Getting started with PyTeal

Lecture 3

Plan

1. Data types in PyTeal
2. Arithmetic operations
3. Transaction fields and global parameters
4. Scratch space
5. Control flow expressions

AVM data types in PyTeal

There are two main types of data:

- Integers
- Bytes

- Integers are actually uint64, which means we can only use positive integers up to 2^64. TealType.uint64, a 64-bit unsigned integer.

  Int(n) creates the constant TealType.uint64, where n >= 0 and n < 2 ^ 64.

- A byte or byte slice is a binary string. TealType.bytes, byte slice

Source: https://pyteal.readthedocs.io/en/stable/data_type.html

Bytes

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

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

  Example: `Bytes("hello world")`

- **Base16:** Byte slices can be created from a binary string in the base16 encoding of RFC 4648#section-8, for example. "0xA21212EF" or "A21212EF".
  Example: `Bytes("base16", "0xA21212EF")` or `Bytes("base16", "A21212EF") # "0x" is optional`

- **Base32:** Byte slices can be created from a RFC 4648#section-6 base32 encoded binary string with or without padding, e.g. "7Z5PWO2C6LFNQFGHWKSK5H47IQP5OJW2M3HA2QPXTY3WTNP5NU2MHBW27M".

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

Source: https://pyteal.readthedocs.io/en/stable/data_type.html

Transformation

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

- `Itob(n)`: generates a Tealtype.bytes value from a value n of type Tealtype.uint64
- `Btoi(b)`: generates a Tealtype.uint64 value from a value b of type Tealtype.bytes

These operations are not intended to convert between human-readable strings and numbers. Itob creates an 8-byte unsigned integer encoding, not a human-readable string. For example, Itob(Int(1)) creates the string "x00x00x00x00x00x00x00x01", not the string "1".

Source: https://pyteal.readthedocs.io/en/stable/data_type.html

Arithmetic operations

An arithmetic expression is an expression that results in a TealType.uint64 value. In PyTeal, arithmetic expressions include integer or Boolean operators (Boolean values are integers 0 or 1).

Checking types

PyTeal has a type-checking subsystem that rejects incorrectly typed PyTeal programs when creating a PyTeal program.

For example,

```
cond = Txn.fee() < Txn.receiver()
```

Running this PyTeal program in Python produces an error message:

```
TealTypeError: Type error: TealType.bytes while expected TealType.uint64
```

This is due to the fact that Txn.receiver() is of type TealType.bytes, while the overloaded < expects operands of type uint64 on both sides.

