PYTEAL

PyTeal Smart Contract Development

Overview

PyTeal is a powerful toolset designed to simplify the development of Algorand smart contracts. By leveraging Python's expressive syntax and providing a rich set of abstractions, PyTeal empowers developers to write secure, efficient, and maintainable blockchain applications. This comprehensive guide covers the fundamental topics from installation and examples to advanced topics like compiler optimization and ABI interfaces providing an essential reference for anyone looking to dive into smart contract development on the Algorand platform.

What is PyTeal?

PyTeal is a Python language binding for Algorand's Transaction Execution Approval Language (TEAL). It allows developers to write Algorand smart contracts using a higher-level Python syntax that then compiles down to the TEAL language. The main benefits of PyTeal include:

- **Ease of Use:** Write smart contracts in Python, leveraging Python's familiar syntax and programming paradigms.
- **Abstraction:** Abstract away low-level TEAL instructions into higher-level constructs.
- **Maintainability:** Improved readability and maintainability compared to raw TEAL code.
- **Rapid Development:** Develop, test, and deploy Algorand smart contracts faster using Python's ecosystem.

The overview section in the documentation introduces the following aspects:

- **Motivation:** Simplifies the development of smart contracts on Algorand by using Python.
- **Philosophy:** Emphasizes clarity and conciseness while preserving the performance and safety guarantees provided by TEAL.

- **Target Audience:** Developers looking to build decentralized applications (dApps) and smart contracts on Algorand using high-level languages.

Core Features

- **High-Level Abstractions:** Write contracts using familiar Python syntax.
- **Rich Expression Support:** Includes arithmetic, byte-wise operations, and access to transaction fields.
- **Advanced Constructs:** Offers control structures, stateful smart contract management, and secure handling of assets.
- **Optimization and Debugging:** Includes compiler optimization options and sourcemap generation to trace back the generated TEAL code to your PyTeal source.
- **ABI Support:** Allows the creation of contracts that expose functions and methods for standardized interaction with external applications.

Installation

- **Pre-requisites:** Before using PyTeal, ensure you have Python installed (typically Python 3.6+ is recommended).
- **Dependencies:**

  PyTeal might depend on certain standard libraries or specific versions of Python. Make sure your environment satisfies these requirements.

- **Virtual Environment:**

  It is recommended to install PyTeal within a virtual environment to avoid conflicts with other packages.

- **Development Installation:**

  For contributors or those who wish to work on the source code, instructions on cloning the repository and setting up the development environment (using pip’s editable installs) are provided.

Additionally, troubleshooting steps for common installation issues (such as dependency conflicts or Python version mismatches) are discussed.

I

nstallation Method:

- You can install PyTeal via PyPI using pip.

```
pip install pyteal
```

Examples

The documentation features a wide range of examples to help developers understand how to use PyTeal in practice:

- **Basic Smart Contract:** A simple contract that illustrates condition checking and returns a boolean result.
- **Conditional Logic:** Examples demonstrating if/else constructs and logical operators within smart contracts.
- **Asset Management:** Code samples that interact with Algorand Standard Assets (ASA).
- **Stateful Contracts:** Detailed examples showing how to manage global and local state within a contract.
- **Complex Contracts:** Multi-condition contracts and group transactions showing advanced use cases.

These examples are meant both to provide a quick-start guide and to serve as a reference when building more complex applications.

Data Types

PyTeal introduces abstractions for managing data types that map directly to TEAL's own data constructs. The documentation covers:

- **Primitive Types:**
  - **Integers:** Represented as Python integers, used for counters, indices, and loop counters.
  - **Byte Strings:** Often used to handle keys, addresses, or arbitrary byte data.

    - **Booleans:** True/False values that control flow within smart contracts.

- **Complex Structures:**

  Composed types that represent more complicated state or interactions, useful in advanced contract designs.

- **Type Checking:**

  How PyTeal enforces type compatibility and conversions between Python types and TEAL types.

Understanding these data types is crucial since smart contract logic, arithmetic, and state operations depend on their correct usage.

- `Int(n)` – Creates a 64-bit unsigned integer (`TealType.uint64`)
- `Bytes("string")` – Creates a byte slice (`TealType.bytes`)
- `Bytes("base16" | "base32" | "base64", value)` – Encodes values from respective formats
- `Itob(n)` – Converts `uint64` to `bytes` (big-endian)
- `Btoi(b)` – Converts `bytes` to `uint64`

Arithmetic Expressions

Arithmetic is a fundamental component of smart contracts. In PyTeal:

- **Operators:**

  PyTeal provides operator overloading for Python's arithmetic operators (+, -, *, /, %). These operators are compiled into their TEAL counterparts.

