType.Payment)
```

The above example will cause the program to immediately fail with an error if the transaction type is not a payment.

9.5 Chaining Tests: *Cond*

A *Cond* expression chains a series of tests to select a result expression. The syntax of *Cond* is:

```
Cond([test-expr-1, body-1],
     [test-expr-2, body-2],
     . . . )
```

Each *test-expr* is evaluated in order. If it produces 0, the paired *body* is ignored, and evaluation proceeds to the next *test-expr*. As soon as a *test-expr* produces a true value (> 0), its *body* is evaluated to produce the value

for this `Cond` expression. If none of `test-expr` s evaluates to a true value, the `Cond` expression will be evaluated to `err`, a TEAL opcode that causes the runtime panic.

In a `Cond` expression, each `test-expr` needs to be typed `TealType.uint64`. A `body` could be typed either `TealType.uint64` or `TealType.bytes`. However, all `body` s must have the same data type. Otherwise, a `TealTypeError` is triggered.

Example:

```
Cond([Global.group_size() == Int(5), bid],
     [Global.group_size() == Int(4), redeem],
     [Global.group_size() == Int(1), wrapup])
```

This PyTeal code branches on the size of the atomic transaction group.

CHAPTER 10

State Access and Manipulation

PyTeal can be used to write [Stateful Algorand Smart Contracts](#) as well. Stateful contracts, also known as applications, can access and manipulate state on the Algorand blockchain.

State consists of key-value pairs, where keys are byte slices and values can be integers or byte slices. There are multiple types of state that an application can use.

10.1 State Operation Table

Context	Write	Read	Delete	Check If Exists
Current App Global	<code>App.globalPut</code>	<code>App.globalGet</code>	<code>App.globalDel</code>	<code>App.globalGetEx</code>
Current App Local	<code>App.localPut</code>	<code>App.localGet</code>	<code>App.localDel</code>	<code>App.localGetEx</code>
Other App Global		<code>App.globalGetEx</code>		<code>App.globalGetEx</code>
Other App Local		<code>App.localGetEx</code>		<code>App.localGetEx</code>

10.2 Global State

Global state consists of key-value pairs that are stored in the application's global context. It can be manipulated as follows:

10.2.1 Writing

To write to global state, use the `App.globalPut` function. The first argument is the key to write to, and the second argument is the value to write. For example:

```
App.globalPut(Bytes("status"), Bytes("active")) # write a byte slice
App.globalPut(Bytes("total supply"), Int(100)) # write a uint64
```

10.2.2 Reading

To read from global state, use the `App.globalGet` function. The only argument it takes is the key to read from. For example:

```
App.globalGet(Bytes("status"))
App.globalGet(Bytes("total supply"))
```

If you try to read from a key that does not exist in your app's global state, the integer `0` is returned.

10.2.3 Deleting

To delete a key from global state, use the `App.globalDel` function. The only argument it takes is the key to delete. For example:

```
App.globalDel(Bytes("status"))
App.globalDel(Bytes("total supply"))
```

If you try to delete a key that does not exist in your app's global state, nothing happens.

10.3 Local State

Local state consists of key-value pairs that are stored in a unique context for each account that has opted into your application. As a result, you will need to specify an account when manipulating local state. This is done by passing in an integer that corresponds to an account. The integer `0` is a special case that refers to the sender of the application call transaction. The integer `1` refers to the first element in `Txn.accounts`, `2` refers to the second element, and so on.

10.3.1 Writing

To write to the local state of an account, use the `App.localPut` function. The first argument is an integer corresponding to the account to write to, the second argument is the key to write to, and the third argument is the value to write. For example:

```
App.localPut(Int(0), Bytes("role"), Bytes("admin")) # write a byte slice to the sender
↳ 's account
App.localPut(Int(0), Bytes("balance"), Int(10)) # write a uint64 to the sender's
↳ account
App.localPut(Int(1), Bytes("balance"), Int(10)) # write a uint64 to Txn.accounts[0]
```

Note: It is only possible to write to the local state of an account if that account has opted into your application. If the account has not opted in, the program will fail with an error. The function `App.optedIn` can be used to check if an account has opted into an app.

10.3.2 Reading

To read from the local state of an account, use the `App.localGet` function. The first argument is an integer corresponding to the account to read from and the second argument is the key to read. For example:

```
App.localGet(Int(0), Bytes("role")) # read from the sender's account
App.localGet(Int(0), Bytes("balance")) # read from the the sender's account
App.localGet(Int(1), Bytes("balance")) # read from Txn.accounts[0]
```

If you try to read from a key that does not exist in your app's global state, the integer `0` is returned.

10.3.3 Deleting

To delete a key from local state of an account, use the `App.localDel` function. The first argument is an integer corresponding to the account and the second argument is the key to delete. For example:

```
App.localDel(Int(0), Bytes("role")) # delete "role" from the sender's account
App.localDel(Int(0), Bytes("balance")) # delete "balance" from the the sender's_
↪account
App.localDel(Int(1), Bytes("balance")) # delete "balance" from Txn.accounts[0]
```

If you try to delete a key that does not exist in the account's local state, nothing happens.

10.4 External State

The above functions allow an app to read and write state in its own context. Additionally, it's possible for applications to read state written by other applications. This is possible using the `App.globalGetEx` and `App.localGetEx` functions.

Similar to local state access, to use these functions you need to pass in an integer that represents which application to read from. The integer `0` is a special case that refers to the current application. The integer `1` refers to the first element in `Txn.ForeignApps`, `2` refers to the second element, and so on. Note that the transaction field `ForeignApps` is not accessible from TEAL at this time.

Unlike the other state access functions, `App.globalGetEx` and `App.localGetEx` return a `MaybeValue`. This value cannot be used directly, but has methods `MaybeValue.hasValue()` and `MaybeValue.value()`. If the key being accessed exists in the context of the app being read, `hasValue()` will return `1` and `value()` will return its value. Otherwise, `hasValue()` and `value()` will return `0`.

Note: Even though the `MaybeValue` returned by `App.globalGetEx` and `App.localGetEx` cannot be used directly, it **must** be included in the application before `hasValue()` and `value()` are called on it. You will probably want to use `Seq` to do this.

Since these functions are the only way to check whether a key exists, it can be useful to use them in the current application's context too.

10.4.1 External Global

To read a value from the global state of another application, use the `App.globalGetEx` function. The first argument is an integer corresponding to the application to read from and the second argument is the key to read. For example:

```
# get "status" from the current global context
# if "status" has not been set, returns "none"
myStatus = App.globalGetEx(Int(0), Bytes("status"))
Seq([
    myStatus,
    If(myStatus.hasValue(), myStatus.value(), Bytes("none"))
])
```

(continues on next page)

(continued from previous page)

```

# get "status" from the global context of Txn.ForeignApps[0]
# if "status" has not been set, returns "none"
otherStatus = App.globalGetEx(Int(1), Bytes("status"))
Seq([
    otherStatus,
    If(otherStatus.hasValue(), otherStatus.value(), Bytes("none"))
])

# get "total supply" from the global context of Txn.ForeignApps[0]
# if "total supply" has not been set, returns the default value of 0
otherSupply = App.globalGetEx(Int(1), Bytes("total supply"))
Seq([
    otherSupply,
    otherSupply.value()
])

```

10.4.2 External Local

To read a value from an account's local state for another application, use the `App.localGetEx` function. The first argument is an integer corresponding to the account to read from (in the same format as `App.localGet`), the second argument is an integer corresponding to the application to read from, and the third argument is the key to read. For example:

```

# get "role" from the sender's local state for the current account
# if "role" has not been set, returns "none"
myAppSenderRole = App.localGetEx(Int(0), Int(0), Bytes("role"))
Seq([
    myAppSenderRole,
    If(myAppSenderRole.hasValue(), myAppSenderRole.value(), Bytes("none"))
])

# get "role" from the local state of Txn.accounts[0] for the current account
# if "role" has not been set, returns "none"
myAppOtherAccountRole = App.localGetEx(Int(1), Int(0), Bytes("role"))
Seq([
    myAppOtherAccountRole,
    If(myAppOtherAccountRole.hasValue(), myAppOtherAccountRole.value(), Bytes("none"))
])

# get "role" from the sender's local state for Txn.ForeignApps[0]
# if "role" has not been set, returns "none"
otherAppSenderRole = App.localGetEx(Int(0), Int(1), Bytes("role"))
Seq([
    otherAppSenderRole,
    If(otherAppSenderRole.hasValue(), otherAppSenderRole.value(), Bytes("none"))
])