7

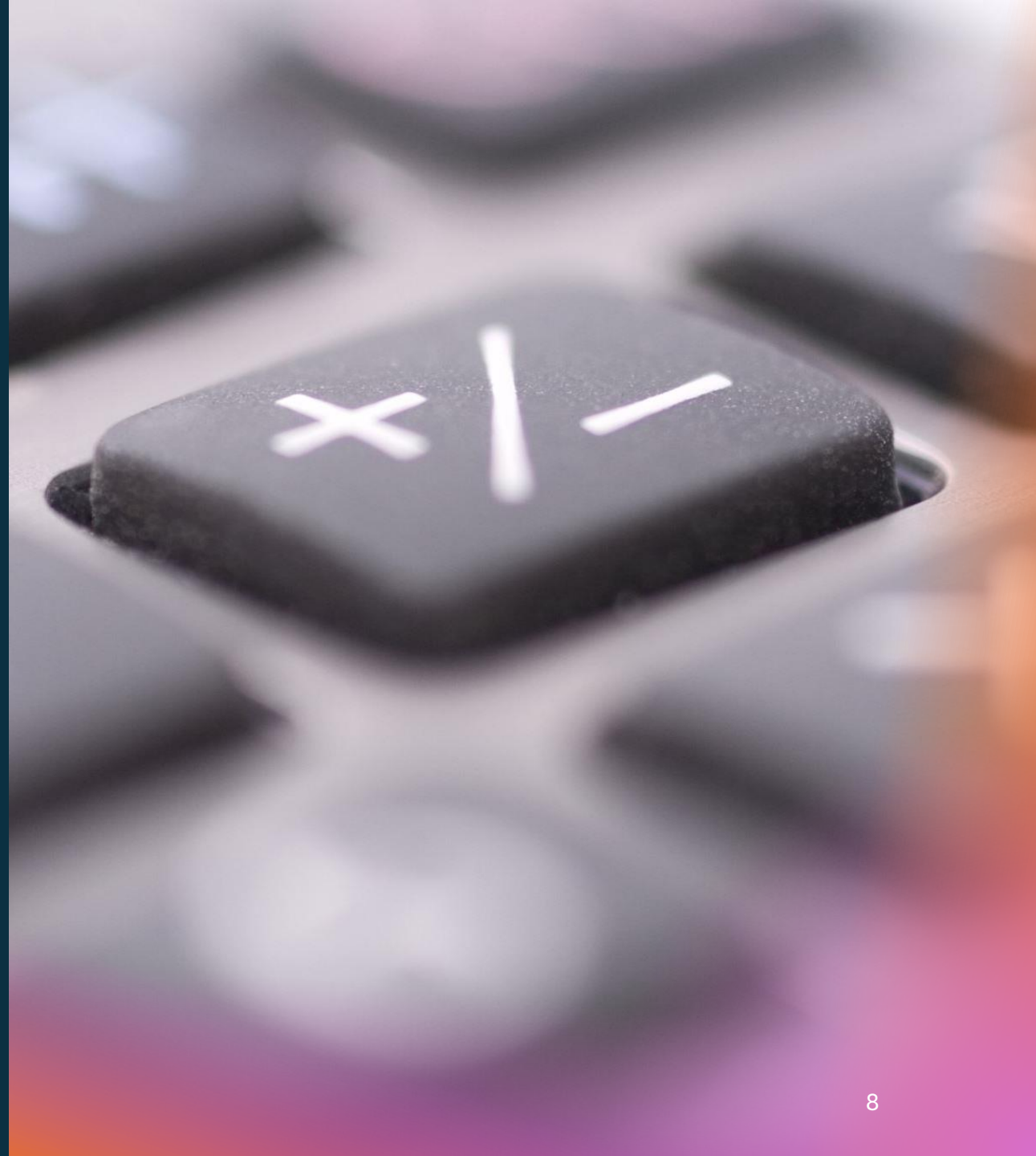
Operator overload

PyTeal overloads Python's arithmetic and comparison operators (+, -, *, /, >, <, >=, >=, ==) so that PyTeal users can express smart contract logic in Python more naturally.

Examples:

`Txn.fee() < Int(4000) # checks` that the current transaction fee is lower than 4000 microAlgo.

`Gtxn.amount(0) == Gtxn.amount(1) * Int(2) # checks` that the amount of Algo transferred in the first transaction of the group transaction is twice the amount of Algo transferred in the second transaction.



Transaction fields and global parameters

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

Transaction fields. General fields

Information about the current transaction being evaluated can be obtained using the **Txn** object using the PyTeal expressions shown below.

Operator.	Type	Notes.
Txn.type()	TealType.bytes	Determines the type of transaction. If the transaction is a payment, "pay" will be returned.
Txn.type_enum()	TealType.uint64	Transaction type as an integer. The value corresponds to a specific type of transaction (for example, 1 for a payment).
Txn.sender()	TealType.bytes	Transaction sender address as a 32-byte Algorand address
Txn.fee()	TealType.uint64	Transaction fees in microAlgos
Txn.first_valid()	TealType.uint64	The first round in which a transaction can be valid
Txn.first_valid_time()	TealType.uint64	UNIX timestamp of the block before Txn.first_valid(). Error if negative
Txn.last_valid()	TealType.uint64	
Txn.note()	TealType.bytes	Transaction note in bytes
Txn.group_index()	TealType.uint64	Position of this transaction in the group of transactions starting from 0
Txn.tx_id()	TealType.bytes	The calculated ID for this transaction. This is a 32-byte transaction identifier.