- **Expression Trees:**

  When you write an arithmetic expression in PyTeal, it builds an underlying expression tree that the compiler later translates into TEAL code.

- **Optimization Considerations:**

  The PyTeal compiler may optimize arithmetic operations if certain conditions (like constant folding) are met.

- **Examples:**

Code examples illustrate addition, subtraction, multiplication, and division, as well as more complex expressions composed of several arithmetic operations.

- `Add(a, b)`, `Minus(a, b)`, `Mul(a, b)`, `Div(a, b)`, `Mod(a, b)`, `Exp(a, b)`
- Comparisons: `Lt`, `Gt`, `Le`, `Ge`, `Eq`, `Neq`
- Logical: `And`, `Or`, `Not`
- Bitwise: `BitwiseAnd`, `BitwiseOr`, `BitwiseXor`, `BitwiseNot`

Byte Expressions

Byte-level operations are often necessary when working with addresses, cryptographic operations, or raw data:

- **Concatenation:**

  Combine multiple byte strings using the appropriate PyTeal functions.

- **Subbytes and Slicing:**

  Extract specific sections of a byte string.

- **Conversion:**

  Converting between numeric types and byte strings, and vice versa.

- **Comparison:**

  Implementing equality checks and ordering on byte arrays.

- **Usage Examples:**

  Demonstrative examples help clarify common byte operations encountered in smart contracts.

- Comparisons: `BytesLt`, `BytesGt`, `BytesEq`, `BytesNeq`, etc.
- Operations: `BytesAdd`, `BytesMinus`, `BytesMul`, `BytesDiv`, `BytesMod`, `BytesAnd`, `BytesOr`, `BytesXor`, `BytesNot`, `BytesZero`

Accessing Transaction Fields

Smart contracts frequently need to inspect data coming from transactions. PyTeal provides a convenient way to:

- **Retrieve Standard Fields:**

  Access fields such as `sender`, `receiver`, `amount`, `fee`, and more.

- **Safety and Validation:**

  Ensure that data types are correctly interpreted and validated within the context of the blockchain's state.

- **Dynamic Conditions:**

  Use transaction field data to implement conditional logic; for instance, approving a transaction only if it meets certain criteria.

- **Group Transactions:**

  How to access and validate fields when multiple transactions are grouped together for atomicity.

Key Transaction Field Functions

- `Txn.sender()`, `Txn.fee()`, `Txn.note()`, `Txn.group_index()`
- App-specific: `Txn.application_id()`, `Txn.application_args`, `Txn.approval_program()`
- Asset-related: `Txn.xfer_asset()`, `Txn.asset_amount()`, `Txn.asset_receiver()`
- Rekeying: `Txn.rekey_to()`
- Group/Inner Txn: `Gtxn[...]`, `InnerTxnBuilder`, `Txn.group_index()`

Global Parameters

- Use `Global.round()`, `Global.latest_timestamp()`, `Global.group_size()` etc.

Cryptographic Functions

Security is at the heart of blockchain smart contracts. PyTeal includes built-in primitives for cryptography:

- **Hashing:**

  Functions to compute secure hash digests, such as SHA-256 or keccak256.

- **Signature Verification:**

Verify signatures against public keys, which is crucial for validating transactions.

- **Randomness and Salting:**

  Limited functions for generating pseudo-random values or salting operations are available.

- **Usage Patterns:**

  Real-world examples demonstrate how to perform cryptographic checks within a smart contract.

- `Sha256(e)`, `Sha3_256(e)`, `Keccak256(e)`, `Sha512_256(e)` – Common hashing algorithms
- `MiMC(e)` – Lightweight hash used in zk-related applications
- `Ed25519Verify(d, s, p)`, `Ed25519Verify_Bare(d, s, p)` – Signature verification
- `EcdsaVerify(...)`, `EcdsaDecompress(...)`, `EcdsaRecover(...)` – ECDSA operations for public key

Scratch Space

Scratch space in TEAL provides temporary storage for intermediate values:

- **Definition and Purpose:**

  Scratch space in PyTeal is a series of slots that can be used to store temporary variables.

- **Declaration and Access:**

  How to allocate scratch variables and read/write from those slots safely.

- **Lifetime:**

  Scratch space values exist only for the duration of the transaction execution, making them ideal for intermediate calculations.

- **Examples:**

  Code samples showing the declaration, initialization, and manipulation of scratch variables.