# get "role" from the local state of Txn.accounts[0] for Txn.ForeignApps[0]
# if "role" has not been set, returns "none"
otherAppOtherAccountRole = App.localGetEx(Int(1), Int(1), Bytes("role"))
Seq([
    otherAppOtherAccountRole,
    If(otherAppOtherAccountRole.hasValue(), otherAppOtherAccountRole.value(), Bytes(
        ↪ "none"))
])

```

(continues on next page)

(continued from previous page)

1)

CHAPTER 11

PyTeal Package

`pyteal.Txn = <pyteal.TxnObject object>`
The current transaction being evaluated.

`pyteal.Gtxn = <pyteal.TxnGroup object>`
The current group of transactions being evaluated.

class `pyteal.Expr`
Bases: `abc.ABC`

Abstract base class for PyTeal expressions.

And (*other: pyteal.Expr*) $\rightarrow$ `pyteal.Expr`
Take the logical And of this expression and another one.

This expression must evaluate to uint64.

This is the same as using `And()` with two arguments.

Or (*other: pyteal.Expr*) $\rightarrow$ `pyteal.Expr`
Take the logical Or of this expression and another one.

This expression must evaluate to uint64.

This is the same as using `Or()` with two arguments.

type_of () $\rightarrow$ `pyteal.TealType`
Get the return type of this expression.

class `pyteal.LeafExpr`
Bases: `pyteal.Expr`

Leaf expression base class.

class `pyteal.Addr` (*address: str*)
Bases: `pyteal.LeafExpr`

An expression that represents an Algorand address.

__init__ (*address: str*) $\rightarrow$ `None`
Create a new Addr expression.

Parameters **address** – A string containing a valid base32 Algorand address

type_of()

Get the return type of this expression.

class `pyteal.Bytes(*args)`
Bases: `pyteal.LeafExpr`

An expression that represents a byte string.

__init__(*args) → None
Create a new byte string.

Depending on the encoding, there are different arguments to pass:

For UTF-8 strings: Pass the string as the only argument. For example, `Bytes("content")`.

For base16, base32, or base64 strings: Pass the base as the first argument and the string as the second argument. For example, `Bytes("base16", "636F6E74656E74")`, `Bytes("base32", "ORFDPQ6ARJK")`, `Bytes("base64", "Y29udGVudA==")`.

Special case for base16: The prefix “0x” may be present in a base16 byte string. For example, `Bytes("base16", "0x636F6E74656E74")`.

type_of()

Get the return type of this expression.

class `pyteal.Err`
Bases: `pyteal.LeafExpr`

Expression that causes the program to immediately fail when executed.

type_of()

Get the return type of this expression.

class `pyteal.Int(value: int)`
Bases: `pyteal.LeafExpr`

An expression that represents a uint64.

__init__(value: int) → None
Create a new uint64.

Parameters **value** – The integer value this uint64 will represent. Must be a positive value less than 2^{64} .

type_of()

Get the return type of this expression.

class `pyteal.EnumInt(name: str)`
Bases: `pyteal.LeafExpr`

An expression that represents uint64 enum values.

__init__(name: str) → None
Create an expression to reference a uint64 enum value.

Parameters **name** – The name of the enum value.

type_of()

Get the return type of this expression.

class `pyteal.Arg(index: int)`
Bases: `pyteal.LeafExpr`

An expression to get an argument when running in signature verification mode.

`__init__(index: int) → None`

Get an argument for this program.

Should only be used in signature verification mode. For application mode arguments, see [TxnObject.application_args](#).

Parameters `index` – The integer index of the argument to get. Must be between 0 and 255 inclusive.

`type_of()`

Get the return type of this expression.

class `pyteal.TxnType`

Bases: `object`

Enum of all possible transaction types.

`ApplicationCall = <pyteal.EnumInt object>`

`AssetConfig = <pyteal.EnumInt object>`

`AssetFreeze = <pyteal.EnumInt object>`

`AssetTransfer = <pyteal.EnumInt object>`

`KeyRegistration = <pyteal.EnumInt object>`

`Payment = <pyteal.EnumInt object>`

`Unknown = <pyteal.EnumInt object>`

class `pyteal.TxnField(id: int, name: str, type: pyteal.TealType)`

Bases: `enum.Enum`

An enumeration.

`accounts = (28, 'Accounts', <TealType.bytes: 1>)`

`amount = (8, 'Amount', <TealType.uint64: 0>)`

`application_args = (26, 'ApplicationArgs', <TealType.bytes: 1>)`

`application_id = (24, 'ApplicationID', <TealType.uint64: 0>)`

`approval_program = (30, 'ApprovalProgram', <TealType.bytes: 1>)`

`asset_amount = (18, 'AssetAmount', <TealType.uint64: 0>)`

`asset_close_to = (21, 'AssetCloseTo', <TealType.bytes: 1>)`

`asset_receiver = (20, 'AssetReceiver', <TealType.bytes: 1>)`

`asset_sender = (19, 'AssetSender', <TealType.bytes: 1>)`

`clear_state_program = (31, 'ClearStateProgram', <TealType.bytes: 1>)`

`close_remainder_to = (9, 'CloseRemainderTo', <TealType.bytes: 1>)`

`config_asset = (33, 'ConfigAsset', <TealType.uint64: 0>)`

`config_asset_clawback = (44, 'ConfigAssetClawback', <TealType.bytes: 1>)`

`config_asset_decimals = (35, 'ConfigAssetDecimals', <TealType.uint64: 0>)`

`config_asset_default_frozen = (36, 'ConfigAssetDefaultFrozen', <TealType.uint64: 0>)`

`config_asset_freeze = (43, 'ConfigAssetFreeze', <TealType.bytes: 1>)`

`config_asset_manager = (41, 'ConfigAssetManager', <TealType.bytes: 1>)`

```
config_asset_metadata_hash = (40, 'ConfigAssetMetadataHash', <TealType.bytes: 1>)
config_asset_name = (38, 'ConfigAssetName', <TealType.bytes: 1>)
config_asset_reserve = (42, 'ConfigAssetReserve', <TealType.bytes: 1>)
config_asset_total = (34, 'ConfigAssetTotal', <TealType.uint64: 0>)
config_asset_unit_name = (37, 'ConfigAssetUnitName', <TealType.bytes: 1>)
config_asset_url = (39, 'ConfigAssetURL', <TealType.bytes: 1>)
fee = (1, 'Fee', <TealType.uint64: 0>)
first_valid = (2, 'FirstValid', <TealType.uint64: 0>)
first_valid_time = (3, 'FirstValidTime', <TealType.uint64: 0>)
freeze_asset = (45, 'FreezeAsset', <TealType.uint64: 0>)
freeze_asset_account = (46, 'FreezeAssetAccount', <TealType.bytes: 1>)
freeze_asset_frozen = (47, 'FreezeAssetFrozen', <TealType.uint64: 0>)
group_index = (22, 'GroupIndex', <TealType.uint64: 0>)
last_valid = (4, 'LastValid', <TealType.uint64: 0>)
lease = (6, 'Lease', <TealType.bytes: 1>)
note = (5, 'Note', <TealType.bytes: 1>)
num_accounts = (2, 'NumAccounts', <TealType.uint64: 0>)
num_app_args = (27, 'NumAppArgs', <TealType.uint64: 0>)
on_completion = (25, 'OnCompletion', <TealType.uint64: 0>)
receiver = (7, 'Receiver', <TealType.bytes: 1>)
rekey_to = (32, 'RekeyTo', <TealType.bytes: 1>)
selection_pk = (11, 'SelectionPK', <TealType.bytes: 1>)
sender = (0, 'Sender', <TealType.bytes: 1>)
tx_id = (23, 'TxID', <TealType.bytes: 1>)
type = (15, 'Type', <TealType.bytes: 1>)
type_enum = (16, 'TypeEnum', <TealType.uint64: 0>)
type_of() → pyteal.TealType
vote_first = (12, 'VoteFirst', <TealType.uint64: 0>)
vote_key_dilution = (14, 'VoteKeyDilution', <TealType.uint64: 0>)
vote_last = (13, 'VoteLast', <TealType.uint64: 0>)
vote_pk = (10, 'VotePK', <TealType.bytes: 1>)
xfer_asset = (17, 'XferAsset', <TealType.uint64: 0>)
```

class `pyteal.TxnExpr` (*field: pyteal.TxnField*)
Bases: `pyteal.LeafExpr`

An expression that accesses a transaction field from the current transaction.

type_of()
Get the return type of this expression.

```

class pyteal.TxnaExpr (field: pyteal.TxnField, index: int)
    Bases: pyteal.LeafExpr

    An expression that accesses a transaction array field from the current transaction.

    type_of()
        Get the return type of this expression.

class pyteal.TxnArray (txnObject: pyteal.TxnObject, accessField: pyteal.TxnField, lengthField: pyteal.TxnField)
    Bases: pyteal.Array

    Represents a transaction array field.

    __getitem__ (index: int) → pyteal.TxnaExpr
        Get the value at a given index in this array.

    length() → pyteal.TxnExpr
        Get the length of the array.

class pyteal.TxnObject (txnType: Callable[[pyteal.TxnField], pyteal.TxnExpr], txnaType: Callable[[pyteal.TxnField, int], pyteal.TxnaExpr])
    Bases: object

    Represents a transaction and its fields.

    accounts
        Application call accounts array.