Source: https://pyteal.readthedocs.io/en/latest/accessing_transaction_field.html

10

Example of use

Txn.sender()

```
from pyteal import *

program = Seq([
    Assert(Txn.sender() == Addr("YOUR_ADDRESS_HERE")),
    Approve()
])
```

Txn.group_index()

```
from pyteal import *

program = Seq([
    Assert(Txn.group_index() == Int(0)),
    Approve()
])
```

Txn.note()

```
from pyteal import *

program = Seq([
    Assert(Txn.note() == Bytes("example note")),
    Approve()
])
```

Txn.fee()

```
from pyteal import *

program = Seq([
    Assert(Txn.fee() < Int(1000)),
    Approve()
])
```

Transaction fields. Calling the application

Operator.	Type	Notes.
Txn.application_id()	TealType.uint64	ID of the called application
Txn.on_completion()	TealType.uint64	The program transaction must specify the action to be performed after its approvalProgram or clearStateProgram is executed. OnComplete
Txn.approval_program()	TealType.bytes	Returns a byte string with the TEAL approval for creating or updating an application
Txn.global_num_uints()	TealType.uint64	The maximum number of global variables of type uint for the application
Txn.global_num_byte_slices()	TealType.uint64	The maximum number of global variables of the byte slice type for an application
Txn.local_num_uints()	TealType.uint64	Number of local variables of type uint for the application
Txn.local_num_byte_slices()	TealType.uint64	Number of local variables of the byte slice type for the application
Txn.accounts	TealType.bytes[]	Array of accounts available to the application
Txn.assets	TealType.uint64[]	Array of assets available to the application
Txn.applications	TealType.uint64[]	Array of applications
Txn.clear_state_program()	TealType.bytes	Returns a byte string with the TEAL program to clear the application state
Txn.application_args	TealType.bytes[]	Array of application arguments

Source: https://pyteal.readthedocs.io/en/latest/accessing_transaction_field.html

Example of use

Checking that the number of global variables of type Integer is 5

```
from pyteal import *

program = Seq([
    Assert(Txn.global_num_uints() == Int(5)),
    Approve()
])
```

Logging of the state clear programme

```
from pyteal import *

program = Seq([
    Log(Txn.clear_state_program()),
    Approve()
])
```

Checks that the first argument of the application matches "arg1"

```
from pyteal import *

program = Seq([
    Assert(Txn.application_args[0] == Bytes("arg1")),
    Approve()
])
```

Transaction fields. Asset transfer

Operator.	Type	Notes.
Txn.xfer_asset()	TealType.uint64	Used to get the ID of an asset that is transferred in an Asset Transfer transaction. This operator helps to identify the asset being transferred.
Txn.asset_amount()	TealType.uint64	The quantity of the asset to be transferred in the Asset Transfer transaction. The value represents the number of units of the asset as an integer (TealType.uint64).
Txn.asset_sender()	TealType.bytes	The address of the account initiating the asset transfer.
Txn.asset_receiver()	TealType.bytes	The address of the account that receives the asset in an Asset Transfer transaction. This field allows you to specify to whom the asset is transferred.
Txn.asset_close_to()	TealType.bytes	Select this field to remove assets from the sender's account and reduce the minimum account balance (i.e., to relinquish the asset).

Other transaction fields when working with assets:
Asset configuration: used to manage the creation and configuration of assets on the Algorand blockchain.
Asset freezes: used to manage the status of asset freezes on the Algorand blockchain.