- `ScratchVar(TealType)` – Defines a temporary variable (slot auto-assigned or manually set).

- `.store(value)` – Writes a value to scratch space.
- `.load()` – Reads a value from scratch space.

For dynamic usage:

- `DynamicScratchVar(TealType)`
- `.set_index(scratch_var)` – Points to a specific scratch slot.
- `.store()` / `.load()` – Operate through a dynamic reference.

Loading Group Transactions

Group transactions allow multiple transactions to be submitted together and executed atomically. In PyTeal:

- **Group Context:**

  How to retrieve and work with the properties of transactions within a group.

- **Indexing Transactions:**

  Accessing a specific transaction's fields based on its index in the group.

- **Atomicity Checks:**

  Ensuring that operations across multiple transactions either all succeed or all fail.

- **Common Patterns:**

  Examples that demonstrate the creation and validation of group transactions.

Group Transaction Utilities in PyTeal

- `GeneratedID(index)` – Accesses the ID of a newly created app or asset from a prior transaction in the same atomic group.
- `ImportScratchValue(txn_index, slot_id)` – Loads a scratch slot value from another transaction in the group.

Control Structures

Control flow in PyTeal is designed to be expressive while mapping neatly to TEAL's control structures:

- **Conditional Branching:**

  `If-Else` constructs to execute different code paths based on contract state or transaction parameters.

- **Loops and Iteration:**

  Although loops are less common in TEAL due to resource constraints, PyTeal provides idiomatic ways to express limited iterative logic.

- **Logical Operators:**

  Combining conditions using AND, OR, and NOT operations.

- **Programmatic Construction:**

  How control structures form the backbone of more complex decision-making logic within smart contracts.

the key **control structure functions and constructs** from PyTeal:

- `Approve()`, `Reject()` – Ends program with success/failure.
- `Seq([...])` – Executes expressions in sequence.
- `If(cond, then, else)` – Conditional branching.
- `Assert(cond)` – Fails if `cond` is false.
- `Cond((cond1, expr1), (cond2, expr2), ...)` – Conditional evaluation chain.
- `While(cond).Do(body)` – Loops while condition is true.
- `For(init, cond, incr).Do(body)` – Traditional loop.
- `Break()`, `Continue()` – Loop control.
- `Subroutine(...)` – Function abstraction.

State Management

State management is critical in designing both stateless and stateful smart contracts:

- **Global State:**

  Variables stored on-chain that are accessible to all transactions interacting with a smart contract.

- **Local State:**

  Data stored per account that interacts with the smart contract, allowing for personalized contract experiences.

- **State Transitions:**

  How state changes occur based on transaction outcomes, together with mechanisms to guard state updates.

- **Storage Operations:**

  Read and write operations, and constraints imposed by Algorand on state storage.

the key
State Management functions in PyTeal:

Global State

- `App.globalPut(key, value)` – Write to global state.
- `App.globalGet(key)` – Read from global state.
- `App.globalDel(key)` – Delete from global state.
- `App.globalGetEx(app_id, key)` – Read global state from another app (`MaybeValue`).

Local State

- `App.localPut(account, key, value)` – Write to account's local state.
- `App.localGet(account, key)` – Read from account's local state.
- `App.localDel(account, key)` – Delete from account's local state.
- `App.localGetEx(account, app_id, key)` – Read local state from another app.

Box Storage

- `App.box_create`, `App.box_put`, `App.box_get`, `App.box_delete`, etc.

Assets

Algorand supports native assets (Algorand Standard Assets - ASA). PyTeal includes support for:

- **Asset Creation:**

  Steps and necessary conditions for creating a new asset using PyTeal.

- **Asset Transfer and Management:**

  Code patterns for transferring assets, setting up clawback mechanisms, and other asset-related operations.

- **Asset-Specific Functions:**

  How to interact with asset-specific fields and validate asset-related transactions.

the key **Asset-related functions** in PyTeal:

Algo Balances

- `Balance(account)` – Get microAlgos held by an account.
- `MinBalance(account)` – Get minimum balance requirement for an account.

ASA Holdings (`AssetHolding`)

- `.balance(account, asset_id)` – ASA units held.
- `.frozen(account, asset_id)` – Frozen status of asset.
- `.value()` , `.hasValue()` – Retrieve the value or check if it exists.

ASA Parameters (`AssetParam`)

- `.total()` , `.decimals()` , `.defaultFrozen()`
- `.unitName()` , `.name()` , `.url()` , `.metadataHash()`
- `.creator()` , `.manager()` , `.reserve()` , `.freeze()` , `.clawback()`

Versions

Given the evolution of the Algorand blockchain and TEAL itself, the documentation discusses:

- **TEAL Versions:**

  Which version of the TEAL language is being targeted, and what new features or restrictions each version brings.