        Type TxnArray

    amount() → pyteal.TxnExpr
        Get the amount of the transaction in micro Algos.

        Only set when type_enum() is TxnType.Payment.

        For more information, see https://developer.algorand.org/docs/reference/transactions/#amount

    application_args
        Application call arguments array.

        Type TxnArray

    application_id() → pyteal.TxnExpr
        Get the application ID from the ApplicationCall portion of the current transaction.

        Only set when type_enum() is TxnType.ApplicationCall.

    approval_program() → pyteal.TxnExpr
        Get the approval program.

        Only set when type_enum() is TxnType.ApplicationCall.

    asset_amount() → pyteal.TxnExpr
        Get the amount of the asset being transferred, measured in the asset's units.

        Only set when type_enum() is TxnType.AssetTransfer.

        For more information, see https://developer.algorand.org/docs/reference/transactions/#assetamount

    asset_close_to() → pyteal.TxnExpr
        Get the closeout address of the asset transfer.

        Only set when type_enum() is TxnType.AssetTransfer.

        For more information, see https://developer.algorand.org/docs/reference/transactions/#assetclose\_to

```

asset_receiver() → pyteal.TxnExpr

Get the recipient of the asset transfer.

Only set when *type_enum()* is *TxnType.AssetTransfer*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#assetreceiver>

asset_sender() → pyteal.TxnExpr

Get the 32 byte address of the subject of clawback.

Only set when *type_enum()* is *TxnType.AssetTransfer*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#assetsender>

clear_state_program() → pyteal.TxnExpr

Get the clear state program.

Only set when *type_enum()* is *TxnType.ApplicationCall*.

close_remainder_to() → pyteal.TxnExpr

Get the 32 byte address of the CloseRemainderTo field.

Only set when *type_enum()* is *TxnType.Payment*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#closeremainderto>

config_asset() → pyteal.TxnExpr

Get the asset ID in asset config transaction.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#configasset>

config_asset_clawback() → pyteal.TxnExpr

Get the 32 byte asset clawback address.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#clawbackaddr>

config_asset_decimals() → pyteal.TxnExpr

Get the number of digits to display after the decimal place when displaying the asset.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#decimals>

config_asset_default_frozen() → pyteal.TxnExpr

Check if the asset's slots are frozen by default or not.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#defaultfrozen>

config_asset_freeze() → pyteal.TxnExpr

Get the 32 byte asset freeze address.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#freezeaddr>

config_asset_manager() → pyteal.TxnExpr

Get the 32 byte asset manager address.

Only set when *type_enum()* is *TxnType.AssetConfig*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#manageraddr>

config_asset_metadata_hash () → pyteal.TxnExpr
Get the 32 byte commitment to some unspecified asset metadata.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#metadatabash>

config_asset_name () → pyteal.TxnExpr
Get the asset name.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#assetname>

config_asset_reserve () → pyteal.TxnExpr
Get the 32 byte asset reserve address.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#reserveaddr>

config_asset_total () → pyteal.TxnExpr
Get the total number of units of this asset created.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#total>

config_asset_unit_name () → pyteal.TxnExpr
Get the unit name of the asset.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#unitname>

config_asset_url () → pyteal.TxnExpr
Get the asset URL.
Only set when *type_enum* () is *TxnType.AssetConfig*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#url>

fee () → pyteal.TxnExpr
Get the transaction fee in micro Algos.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#fee>

first_valid () → pyteal.TxnExpr
Get the first valid round number.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#firstvalid>

freeze_asset () → pyteal.TxnExpr
Get the asset ID being frozen or un-frozen.
Only set when *type_enum* () is *TxnType.AssetFreeze*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#freezeasset>

freeze_asset_account () → pyteal.TxnExpr
Get the 32 byte address of the account whose asset slot is being frozen or un-frozen.
Only set when *type_enum* () is *TxnType.AssetFreeze*.
For more information, see <https://developer.algorand.org/docs/reference/transactions/#freezeaccount>

freeze_asset_frozen() → pyteal.TxnExpr

Get the new frozen value for the asset.

Only set when *type_enum()* is *TxnType.AssetFreeze*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#assetfrozen>

group_index() → pyteal.TxnExpr

Get the position of the transaction within the atomic transaction group.

A stand-alone transaction is implicitly element 0 in a group of 1.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#group>

last_valid() → pyteal.TxnExpr

Get the last valid round number.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#lastvalid>

lease() → pyteal.TxnExpr

Get the transaction lease.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#lease>

note() → pyteal.TxnExpr

Get the transaction note.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#note>

on_completion() → pyteal.TxnExpr

Get the on completion action from the ApplicationCall portion of the transaction.

Only set when *type_enum()* is *TxnType.ApplicationCall*.

receiver() → pyteal.TxnExpr

Get the 32 byte address of the receiver.

Only set when *type_enum()* is *TxnType.Payment*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#receiver>

rekey_to() → pyteal.TxnExpr

Get the sender's new 32 byte AuthAddr.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#rekeyto>

selection_pk() → pyteal.TxnExpr

Get the VRF public key.

Only set when *type_enum()* is *TxnType.KeyRegistration*.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#selectionpk>

sender() → pyteal.TxnExpr

Get the 32 byte address of the sender.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#sender>

tx_id() → pyteal.TxnExpr

Get the 32 byte computed ID for the transaction.

type() → pyteal.TxnExpr

Get the type of this transaction as a byte string.

In most cases it is preferable to use *type_enum()* instead.

For more information, see <https://developer.algorand.org/docs/reference/transactions/#type>

type_enum() → `pyteal.TxnExpr`
 Get the type of this transaction.
 See [TxnType](#) for possible values.

vote_first() → `pyteal.TxnExpr`
 Get the first round that the participation key is valid.
 Only set when `type_enum()` is `TxnType.KeyRegistration`.
 For more information, see <https://developer.algorand.org/docs/reference/transactions/#votefirst>

vote_key_dilution() → `pyteal.TxnExpr`
 Get the dilution for the 2-level participation key.
 Only set when `type_enum()` is `TxnType.KeyRegistration`.
 For more information, see <https://developer.algorand.org/docs/reference/transactions/#votekeydilution>

vote_last() → `pyteal.TxnExpr`
 Get the last round that the participation key is valid.
 Only set when `type_enum()` is `TxnType.KeyRegistration`.
 For more information, see <https://developer.algorand.org/docs/reference/transactions/#votelast>

vote_pk() → `pyteal.TxnExpr`
 Get the root participation public key.
 Only set when `type_enum()` is `TxnType.KeyRegistration`.
 For more information, see <https://developer.algorand.org/docs/reference/transactions/#votepk>

xfer_asset() → `pyteal.TxnExpr`
 Get the ID of the asset being transferred.
 Only set when `type_enum()` is `TxnType.AssetTransfer`.
 For more information, see <https://developer.algorand.org/docs/reference/transactions/#xferasset>

class `pyteal.GtxnExpr` (*txnIndex: int, field: pyteal.TxnField*)
 Bases: `pyteal.TxnExpr`
 An expression that accesses a transaction field from a transaction in the current group.

class `pyteal.GtxnaExpr` (*txnIndex: int, field: pyteal.TxnField, index: int*)
 Bases: `pyteal.TxnaExpr`
 An expression that accesses a transaction array field from a transaction in the current group.

class `pyteal.TxnGroup`
 Bases: `object`
 Represents a group of transactions.
__getitem__ (*txnIndex: int*) → `pyteal.TxnObject`

class `pyteal.Global` (*field: pyteal.GlobalField*)
 Bases: `pyteal.LeafExpr`
 An expression that accesses a global property.
classmethod **current_application_id**() → `pyteal.Global`
 Get the ID of the current application executing.
 Fails if no application is executing.

```
classmethod group_size() → pyteal.Global
    Get the number of transactions in this atomic transaction group.

    This will be at least 1.

classmethod latest_timestamp() → pyteal.Global
    Get the latest confirmed block UNIX timestamp.