Source: https://pyteal.readthedocs.io/en/latest/accessing_transaction_field.html

Types of transactions

The value of Txn.type_enum() can be checked using the TxnType enumeration:

Meaning.	Numerical value	Type string	Description.
TxnType.Unknown	0	unknown	unknown type, invalid
TxnType.Payment	1	pay	payment transaction
TxnType.KeyRegistration	2	keyreg	
TxnType.AssetConfig	3	acfg	
TxnType.AssetTransfer	4	axfer	asset transfer transaction
TxnType.AssetFreeze	5	afrz	asset freeze transaction
TxnType.ApplicationCall	6	appl	Application call transaction

Example:
Txn.type_enum() == Int(1)

Scratch space

Scratch Space in PyTeal is **a temporary place** to store values for later use in the application. This storage location is temporary, as any changes to it **are not saved after** the current transaction **is completed**.

PyTeal's Scratch Space allows you to store intermediate values while executing smart contracts.

The Scratch Space consists of **256 slots**, each of which can store **one value of the Int or Byte slices type**. When you use the `ScratchVar` class to work with Scratch Space, a slot is automatically assigned to each variable.

ScratchVar: Writing and Reading from Scratch Space

1. **Create a ScratchVar object:** create a ScratchVar object. Pass the type of values to be stored. This can be `TealType.uint64` for integers or `TealType.bytes` for byte arrays.

   ```
   myVar = ScratchVar(TealType.uint64)
   ```

2. **Storing a value in a ScratchVar:** the `store` method is used to write a value. It accepts the value to be stored in the `ScratchVar`.

   ```
   myVar.store(Int(5))
   ```

3. **Reading a value from a ScratchVar:** the `load` method is used to read a value. It returns the value stored in the `ScratchVar`.

It is also possible to manually specify which slot the ScratchVar should be assigned to in the TEAL code. If the slot ID is not specified, the compiler will assign it to any available slot.

An example of using ScratchVar

```
myvar = ScratchVar(TealType.uint64) # assign a scratch slot in any available slot
programme = Seq([
    myvar.store(Int(5)),
    Assert(myvar.load() == Int(5))
])
anotherVar = ScratchVar(TealType.bytes, 4) # assign this scratch slot to slot #4
```

In this example, we create two ScratchVar variables: one to store an integer, the other to store a byte array, and the second variable is assigned to a specific slot (slot #4).

A PyTeal program is a PyTeal expression that consists of other PyTeal expressions.

You cannot include your own Python expressions in the PyTeal expression tree.

```
def approval_program():
        program = Return(Int(1))
        return compileTeal(programme, Mode.Application, version=5)
```

In the example above, the entire program is Return(Int(1)). Here, Return is a PyTeal expression that takes another PyTeal expression as an argument. The argument we pass to it is the PyTeal expression Int(1).

For example, if instead of passing Int(1), we create a Return(1) program, PyTeal will generate an error saying that 1 is not a valid PyTeal expression.

*If you ever see any Python error, such as `...has no 'type_of' attribute`, PyTeal is probably trying to tell you that you have included something that is not a valid PyTeal expression.

Source: https://developer.algorand.org/docs/get-details/dapps/writing-contracts/pyteal/

Control flow expressions

Exit the application: Approve and Reject

The *Approve* and *Reject* expressions result in an immediate exit from the programme. If Approve is used, the execution is marked as successful, and if Reject is used, the execution is marked as unsuccessful.

TEAL Version 4+	Equivalent expression
Approve()	Return(Int(1))
Reject()	Return(Int(0))

These expressions also work inside subroutines. When they are used inside subroutines, they also lead to an immediate exit from the program, unlike Return(...), which simply returns from the subroutine.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Combining expressions into a chain: Seq

You can use the Seq expression to create a sequence of multiple expressions. Arguments are expressions that you want to include in the sequence, either as a variable number of arguments or as a single list.

Example:

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

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Combining expressions into a chain: Seq

Features of Seq:

- The **Seq** expression takes the value of the last expression in the sequence.
- All expressions in **Seq**, except the last one, must not return any values (e.g., must evaluate to TealType.none). This restriction exists because intermediate values should not add anything to the TEAL stack.