- **PyTeal Versions:**

  Correspondence between PyTeal releases and the underlying TEAL language support.

- **Backward Compatibility:**

  Ensuring that smart contracts written in earlier versions can operate in newer runtime environments.

- **Migration Guides:**

  Tips and instructions on how to upgrade existing contracts to take advantage of new PyTeal features.

PyTeal Versioning Functions and Info

- **AVM version compatibility** is tied to the **PyTeal version**, with mappings (e.g., PyTeal ≥ 0.26.0 targets AVM v10).
- `pragma()` – Used to assert and enforce a specific PyTeal version in your contract to maintain compatibility.
- `Pragma(...)` – Apply version constraints to sections of code.

This ensures your contracts compile consistently across environments and avoid future-breaking changes.

Compiler Optimization

The PyTeal compiler not only translates Python code to TEAL code, it also performs optimizations:

- **Constant Folding:**

  The process of computing constant expressions during compile time to reduce runtime overhead.

- **Instruction Simplification:**

Reducing complex instruction sequences into simpler ones without changing semantics.

- **Resource Constraints:**

  Optimizations aimed at ensuring that compiled TEAL code meets the strict resource limits imposed by the Algorand Virtual Machine.

- **Debugging and Efficiency:**

  Discussion on how these optimizations influence debugging and the performance of the smart contract on-chain.

the **Compiler Optimization functions and options** in PyTeal:

- `OptimizeOptions(...)` – Accepts optimization flags like:
  - `scratch_slots=True|False` – Optimize scratch space.
  - `frame_pointers=True|False` – Enable/disable frame pointer optimization.
- These options are passed into:
  - `compileTeal(...)`
  - `Router.compile_program(...)`

You can omit `OptimizeOptions` to use version-dependent defaults.

Source Map

To help bridge the gap between high-level PyTeal code and the generated TEAL, source maps are provided:

- **Purpose:**

  They relate locations in the PyTeal code to corresponding locations in the compiled TEAL code.

- **Usage:**

  Helpful for debugging and for tools that need to map runtime errors back to the original source code.

- **Format and Tools:**

Explanation of the source map format and potential integrations with development tools.

To enable **source mapping** in PyTeal:

1. Import and enable it using `FeatureGates.set_sourcemap_enabled(True)` *before* importing PyTeal.
2. Use `compile(...)` or `Router.compile(...)` with `with_sourcemaps=True`, `annotate_teal=True`, and related flags.
3. The result contains annotated TEAL with line numbers, source file paths, and program counters for better debugging.

Perfect for tracing back generated TEAL to your PyTeal code.

Opup

(Note: The "opup" section in the documentation is less standard than the other topics. It may refer to experimental features or a specific module that provides additional functionality such as user interface popups for contract debugging or a utility package. In some cases, this could also be a typographical error in the docs. The key points include:)

- **Utility Functions:**

  Functions or helpers provided by PyTeal that may enhance the developer experience.

- **Experimental Features:**

  If "opup" refers to experimental constructs, documentation may explain how to safely use or test these features.

- **Usage Guidelines:**

  Best practices and warnings about relying on non-stable features until they are fully supported.

For the most accurate description, please check the latest version of the documentation or release notes.

- `OpUp(mode, target_app_id)`: Initializes the budget augmenter in a selected mode (e.g., `Explicit`, `OnCall`) for the target app.

- `ensure_budget(Int)`: Ensures opcode budget is available for intensive operations.
- `maximize_budget(Int)`: Tries to max out the opcode budget within a fee limit.

Useful for boosting budget via inner calls to a budget app.

ABI (Application Binary Interface)

The ABI section covers the interaction between smart contracts and client applications:

- **Contract Interfaces:**

  Defining the callable functions (methods) within a smart contract.

- **Type Signatures:**

  How to specify input and output types for smart contract methods.

- **Encoding and Decoding:**

  The process of converting between native Python types and the on-chain binary format.

- **Integration:**

  Examples of how to integrate PyTeal smart contracts with client libraries and higher-level SDKs using the ABI specifications.

- **Benefits:**

  ABI support makes it easier to interact with smart contracts, abstracting much of the binary interfacing details.

- **ABI Types**: `abi.Uint8()`, `abi.String()`, `abi.Bool()`, etc.
- `TypeSpec.new_instance()`: Instantiates an ABI type.
- `abi.make(abi_type)`: Simplified instance creation.
- `ABIReturnSubroutine`: Decorates functions with ABI I/O.