    Fails if negative.

classmethod logic_sig_version() → pyteal.Global
    Get the maximum supported TEAL version.

classmethod max_txn_life() → pyteal.Global
    Get the maximum number of rounds a transaction can have.

classmethod min_balance() → pyteal.Global
    Get the minumum balance in micro Algos.

classmethod min_txn_fee() → pyteal.Global
    Get the minumum transaction fee in micro Algos.

classmethod round() → pyteal.Global
    Get the current round number.

type_of()
    Get the return type of this expression.

classmethod zero_address() → pyteal.Global
    Get the 32 byte zero address.

class pyteal.GlobalField(id: int, name: str, type: pyteal.TealType)
    Bases: enum.Enum

    An enumeration.

    current_app_id = (8, 'CurrentApplicationID', <TealType.uint64: 0>)
    group_size = (4, 'GroupSize', <TealType.uint64: 0>)
    latest_timestamp = (7, 'LatestTimestamp', <TealType.uint64: 0>)
    logic_sig_version = (5, 'LogicSigVersion', <TealType.uint64: 0>)
    max_txn_life = (2, 'MaxTxnLife', <TealType.uint64: 0>)
    min_balance = (1, 'MinBalance', <TealType.uint64: 0>)
    min_txn_fee = (0, 'MinTxnFee', <TealType.uint64: 0>)
    round = (6, 'Round', <TealType.uint64: 0>)
    type_of() → pyteal.TealType
    zero_address = (3, 'ZeroAddress', <TealType.bytes: 1>)

class pyteal.App(field: pyteal.AppField, args)
    Bases: pyteal.LeafExpr

    An expression related to applications.

    classmethod globalDel(key: pyteal.Expr) → pyteal.App
        Delete a key from the global state of the current application.

        Parameters key – The key to delete from the global application state. Must evaluate to bytes.
```

classmethod **globalGet** (*key: pyteal.Expr*) → *pyteal.App*

Read from the global state of the current application.

Parameters **key** – The key to read from the global application state. Must evaluate to bytes.

classmethod **globalGetEx** (*app: pyteal.Expr, key: pyteal.Expr*) → *pyteal.MaybeValue*

Read from the global state of an application.

Parameters

- **app** – An index into *Txn.ForeignApps* that corresponds to the application to read from. Must evaluate to uint64.
- **key** – The key to read from the global application state. Must evaluate to bytes.

classmethod **globalPut** (*key: pyteal.Expr, value: pyteal.Expr*) → *pyteal.App*

Write to the global state of the current application.

Parameters

- **key** – The key to write in the global application state. Must evaluate to bytes.
- **value** – The value to write in the global application state. Can evaluate to any type.

classmethod **id** () → *pyteal.Global*

Get the ID of the current running application.

This is the same as *Global.current_application_id()*.

classmethod **localDel** (*account: pyteal.Expr, key: pyteal.Expr*) → *pyteal.App*

Delete a key from an account's local state for the current application.

Parameters

- **account** – An index into *Txn.Accounts* that corresponds to the account from which the key should be deleted. Must evaluate to uint64.
- **key** – The key to delete from the account's local state. Must evaluate to bytes.

classmethod **localGet** (*account: pyteal.Expr, key: pyteal.Expr*) → *pyteal.App*

Read from an account's local state for the current application.

Parameters

- **account** – An index into *Txn.Accounts* that corresponds to the account to read from. Must evaluate to uint64.
- **key** – The key to read from the account's local state. Must evaluate to bytes.

classmethod **localGetEx** (*account: pyteal.Expr, app: pyteal.Expr, key: pyteal.Expr*) → *pyteal.MaybeValue*

Read from an account's local state for an application.

Parameters

- **account** – An index into *Txn.Accounts* that corresponds to the account to read from. Must evaluate to uint64.
- **app** – The ID of the application being checked. Must evaluate to uint64.
- **key** – The key to read from the account's local state. Must evaluate to bytes.

classmethod **localPut** (*account: pyteal.Expr, key: pyteal.Expr, value: pyteal.Expr*) → *pyteal.App*

Write to an account's local state for the current application.

Parameters

- **account** – An index into Txn.Accounts that corresponds to the account to write to. Must evaluate to uint64.
- **key** – The key to write in the account’s local state. Must evaluate to bytes.
- **value** – The value to write in the account’s local state. Can evaluate to any type.

classmethod **optedIn** (*account: pyteal.Expr, app: pyteal.Expr*) → pyteal.App

Check if an account has opted in for an application.

Parameters

- **account** – An index into Txn.Accounts that corresponds to the account to check. Must evaluate to uint64.
- **app** – The ID of the application being checked. Must evaluate to uint64.

type_of ()

Get the return type of this expression.

class **pyteal.AppField** (*op: pyteal.Op, type: pyteal.TealType*)

Bases: `enum.Enum`

Enum of app fields used to create [App](#) objects.

get_op () → pyteal.Op

globalDel = (<Op.app_global_del: 'app_global_del'>, <TealType.none: 3>)

globalGet = (<Op.app_global_get: 'app_global_get'>, <TealType.anytype: 2>)

globalGetEx = (<Op.app_global_get_ex: 'app_global_get_ex'>, <TealType.none: 3>)

globalPut = (<Op.app_global_put: 'app_global_put'>, <TealType.none: 3>)

localDel = (<Op.app_local_del: 'app_local_del'>, <TealType.none: 3>)

localGet = (<Op.app_local_get: 'app_local_get'>, <TealType.anytype: 2>)

localGetEx = (<Op.app_local_get_ex: 'app_local_get_ex'>, <TealType.none: 3>)

localPut = (<Op.app_local_put: 'app_local_put'>, <TealType.none: 3>)

optedIn = (<Op.app_opted_in: 'app_opted_in'>, <TealType.uint64: 0>)

type_of () → pyteal.TealType

class **pyteal.OnComplete**

Bases: `object`

An enum of values that *TxnObject.on_completion()* may return.

ClearState = <pyteal.EnumInt object>

CloseOut = <pyteal.EnumInt object>

DeleteApplication = <pyteal.EnumInt object>

NoOp = <pyteal.EnumInt object>

OptIn = <pyteal.EnumInt object>

UpdateApplication = <pyteal.EnumInt object>

class **pyteal.AssetHolding**

Bases: `object`

classmethod **balance** (*account: pyteal.Expr, asset: pyteal.Expr*) → pyteal.MaybeValue

Get the amount of an asset held by an account.

Parameters

- **account** – An index into Txn.Accounts that corresponds to the account to check. Must evaluate to uint64.
- **asset** – The ID of the asset to get. Must evaluate to uint64.

classmethod frozen (*account: pyteal.Expr, asset: pyteal.Expr*) → pyteal.MaybeValue

Check if an asset is frozen for an account.

Parameters

- **account** – An index into Txn.Accounts that corresponds to the account to check. Must evaluate to uint64.
- **asset** – The ID of the asset to check. Must evaluate to uint64.

class pyteal.AssetParam

Bases: object

classmethod clawback (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the clawback address for an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod decimals (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the number of decimals for an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod defaultFrozen (*asset: pyteal.Expr*) → pyteal.MaybeValue

Check if an asset is frozen by default.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod freeze (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the freeze address for an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod manager (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the manager commitment for an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod metadataHash (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the arbitrary commitment for an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod name (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the name of an asset.

Parameters asset – An index into Txn.ForeignAssets that corresponds to the asset to check. Must evaluate to uint64.

classmethod reserve (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the reserve address for an asset.

Parameters **asset** – An index into Txn.ForeignAssets that corresponds to the asset to check.
Must evaluate to uint64.

classmethod **total** (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the total number of units of an asset.

Parameters **asset** – An index into Txn.ForeignAssets that corresponds to the asset to check.
Must evaluate to uint64.

classmethod **unitName** (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the unit name of an asset.

Parameters **asset** – An index into Txn.ForeignAssets that corresponds to the asset to check.
Must evaluate to uint64.

classmethod **url** (*asset: pyteal.Expr*) → pyteal.MaybeValue

Get the URL of an asset.

Parameters **asset** – An index into Txn.ForeignAssets that corresponds to the asset to check.
Must evaluate to uint64.

class **pyteal.Array**

Bases: `abc.ABC`

Represents a variable length array of objects.

__getitem__ (*index: int*)

Get the value at a given index in this array.

length () → pyteal.Expr

Get the length of the array.

class **pyteal.Tmpl** (*op: pyteal.Op, type: pyteal.TealType, name: str*)

Bases: `pyteal.LeafExpr`

Template expression for creating placeholder values.

classmethod **Addr** (*placeholder: str*)

Create a new Addr template.

Parameters **placeholder** – The name to use for this template variable. Must start with `TMPL_` and only consist of uppercase alphanumeric characters and underscores.

classmethod **Bytes** (*placeholder: str*)

Create a new Bytes template.

Parameters **placeholder** – The name to use for this template variable. Must start with `TMPL_` and only consist of uppercase alphanumeric characters and underscores.

classmethod **Int** (*placeholder: str*)

Create a new Int template.

Parameters **placeholder** – The name to use for this template variable. Must start with `TMPL_` and only consist of uppercase alphanumeric characters and underscores.

type_of ()

Get the return type of this expression.

class **pyteal.Nonce** (*base: str, nonce: str, child: pyteal.Expr*)

Bases: `pyteal.Expr`

A meta expression only used to change the hash of a TEAL program.

__init__ (*base: str, nonce: str, child: pyteal.Expr*) → None

Create a new Nonce.

The Nonce expression behaves exactly like the child expression passed into it, except it uses the provided nonce string to alter its structure in a way that does not affect execution.

Parameters

- **base** – The base of the nonce. Must be one of utf8, base16, base32, or base64.
- **nonce** – An arbitrary nonce string that conforms to base.
- **child** – The expression to wrap.

type_of()

Get the return type of this expression.

class `pyteal.UnaryExpr` (*op: pyteal.Op, inputType: pyteal.TealType, outputType: pyteal.TealType, arg: pyteal.Expr*)

Bases: `pyteal.Expr`

An expression with a single argument.

type_of()

Get the return type of this expression.

`pyteal.Btoi` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Convert a byte string to a uint64.

`pyteal.Itob` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Convert a uint64 string to a byte string.

`pyteal.Len` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the length of a byte string.

`pyteal.Sha256` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the SHA-256 hash of a byte string.

`pyteal.Sha512_256` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the SHA-512/256 hash of a byte string.

`pyteal.Keccak256` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the KECCAK-256 hash of a byte string.

`pyteal.Not` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the logical inverse of a uint64.

If the argument is 0, then this will produce 1. Otherwise this will produce 0.

`pyteal.BitwiseNot` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the bitwise inverse of a uint64.

Produces $\sim$ arg.

`pyteal.Pop` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Pop a value from the stack.

`pyteal.Return` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Immediately exit the program with the given success value.

`pyteal.Balance` (*arg: pyteal.Expr*) → `pyteal.UnaryExpr`

Get the balance of a user in micro Algos.

Argument must be an index into Txn.Accounts that corresponds to the account to read from. It must evaluate to uint64.

This operation is only permitted in application mode.

class `pyteal.BinaryExpr` (*op: pyteal.Op, inputType: pyteal.TealType, outputType: pyteal.TealType, argLeft: pyteal.Expr, argRight: pyteal.Expr*)

Bases: `pyteal.Expr`

An expression with two arguments.

type_of ()

Get the return type of this expression.

`pyteal.Add` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Add two numbers.

Produces left + right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Minus` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Subtract two numbers.

Produces left - right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Mul` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Multiply two numbers.

Produces left * right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Div` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Divide two numbers.

Produces left / right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.BitwiseAnd` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Bitwise and expression.

Produces left & right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.BitwiseOr` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`

Bitwise or expression.

Produces left | right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.BitwiseXor(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Bitwise xor expression.

Produces $\text{left} \wedge \text{right}$.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Mod(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Modulo expression.

Produces $\text{left} \% \text{right}$.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Eq(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Equality expression.

Checks if $\text{left} == \text{right}$.

Parameters

- **left** – A value to check.
- **right** – The other value to check. Must evaluate to the same type as left.

`pyteal.Neq(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Difference expression.

Checks if $\text{left} != \text{right}$.

Parameters

- **left** – A value to check.
- **right** – The other value to check. Must evaluate to the same type as left.

`pyteal.Lt(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Less than expression.

Checks if $\text{left} < \text{right}$.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Le(left: pyteal.Expr, right: pyteal.Expr) → pyteal.BinaryExpr`

Less than or equal to expression.

Checks if $\text{left} \leq \text{right}$.

Parameters

- **left** – Must evaluate to uint64.

- **right** – Must evaluate to uint64.

`pyteal.Gt` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`
Greater than expression.

Checks if left > right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`pyteal.Ge` (*left: pyteal.Expr, right: pyteal.Expr*) → `pyteal.BinaryExpr`
Greater than or equal to expression.

Checks if left >= right.

Parameters

- **left** – Must evaluate to uint64.
- **right** – Must evaluate to uint64.

`class pyteal.Ed25519Verify` (*data: pyteal.Expr, sig: pyteal.Expr, key: pyteal.Expr*)
Bases: `pyteal.Expr`

An expression to verify ed25519 signatures.

`__init__` (*data: pyteal.Expr, sig: pyteal.Expr, key: pyteal.Expr*) → None
Verify the ed25519 signature of (“ProgData” || program_hash || data).

Parameters

- **data** – The data signed by the public. Must evaluate to bytes.
- **sig** – The proposed 64 byte signature of (“ProgData” || program_hash || data). Must evaluate to bytes.
- **key** – The 32 byte public key that produced the signature. Must evaluate to bytes.

`type_of` ()
Get the return type of this expression.

`class pyteal.Substring` (*string: pyteal.Expr, start: pyteal.Expr, end: pyteal.Expr*)
Bases: `pyteal.Expr`

Take a substring of a byte string.

`__init__` (*string: pyteal.Expr, start: pyteal.Expr, end: pyteal.Expr*) → None
Create a new Substring expression.

Produces a new byte string consisting of the bytes starting at start up to but not including end.

Parameters

- **string** – The byte string.
- **start** – The starting index for the substring.
- **end** – The ending index for the substring.

`type_of` ()
Get the return type of this expression.

`class pyteal.NaryExpr` (*op: pyteal.Op, inputType: pyteal.TealType, outputType: pyteal.TealType, args: Sequence[pyteal.Expr]*)
Bases: `pyteal.Expr`

N-ary expression base class.

This type of expression takes an arbitrary number of arguments.

type_of()

Get the return type of this expression.

`pyteal.And(*args) → pyteal.NaryExpr`

Logical and expression.

Produces 1 if all arguments are nonzero. Otherwise produces 0.

All arguments must be PyTeal expressions that evaluate to uint64, and there must be at least two arguments.

Example