The invalid expression Seq:

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

This expression is incorrect because Txn.sender() returns the address of the sender of the transaction and adds this value to the TEAL stack, but this value is not used before Return(Int(1)) is executed. As a result, an unused value is left on the TEAL stack, which causes an error.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Combining expressions into a chain: Seq

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

The correct sequence looks like this:

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

*Pop is used to remove a value from the stack

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Simple branching: If

In PyTeal, conditional statements are implemented through **If** statements. These expressions allow you to execute different parts of the code depending on the result of a logical test.

The `If` expression has the following format:

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

Here:

- `test-expr` is always evaluated and must be of type `TealType.uint64`.
- If the result of `test-expr` is greater than `0`, `then-expr` is executed.
- If the result of `test-expr` is `0`, then `else-expr` is executed.

Note that `then-expr` and `else-expr` must be of the same type (for example, both must be `TealType.uint64`).

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Simple branching: If

There is an alternative way to write an **If** statement that makes it easier to read complex conditional statements:

```
If(test-expr)
    .Then(then-expr)
    .ElseIf(test-expr)
    .Then(then-expr)
    .Else(else-expr)
```

This format makes it easy to read and understand complex conditional logic by breaking it down into multiple lines. It also provides the ability to add multiple **ElseIf** conditions, which makes the code more organised and readable.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Checking conditions in PyTeal: Assert

Assert is used to ensure that certain conditions are met before the program continues.

Syntax:

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

How does Assert work?

- test-expr is always evaluated.
- test-expr must be of type TealType.uint64.
- If the test-expr result is greater than 0, the program continues to work.
- If the test-expr result is 0, the program exits with an error.
- In the case of more than one condition, use And().

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Checking conditions in PyTeal: Assert

Example:

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

In this example, the application will immediately exit with an error if the transaction type is not a payment.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Chain of tests in PyTeal: Cond

Cond is a PyTeal expression that creates a chain of tests to select a result expression.

Syntax:

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

How does Cond work?

- Each `test-expr` is evaluated in order, and the corresponding body is executed for a test that returns a valid value.
- If `test-expr` returns 0, the corresponding `body` is ignored and the next `test-expr` is evaluated.
- As soon as `test-expr` returns a true value (> 0), the corresponding `body` is evaluated to get the value of this **Cond** expression.
- If no `test-expr` returns a true value, the **Cond** expression is evaluated to `err` (TEAL opcode), which causes the program to crash.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Chain of tests in PyTeal: Cond

Data types in Cond

- Each `test-expr` must be of type `TealType.uint64`.
- `body` can be of type `Tealtype.uint64` or `Tealtype.bytes`.
- All `bodies` must have the same data type. Otherwise, the `TealTypeError` error is thrown.

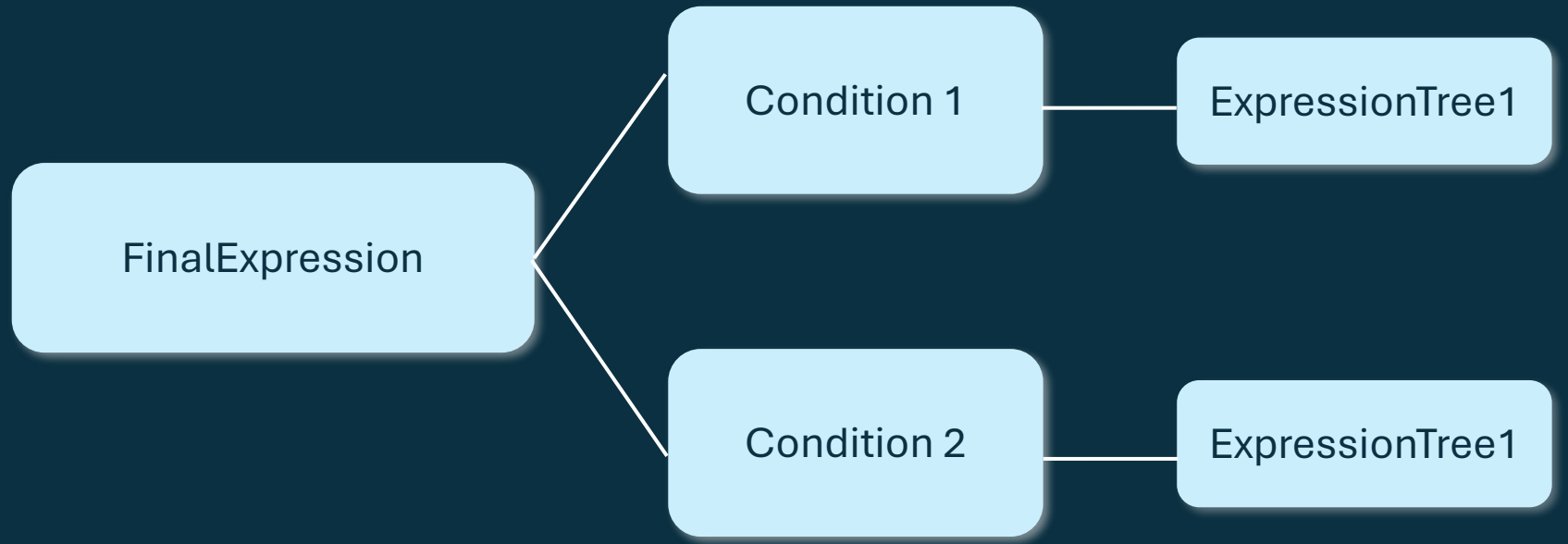
An example of using Cond:

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

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

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

PyTeal expression tree control flow




```
return Cond(
    [Int(1) == Int(0), tree1],
    [Int(1) == Int(1), tree2],
)
```

Cycles in PyTeal: While

While allows you to create simple loops in PyTeal.

Syntax:

```
While(loop-condition).Do(loop-body)
```

- loop-condition - an expression that should be evaluated to TealType.uint64.
- loop-body - an expression that should be evaluated to TealType.none.
- loop-body is executed as long as loop-condition returns a true value (> 0).

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Cycles in PyTeal: While

Example of using While

Let's look at the code that uses ScratchVar to iterate through each transaction in the current group and sum up all their fees:

```
totalFees = ScratchVar(TealType.uint64) # variable for storing the
  total amount of fees
i = ScratchVar(TealType.uint64) # counter variable for iteration


Seq([
    i.store(Int(0)),
    totalFees.store(Int(0)),
    While(i.load() < Global.group_size()).Do(
        totalFees.store(totalFees.load() + Gtxn[i.load()].fee()),
        i.store(i.load() + Int(1))
    )
])
```

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Cycles in PyTeal: For

For loops allow you to perform a series of actions repeatedly based on an initial value, a loop condition, and an iteration step.

Syntax:

```
For(loop-start, loop-condition, loop-step).Do(loop-body)
```

- **loop-start**: An initial action that is performed once before the loop starts.
- **loop-condition**: A condition that is checked before each iteration of a loop. If the condition is true (> 0), the loop body is executed.
- **loop-step**: An action that is performed after each iteration of a loop.
- **loop-body**: The body of the loop that is executed in each iteration if the condition is true.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Cycles in PyTeal: For

Example of using For

```
totalFees = ScratchVar(Tealtype.uint64)
i = ScratchVar(TealType.uint64)

Seq([
    totalFees.store(Int(0)),
    For(i.store(Int(0)), i.load() < Global.group_size(),
i.store(i.load() + Int(1))).Do(
        totalFees.store(totalFees.load() + Gtxn[i.load()].fee())
    )
])
```

The code below uses a ScratchVar to iterate over each transaction in the current group and sum up all their fees. The code here is functionally equivalent to the While loop example given earlier.