```
And(Txn.amount() == Int(500), Txn.fee() <= Int(10))
```

`pyteal.Or(*args) → pyteal.NaryExpr`

Logical or expression.

Produces 1 if any argument is nonzero. Otherwise produces 0.

All arguments must be PyTeal expressions that evaluate to uint64, and there must be at least two arguments.

`pyteal.Concat(*args) → pyteal.NaryExpr`

Concatenate byte strings.

Produces a new byte string consisting of the contents of each of the passed in byte strings joined together.

All arguments must be PyTeal expressions that evaluate to bytes, and there must be at least two arguments.

Example

```
Concat(Bytes("hello"), Bytes(" "), Bytes("world"))
```

class `pyteal.If` (*cond: pyteal.Expr, thenBranch: pyteal.Expr, elseBranch: pyteal.Expr = None*)

Bases: `pyteal.Expr`

Simple two-way conditional expression.

__init__ (*cond: pyteal.Expr, thenBranch: pyteal.Expr, elseBranch: pyteal.Expr = None*) → None

Create a new If expression.

When this If expression is executed, the condition will be evaluated, and if it produces a true value, thenBranch will be executed and used as the return value for this expression. Otherwise, elseBranch will be executed and used as the return value, if it is provided.

Parameters

- **cond** – The condition to check. Must evaluate to uint64.
- **thenBranch** – Expression to evaluate if the condition is true.
- **elseBranch** (*optional*) – Expression to evaluate if the condition is false. Must evaluate to the same type as thenBranch, if provided. Defaults to None.

type_of()

Get the return type of this expression.

class `pyteal.Cond` (**argv*)

Bases: `pyteal.Expr`

A chainable branching expression that supports an arbitrary number of conditions.

`__init__ (*argv)`

Create a new Cond expression.

At least one argument must be provided, and each argument must be a list with two elements. The first element is a condition which evaluates to uint64, and the second is the body of the condition, which will execute if that condition is true. All condition bodies must have the same return type. During execution, each condition is tested in order, and the first condition to evaluate to a true value will cause its associated body to execute and become the value for this Cond expression. If no condition evaluates to a true value, the Cond expression produces an error and the TEAL program terminates.

Example

```
Cond([Global.group_size() == Int(5), bid],
     [Global.group_size() == Int(4), redeem],
     [Global.group_size() == Int(1), wrapup])
```

`type_of()`

Get the return type of this expression.

class `pyteal.Seq` (*exprs: List[pyteal.Expr]*)

Bases: `pyteal.Expr`

A control flow expression to represent a sequence of expressions.

`__init__` (*exprs: List[pyteal.Expr]*)

Create a new Seq expression.

The new Seq expression will take on the return value of the final expression in the sequence.

Parameters *exprs* – The expressions to include in this sequence. All expressions that are not the final one in this list must not return any values.

Example