Source: https://pyteal.readthedocs.io/en/latest/control_structures.html?highlight=Cond#chaining-tests-cond

Exiting loops in PyTeal: **Continue and Break**

The Continue and Break expressions can be used to exit the While and For loops.

- `The Continue` statement tells the program to skip the rest of the loop body and move to the next iteration. The loop continues as long as the loop condition remains true.
- The Break statement tells the program to exit the current loop completely. The loop is terminated regardless of whether the loop condition remains true.

37

Continue. Example of use

Consider an example where we iterate through each transaction in the current group and count the number of payment transactions using the **`Continue` operator**.

```
numPayments = ScratchVar(TealType.uint64)
i = ScratchVar(TealType.uint64)

Seq([
    numPayments.store(Int(0)),
    For(i.store(Int(0)), i.load() < Global.group_size(), i.store(i.load() + Int(1))).Do(
        If(Gtxn[i.load()].type_enum() != TxnType.Payment)
        .Then(Continue()),
        numPayments.store(numPayments.load() + Int(1))
    )
])
```

In this example, if the transaction type is not payment (TxnType.Payment), the Continue operator skips the rest of the loop body and moves to the next iteration. Otherwise, the counter of payment transactions is incremented by one.

Break. Example of use

Consider an example where we are looking for the index of the first payment transaction in the current group using the **Break operator**.

```
firstPaymentIndex = ScratchVar(TealType.uint64)
i = ScratchVar(TealType.uint64)

Seq([
    # store a default value in case no payment transactions are found
    firstPaymentIndex.store(Global.group_size()),
    For(i.store(Int(0)), i.load() < Global.group_size(), i.store(i.load() + Int(1))).Do(
        If(Gtxn[i.load()].type_enum() == TxnType.Payment)
        .Then(
            firstPaymentIndex.store(i.load()),
            Break()
        )
    ),
    # assert that a payment was found
    Assert(firstPaymentIndex.load() < Global.group_size())
])
```

In this example, if the transaction type is payment, we store the index of this transaction and use the Break statement to exit the loop. This allows us to find the first payment transaction and avoid unnecessary iterations.

Compilation

The `compileTeal` function allows you to compile a PyTeal smart contract in TEAL.

Example:

```
# Compile programs in TEAL
compiled_approval = compileTeal(approval, mode=Mode.Application, version=3)

compiled_clear = compileTeal(clear, mode=Mode.Application, version=3)
```

- The first argument is a PyTeal program that executes certain smart contract logic. Usually, this is approval_program().
- The second argument specifies the compilation mode. This can be Mode.Application - the smart contract will be used as an application on the Algorand blockchain, or Mode.Signature, which is used to write the logic for signing transactions.
- The third argument specifies the version of TEAL that will be used for compilation.

The result of calling the compileTeal method is a string representing the compiled TEAL code.

Compiling the code and writing to a file

```
if __name__ == "__main__":
    approval_program = approval()
    clear_program = clear()
    # This is where the PyTeal code is compiled to TEAL
    approval_teal = compileTeal(approval_program, mode=Mode.Application, version=6)
    clear_state_teal = compileTeal(clear_program, mode=Mode.Application, version=6)

    with open("approval.teal", "w") as f:
        f.write(approval_teal)
        f.close()

    with open("clear.teal", "w") as f:
        f.write(clear_state_teal)
        f.close()
```

Additional materials

- PyTeal for beginners (Video course). In this course, you'll learn how to get started building Algorand smart contracts with PyTeal and dive deep into each of the PyTeal operations you'll need to write complex and powerful smart contracts.
- Algorand PyTeal Github repository. This repository contains examples of smart contracts and smart signatures written in PyTeal.
- PyTeal Fundamentals (Course). This PyTeal course provides a comprehensive introduction to writing smart contracts for the Algorand blockchain using PyTeal. Students will learn key concepts such as routers, expressions, AVM and ABI data types, transaction fields, atomic transfers, and subroutines. It is possible to obtain an NFT certificate.