```
Seq([
    App.localPut(Bytes("key"), Bytes("value")),
    Int(1)
])
```

`type_of()`

Get the return type of this expression.

class `pyteal.Assert` (*cond: pyteal.Expr*)

Bases: `pyteal.Expr`

A control flow expression to verify that a condition is true.

`__init__` (*cond: pyteal.Expr*) → None

Create an assert statement that raises an error if the condition is false.

Parameters *cond* – The condition to check. Must evaluate to a uint64.

`type_of()`

Get the return type of this expression.

class `pyteal.ScratchSlot`

Bases: `object`

Represents the allocation of a scratch space slot.

load (*type: pyteal.TealType = <TealType.anytype: 2>*)

Get an expression to load a value from this slot.

Parameters **type** (*optional*) – The type being loaded from this slot, if known. Defaults to `TealType.anytype`.

slotId = 0

store ()

Get an expression to store a value in this slot.

class `pyteal.ScratchLoad` (*slot: pyteal.ScratchSlot, type: pyteal.TealType = <TealType.anytype: 2>*)

Bases: `pyteal.Expr`

Expression to load a value from scratch space.

__init__ (*slot: pyteal.ScratchSlot, type: pyteal.TealType = <TealType.anytype: 2>*)

Create a new `ScratchLoad` expression.

Parameters

- **slot** – The slot to load the value from.
- **type** (*optional*) – The type being loaded from this slot, if known. Defaults to `TealType.anytype`.

type_of ()

Get the return type of this expression.

class `pyteal.ScratchStore` (*slot: pyteal.ScratchSlot*)

Bases: `pyteal.Expr`

Expression to store a value in scratch space.

__init__ (*slot: pyteal.ScratchSlot*)

Create a new `ScratchStore` expression.

Parameters **slot** – The slot to store the value in.

type_of ()

Get the return type of this expression.

class `pyteal.MaybeValue` (*op: pyteal.Op, type: pyteal.TealType, *, immediate_args: List[Union[int, str]] = None, args: List[pyteal.Expr] = None*)

Bases: `pyteal.LeafExpr`

Represents a get operation returning a value that may not exist.

__init__ (*op: pyteal.Op, type: pyteal.TealType, *, immediate_args: List[Union[int, str]] = None, args: List[pyteal.Expr] = None*)

Create a new `MaybeValue`.

Parameters

- **op** – The operation that returns values.
- **type** – The type of the returned value.
- **immediate_args** (*optional*) – Immediate arguments for the op. Defaults to `None`.
- **args** (*optional*) – Stack arguments for the op. Defaults to `None`.

hasValue () → `pyteal.ScratchLoad`

Check if the value exists.

This will return 1 if the value exists, otherwise 0.

type_of()

Get the return type of this expression.

value() → `pyteal.ScratchLoad`

Get the value.

If the value exists, it will be returned. Otherwise, the zero value for this type will be returned (i.e. either 0 or an empty byte string, depending on the type).

class `pyteal.Op`(*value: str, mode: pyteal.Mode*)

Bases: `enum.Enum`

Enum of program opcodes.

`add` = '+'

`addr` = 'addr'

`addw` = 'addw'

`app_global_del` = 'app_global_del'

`app_global_get` = 'app_global_get'

`app_global_get_ex` = 'app_global_get_ex'

`app_global_put` = 'app_global_put'

`app_local_del` = 'app_local_del'

`app_local_get` = 'app_local_get'

`app_local_get_ex` = 'app_local_get_ex'

`app_local_put` = 'app_local_put'

`app_opted_in` = 'app_opted_in'

`arg` = 'arg'

`asset_holding_get` = 'asset_holding_get'

`asset_params_get` = 'asset_params_get'

`b` = 'b'

`balance` = 'balance'

`bitwise_and` = '&'

`bitwise_not` = '~'

`bitwise_or` = '|'

`bitwise_xor` = '^'

`bnz` = 'bnz'

`btoi` = 'btoi'

`byte` = 'byte'

`bz` = 'bz'

`concat` = 'concat'

`div` = '/'

`dup` = 'dup'

```
dup2 = 'dup2'
ed25519verify = 'ed25519verify'
eq = '=='
err = 'err'
ge = '>='
global_ = 'global'
gt = '>'
gtxn = 'gtxn'
gtxna = 'gtxna'
int = 'int'
itob = 'itob'
keccak256 = 'keccak256'
le = '<='
len = 'len'
load = 'load'
logic_and = '&&'
logic_not = '!'
logic_or = '||'
lt = '<'
minus = '-'
mod = '%'
mul = '*'
mulw = 'mulw'
neq = '!='
pop = 'pop'
return_ = 'return'
sha256 = 'sha256'
sha512_256 = 'sha512_256'
store = 'store'
substring = 'substring'
substring3 = 'substring3'
txn = 'txn'
txna = 'txna'

class pyteal.Mode
    Bases: enum.Flag
    Enum of program running modes.
```

```
Application = 2
Signature = 1
class pyteal.TealComponent
    Bases: abc.ABC

    assemble() → str
    assignSlot(slot: ScratchSlot, location: int)
    getSlots() → List[ScratchSlot]
class pyteal.TealOp(op: pyteal.Op, *args)
    Bases: pyteal.TealComponent

    assemble() → str
    assignSlot(slot: ScratchSlot, location: int)
    getOp() → pyteal.Op
    getSlots() → List[ScratchSlot]
class pyteal.TealLabel(label: str)
    Bases: pyteal.TealComponent

    assemble() → str
pyteal.compileTeal(ast: pyteal.Expr, mode: pyteal.Mode) → str
    Compile a PyTeal expression into TEAL assembly.

    Parameters ast – The PyTeal expression to assemble.

    Returns A TEAL assembly program compiled from the input expression.

    Return type str
class pyteal.TealType
    Bases: enum.Enum

    Teal type enum.

    anytype = 2
    bytes = 1
    none = 3
    uint64 = 0
exception pyteal.TealInternalError(message: str)
    Bases: Exception
exception pyteal.TealTypeError(actual, expected)
    Bases: Exception
exception pyteal.TealInputError(msg)
    Bases: Exception
pyteal.execute(args)
    Execute in bash, return stdout and stderr in string

    Arguments: args: command and arguments to run, e.g. ['ls', '-l']
```

CHAPTER 12

Indices and tables

- `genindex`

p

pyteal, [51](#)

Symbols

[__getitem__\(\)](#) (*pyteal.Array* method), 64
[__getitem__\(\)](#) (*pyteal.TxnArray* method), 55
[__getitem__\(\)](#) (*pyteal.TxnGroup* method), 59
[__init__\(\)](#) (*pyteal.Addr* method), 51
[__init__\(\)](#) (*pyteal.Arg* method), 52
[__init__\(\)](#) (*pyteal.Assert* method), 70
[__init__\(\)](#) (*pyteal.Bytes* method), 52
[__init__\(\)](#) (*pyteal.Cond* method), 69
[__init__\(\)](#) (*pyteal.Ed25519Verify* method), 68
[__init__\(\)](#) (*pyteal.EnumInt* method), 52
[__init__\(\)](#) (*pyteal.If* method), 69
[__init__\(\)](#) (*pyteal.Int* method), 52
[__init__\(\)](#) (*pyteal.MaybeValue* method), 71
[__init__\(\)](#) (*pyteal.Nonce* method), 64
[__init__\(\)](#) (*pyteal.ScratchLoad* method), 71
[__init__\(\)](#) (*pyteal.ScratchStore* method), 71
[__init__\(\)](#) (*pyteal.Seq* method), 70
[__init__\(\)](#) (*pyteal.Substring* method), 68

A

[accounts](#) (*pyteal.TxnField* attribute), 53
[accounts](#) (*pyteal.TxnObject* attribute), 55
[add](#) (*pyteal.Op* attribute), 72
[Add\(\)](#) (in module *pyteal*), 66
[Addr](#) (class in *pyteal*), 51
[addr](#) (*pyteal.Op* attribute), 72
[Addr\(\)](#) (*pyteal.Tmpl* class method), 64
[addw](#) (*pyteal.Op* attribute), 72
[amount](#) (*pyteal.TxnField* attribute), 53
[amount\(\)](#) (*pyteal.TxnObject* method), 55
[And\(\)](#) (in module *pyteal*), 69
[And\(\)](#) (*pyteal.Expr* method), 51
[anytype](#) (*pyteal.TealType* attribute), 74
[App](#) (class in *pyteal*), 60
[app_global_del](#) (*pyteal.Op* attribute), 72
[app_global_get](#) (*pyteal.Op* attribute), 72
[app_global_get_ex](#) (*pyteal.Op* attribute), 72
[app_global_put](#) (*pyteal.Op* attribute), 72

[app_local_del](#) (*pyteal.Op* attribute), 72
[app_local_get](#) (*pyteal.Op* attribute), 72
[app_local_get_ex](#) (*pyteal.Op* attribute), 72
[app_local_put](#) (*pyteal.Op* attribute), 72
[app_opted_in](#) (*pyteal.Op* attribute), 72
[AppField](#) (class in *pyteal*), 62
[Application](#) (*pyteal.Mode* attribute), 73
[application_args](#) (*pyteal.TxnField* attribute), 53
[application_args](#) (*pyteal.TxnObject* attribute), 55
[application_id](#) (*pyteal.TxnField* attribute), 53
[application_id\(\)](#) (*pyteal.TxnObject* method), 55
[ApplicationCall](#) (*pyteal.TxnType* attribute), 53
[approval_program](#) (*pyteal.TxnField* attribute), 53
[approval_program\(\)](#) (*pyteal.TxnObject* method), 55
[Arg](#) (class in *pyteal*), 52
[arg](#) (*pyteal.Op* attribute), 72
[Array](#) (class in *pyteal*), 64
[assemble\(\)](#) (*pyteal.TealComponent* method), 74
[assemble\(\)](#) (*pyteal.TealLabel* method), 74
[assemble\(\)](#) (*pyteal.TealOp* method), 74
[Assert](#) (class in *pyteal*), 70
[asset_amount](#) (*pyteal.TxnField* attribute), 53
[asset_amount\(\)](#) (*pyteal.TxnObject* method), 55
[asset_close_to](#) (*pyteal.TxnField* attribute), 53
[asset_close_to\(\)](#) (*pyteal.TxnObject* method), 55
[asset_holding_get](#) (*pyteal.Op* attribute), 72
[asset_params_get](#) (*pyteal.Op* attribute), 72
[asset_receiver](#) (*pyteal.TxnField* attribute), 53
[asset_receiver\(\)](#) (*pyteal.TxnObject* method), 55
[asset_sender](#) (*pyteal.TxnField* attribute), 53
[asset_sender\(\)](#) (*pyteal.TxnObject* method), 56
[AssetConfig](#) (*pyteal.TxnType* attribute), 53
[AssetFreeze](#) (*pyteal.TxnType* attribute), 53
[AssetHolding](#) (class in *pyteal*), 62
[AssetParam](#) (class in *pyteal*), 63
[AssetTransfer](#) (*pyteal.TxnType* attribute), 53
[assignSlot\(\)](#) (*pyteal.TealComponent* method), 74
[assignSlot\(\)](#) (*pyteal.TealOp* method), 74

B

b (*pyteal.Op attribute*), 72
balance (*pyteal.Op attribute*), 72
Balance() (*in module pyteal*), 65
balance() (*pyteal.AssetHolding class method*), 62
BinaryExpr (*class in pyteal*), 65
bitwise_and (*pyteal.Op attribute*), 72
bitwise_not (*pyteal.Op attribute*), 72
bitwise_or (*pyteal.Op attribute*), 72
bitwise_xor (*pyteal.Op attribute*), 72
BitwiseAnd() (*in module pyteal*), 66
BitwiseNot() (*in module pyteal*), 65
BitwiseOr() (*in module pyteal*), 66
BitwiseXor() (*in module pyteal*), 67
bnz (*pyteal.Op attribute*), 72
btoi (*pyteal.Op attribute*), 72
Btoi() (*in module pyteal*), 65
byte (*pyteal.Op attribute*), 72
Bytes (*class in pyteal*), 52
bytes (*pyteal.TealType attribute*), 74
Bytes() (*pyteal.Tmpl class method*), 64
bz (*pyteal.Op attribute*), 72

C

clawback() (*pyteal.AssetParam class method*), 63
clear_state_program (*pyteal.TxnField attribute*), 53
clear_state_program() (*pyteal.TxnObject method*), 56
ClearState (*pyteal.OnComplete attribute*), 62
close_remainder_to (*pyteal.TxnField attribute*), 53
close_remainder_to() (*pyteal.TxnObject method*), 56
CloseOut (*pyteal.OnComplete attribute*), 62
compileTeal() (*in module pyteal*), 74
concat (*pyteal.Op attribute*), 72
Concat() (*in module pyteal*), 69
Cond (*class in pyteal*), 69
config_asset (*pyteal.TxnField attribute*), 53
config_asset() (*pyteal.TxnObject method*), 56
config_asset_clawback (*pyteal.TxnField attribute*), 53
config_asset_clawback() (*pyteal.TxnObject method*), 56
config_asset_decimals (*pyteal.TxnField attribute*), 53
config_asset_decimals() (*pyteal.TxnObject method*), 56
config_asset_default_frozen (*pyteal.TxnField attribute*), 53
config_asset_default_frozen() (*pyteal.TxnObject method*), 56
config_asset_freeze (*pyteal.TxnField attribute*), 53

config_asset_freeze() (*pyteal.TxnObject method*), 56
config_asset_manager (*pyteal.TxnField attribute*), 53
config_asset_manager() (*pyteal.TxnObject method*), 56
config_asset_metadata_hash (*pyteal.TxnField attribute*), 53
config_asset_metadata_hash() (*pyteal.TxnObject method*), 56
config_asset_name (*pyteal.TxnField attribute*), 54
config_asset_name() (*pyteal.TxnObject method*), 57
config_asset_reserve (*pyteal.TxnField attribute*), 54
config_asset_reserve() (*pyteal.TxnObject method*), 57
config_asset_total (*pyteal.TxnField attribute*), 54
config_asset_total() (*pyteal.TxnObject method*), 57
config_asset_unit_name (*pyteal.TxnField attribute*), 54
config_asset_unit_name() (*pyteal.TxnObject method*), 57
config_asset_url (*pyteal.TxnField attribute*), 54
config_asset_url() (*pyteal.TxnObject method*), 57
current_app_id (*pyteal.GlobalField attribute*), 60
current_application_id() (*pyteal.Global class method*), 59

D

decimals() (*pyteal.AssetParam class method*), 63
defaultFrozen() (*pyteal.AssetParam class method*), 63
DeleteApplication (*pyteal.OnComplete attribute*), 62
div (*pyteal.Op attribute*), 72
Div() (*in module pyteal*), 66
dup (*pyteal.Op attribute*), 72
dup2 (*pyteal.Op attribute*), 72

E

Ed25519Verify (*class in pyteal*), 68
ed25519verify (*pyteal.Op attribute*), 73
EnumInt (*class in pyteal*), 52
eq (*pyteal.Op attribute*), 73
Eq() (*in module pyteal*), 67
Err (*class in pyteal*), 52
err (*pyteal.Op attribute*), 73
execute() (*in module pyteal*), 74
Expr (*class in pyteal*), 51

F

fee (*pyteal.TxnField* attribute), 54
fee() (*pyteal.TxnObject* method), 57
first_valid (*pyteal.TxnField* attribute), 54
first_valid() (*pyteal.TxnObject* method), 57
first_valid_time (*pyteal.TxnField* attribute), 54
freeze() (*pyteal.AssetParam* class method), 63
freeze_asset (*pyteal.TxnField* attribute), 54
freeze_asset() (*pyteal.TxnObject* method), 57
freeze_asset_account (*pyteal.TxnField* attribute), 54
freeze_asset_account() (*pyteal.TxnObject* method), 57
freeze_asset_frozen (*pyteal.TxnField* attribute), 54
freeze_asset_frozen() (*pyteal.TxnObject* method), 57
frozen() (*pyteal.AssetHolding* class method), 63

G

ge (*pyteal.Op* attribute), 73
Ge() (in module *pyteal*), 68
get_op() (*pyteal.AppField* method), 62
getOp() (*pyteal.TealOp* method), 74
getSlots() (*pyteal.TealComponent* method), 74
getSlots() (*pyteal.TealOp* method), 74
Global (class in *pyteal*), 59
global_ (*pyteal.Op* attribute), 73
globalDel (*pyteal.AppField* attribute), 62
globalDel() (*pyteal.App* class method), 60
GlobalField (class in *pyteal*), 60
globalGet (*pyteal.AppField* attribute), 62
globalGet() (*pyteal.App* class method), 60
globalGetEx (*pyteal.AppField* attribute), 62
globalGetEx() (*pyteal.App* class method), 61
globalPut (*pyteal.AppField* attribute), 62
globalPut() (*pyteal.App* class method), 61
group_index (*pyteal.TxnField* attribute), 54
group_index() (*pyteal.TxnObject* method), 58
group_size (*pyteal.GlobalField* attribute), 60
group_size() (*pyteal.Global* class method), 59
gt (*pyteal.Op* attribute), 73
Gt() (in module *pyteal*), 68
Gtxn (in module *pyteal*), 51
gtxn (*pyteal.Op* attribute), 73
gtxna (*pyteal.Op* attribute), 73
GtxnaExpr (class in *pyteal*), 59
GtxnExpr (class in *pyteal*), 59

H

hasValue() (*pyteal.MaybeValue* method), 71

I

id() (*pyteal.App* class method), 61

If (class in *pyteal*), 69
Int (class in *pyteal*), 52
int (*pyteal.Op* attribute), 73
Int() (*pyteal.Tmpl* class method), 64
itob (*pyteal.Op* attribute), 73
Itob() (in module *pyteal*), 65

K

keccak256 (*pyteal.Op* attribute), 73
Keccak256() (in module *pyteal*), 65
KeyRegistration (*pyteal.TxnType* attribute), 53

L

last_valid (*pyteal.TxnField* attribute), 54
last_valid() (*pyteal.TxnObject* method), 58
latest_timestamp (*pyteal.GlobalField* attribute), 60
latest_timestamp() (*pyteal.Global* class method), 60
le (*pyteal.Op* attribute), 73
Le() (in module *pyteal*), 67
LeafExpr (class in *pyteal*), 51
lease (*pyteal.TxnField* attribute), 54
lease() (*pyteal.TxnObject* method), 58
len (*pyteal.Op* attribute), 73
Len() (in module *pyteal*), 65
length() (*pyteal.Array* method), 64
length() (*pyteal.TxnArray* method), 55
load (*pyteal.Op* attribute), 73
load() (*pyteal.ScratchSlot* method), 70
localDel (*pyteal.AppField* attribute), 62
localDel() (*pyteal.App* class method), 61
localGet (*pyteal.AppField* attribute), 62
localGet() (*pyteal.App* class method), 61
localGetEx (*pyteal.AppField* attribute), 62
localGetEx() (*pyteal.App* class method), 61
localPut (*pyteal.AppField* attribute), 62
localPut() (*pyteal.App* class method), 61
logic_and (*pyteal.Op* attribute), 73
logic_not (*pyteal.Op* attribute), 73
logic_or (*pyteal.Op* attribute), 73
logic_sig_version (*pyteal.GlobalField* attribute), 60
logic_sig_version() (*pyteal.Global* class method), 60
lt (*pyteal.Op* attribute), 73
Lt() (in module *pyteal*), 67

M

manager() (*pyteal.AssetParam* class method), 63
max_txn_life (*pyteal.GlobalField* attribute), 60
max_txn_life() (*pyteal.Global* class method), 60
MaybeValue (class in *pyteal*), 71

metadataHash() (pyteal.AssetParam class method), 63

min_balance (pyteal.GlobalField attribute), 60
min_balance() (pyteal.Global class method), 60
min_txn_fee (pyteal.GlobalField attribute), 60
min_txn_fee() (pyteal.Global class method), 60
minus (pyteal.Op attribute), 73
Minus() (in module pyteal), 66
mod (pyteal.Op attribute), 73
Mod() (in module pyteal), 67
Mode (class in pyteal), 73
mul (pyteal.Op attribute), 73
Mul() (in module pyteal), 66
mulw (pyteal.Op attribute), 73

N

name() (pyteal.AssetParam class method), 63
NaryExpr (class in pyteal), 68
neq (pyteal.Op attribute), 73
Neq() (in module pyteal), 67
Nonce (class in pyteal), 64
none (pyteal.TealType attribute), 74
NoOp (pyteal.OnComplete attribute), 62
Not() (in module pyteal), 65
note (pyteal.TxnField attribute), 54
note() (pyteal.TxnObject method), 58
num_accounts (pyteal.TxnField attribute), 54
num_app_args (pyteal.TxnField attribute), 54

O

on_completion (pyteal.TxnField attribute), 54
on_completion() (pyteal.TxnObject method), 58
OnComplete (class in pyteal), 62
Op (class in pyteal), 72
optedIn (pyteal.AppField attribute), 62
optedIn() (pyteal.App class method), 62
OptIn (pyteal.OnComplete attribute), 62
Or() (in module pyteal), 69
Or() (pyteal.Expr method), 51

P

Payment (pyteal.TxnType attribute), 53
pop (pyteal.Op attribute), 73
Pop() (in module pyteal), 65
pyteal (module), 51

R

receiver (pyteal.TxnField attribute), 54
receiver() (pyteal.TxnObject method), 58
rekey_to (pyteal.TxnField attribute), 54
rekey_to() (pyteal.TxnObject method), 58
reserve() (pyteal.AssetParam class method), 63
Return() (in module pyteal), 65

return_ (pyteal.Op attribute), 73
RFC
 RFC 4648#section-4, 30
 RFC 4648#section-6, 30
 RFC 4648#section-8, 29
round (pyteal.GlobalField attribute), 60
round() (pyteal.Global class method), 60

S

ScratchLoad (class in pyteal), 71
ScratchSlot (class in pyteal), 70
ScratchStore (class in pyteal), 71
selection_pk (pyteal.TxnField attribute), 54
selection_pk() (pyteal.TxnObject method), 58
sender (pyteal.TxnField attribute), 54
sender() (pyteal.TxnObject method), 58
Seq (class in pyteal), 70
sha256 (pyteal.Op attribute), 73
Sha256() (in module pyteal), 65
sha512_256 (pyteal.Op attribute), 73
Sha512_256() (in module pyteal), 65
Signature (pyteal.Mode attribute), 74
slotId (pyteal.ScratchSlot attribute), 71
store (pyteal.Op attribute), 73
store() (pyteal.ScratchSlot method), 71
Substring (class in pyteal), 68
substring (pyteal.Op attribute), 73
substring3 (pyteal.Op attribute), 73

T

TealComponent (class in pyteal), 74
TealInputError, 74
TealInternalError, 74
TealLabel (class in pyteal), 74
TealOp (class in pyteal), 74
TealType (class in pyteal), 74
TealTypeError, 74
Tpl (class in pyteal), 64
total() (pyteal.AssetParam class method), 64
tx_id (pyteal.TxnField attribute), 54
tx_id() (pyteal.TxnObject method), 58
Txn (in module pyteal), 51
txn (pyteal.Op attribute), 73
txna (pyteal.Op attribute), 73
TxnaExpr (class in pyteal), 54
TxnArray (class in pyteal), 55
TxnExpr (class in pyteal), 54
TxnField (class in pyteal), 53
TxnGroup (class in pyteal), 59
TxnObject (class in pyteal), 55
TxnType (class in pyteal), 53
type (pyteal.TxnField attribute), 54
type() (pyteal.TxnObject method), 58
type_enum (pyteal.TxnField attribute), 54

[type_enum\(\)](#) (*pyteal.TxnObject method*), 58
[type_of\(\)](#) (*pyteal.Addr method*), 52
[type_of\(\)](#) (*pyteal.App method*), 62
[type_of\(\)](#) (*pyteal.AppField method*), 62
[type_of\(\)](#) (*pyteal.Arg method*), 53
[type_of\(\)](#) (*pyteal.Assert method*), 70
[type_of\(\)](#) (*pyteal.BinaryExpr method*), 66
[type_of\(\)](#) (*pyteal.Bytes method*), 52
[type_of\(\)](#) (*pyteal.Cond method*), 70
[type_of\(\)](#) (*pyteal.Ed25519Verify method*), 68
[type_of\(\)](#) (*pyteal.EnumInt method*), 52
[type_of\(\)](#) (*pyteal.Err method*), 52
[type_of\(\)](#) (*pyteal.Expr method*), 51
[type_of\(\)](#) (*pyteal.Global method*), 60
[type_of\(\)](#) (*pyteal.GlobalField method*), 60
[type_of\(\)](#) (*pyteal.If method*), 69
[type_of\(\)](#) (*pyteal.Int method*), 52
[type_of\(\)](#) (*pyteal.MaybeValue method*), 71
[type_of\(\)](#) (*pyteal.NaryExpr method*), 69
[type_of\(\)](#) (*pyteal.Nonce method*), 65
[type_of\(\)](#) (*pyteal.ScratchLoad method*), 71
[type_of\(\)](#) (*pyteal.ScratchStore method*), 71
[type_of\(\)](#) (*pyteal.Seq method*), 70
[type_of\(\)](#) (*pyteal.Substring method*), 68
[type_of\(\)](#) (*pyteal.Tmpl method*), 64
[type_of\(\)](#) (*pyteal.TxnaExpr method*), 55
[type_of\(\)](#) (*pyteal.TxnExpr method*), 54
[type_of\(\)](#) (*pyteal.TxnField method*), 54
[type_of\(\)](#) (*pyteal.UnaryExpr method*), 65

U

[uint64](#) (*pyteal.TealType attribute*), 74
[UnaryExpr](#) (*class in pyteal*), 65
[unitName\(\)](#) (*pyteal.AssetParam class method*), 64
[Unknown](#) (*pyteal.TxnType attribute*), 53
[UpdateApplication](#) (*pyteal.OnComplete attribute*), 62
[url\(\)](#) (*pyteal.AssetParam class method*), 64

V

[value\(\)](#) (*pyteal.MaybeValue method*), 72
[vote_first](#) (*pyteal.TxnField attribute*), 54
[vote_first\(\)](#) (*pyteal.TxnObject method*), 59
[vote_key_dilution](#) (*pyteal.TxnField attribute*), 54
[vote_key_dilution\(\)](#) (*pyteal.TxnObject method*), 59
[vote_last](#) (*pyteal.TxnField attribute*), 54
[vote_last\(\)](#) (*pyteal.TxnObject method*), 59
[vote_pk](#) (*pyteal.TxnField attribute*), 54
[vote_pk\(\)](#) (*pyteal.TxnObject method*), 59

X

[xfer_asset](#) (*pyteal.TxnField attribute*), 54
[xfer_asset\(\)](#) (*pyteal.TxnObject method*), 59

Z

[zero_address](#) (*pyteal.GlobalField attribute*), 60
[zero_address\(\)](#) (*pyteal.Global class method*), 60