Enables ARC-4 smart contract compatibility.

API Reference

The API section is a complete reference guide for all classes, functions, and constants provided by PyTeal:

- **Modules and Packages:**

  Detailed breakdown of each module, including data types, expressions, and helper functions.

- **Function Documentation:**

  For every function, the documentation includes:

  - **Parameters and Return Types**
  - **Usage Examples**
  - **Edge Cases and Exceptions**

- **Extensibility:**

  Guidelines on how to extend or customize PyTeal functions if necessary.

- **Best Practices:**

  Suggestions to write clear, efficient, and secure smart contracts using the API.

- **Versioning Notes:**

  Any changes between versions of PyTeal that impact the API, helping developers migrate their code if needed.

The **PyTeal API Reference** includes detailed documentation for all modules, functions, and classes. It covers:

- `Txn`, `Gtxn`, `InnerTxn` – Access transaction fields.
- `Expr`, `Int`, `Bytes` – Core expression types.
- `App`, `AssetHolding`, `AccountParam` – Access state, assets, and accounts.
- `abi` submodule – For ARC-4 ABI-compliant methods.
- `compileTeal()`, `Mode`, `OptimizeOptions`, etc.

These are the building blocks of PyTeal smart contracts.

Key Concepts

1. **Expressions**: Represent TEAL operations (arithmetic, logic, etc.)
2. **Approval and Clear Programs**: Two programs required for stateful applications
3. **Subroutines**: Reusable code blocks within smart contracts
4. **State Management**: Global and local state operations
5. **Transaction References**: Accessing transaction fields and properties

Program Modes

- `Mode.Application`: For stateful contracts
- `Mode.Signature`: For stateless contracts

Core Types

- `TealType.uint64`: 64-bit unsigned integers
- `TealType.bytes`: Byte strings (for text and binary data)

Basic Structure

```
from pyteal import *

def approval_program():
    # Program logic here
    return Approve()

def clear_program():
    return Approve()

# Compile to TEAL
compiled_approval = compileTeal(approval_program(), mode=Mode.Applicatio
n, version=6)
compiled_clear = compileTeal(clear_program(), mode=Mode.Application, versi
on=6)
```

Common PyTeal Patterns

1. **Basic Token Contract**
2. **Escrow Accounts**
3. **Atomic Swaps**
4. **Voting Applications**
5. **Oracle Data Feeds**

Standard Libraries

- ARC4 ABI
- Base64
- SHA256
- Ed25519
- And more cryptographic primitives

Security Considerations

- Recursion and loop limitations
- Deterministic execution
- Limited opcodes and computational resources
- Transaction validation best practices
- Signature verification requirements

Beaker Framework

What is Beaker?

Beaker is a higher-level framework built on top of PyTeal for creating Algorand smart contracts with a more intuitive API. It provides class-based application development, decorators for ABI methods, and state management utilities.

Installation

```
pip install beaker-pyteal
```

Key Features

- Class-based application structure
- Built-in state management
- Method decorators for different transaction types
- ABI method support for type-safe interfaces
- Testing utilities

Application Structure

- State variables with type specification
- Method decorators (`@create`, `@update`, `@delete`, `@external`)
- ABI methods with proper typing

Basic Structure

```
from beaker import Application, GlobalStateValue
from pyteal import *

class MyApp(Application):
    counter = GlobalStateValue(stack_type=TealType.uint64, default=Int(0))

    @create
    def create(self):
        return self.initialize_global_state()

    @external
    def increment(self):
        return self.counter.set(self.counter + Int(1))

app = MyApp()
```

Blueprint Generation

- ABI specification for frontend integration

Algorand Smart Contracts (ASC1)

Types

- **Stateful contracts**: Applications with persistent state storage
- **Stateless contracts**: Logic signatures that approve transactions

Execution Model

- Written in TEAL (Transaction Execution Approval Language) or PyTeal (Python wrapper for TEAL)
- Deterministic execution with limited computational resources
- Security features: no loops, bounded execution time, predictable resource usage
- ABI support: Cross-language compatibility for contract method calls
- Application call types: NoOp, OptIn, CloseOut, ClearState, DeleteApplication
- Inner transactions: Contracts can issue transactions to other accounts or contracts

Smart Contract Patterns

Token Contracts

- **Fungible tokens**: ASA-based or custom application logic
- **Non-fungible tokens**: Unique assets with metadata
- **Semi-fungible tokens**: Batch minting with unique properties
- **Frozen status**: Control over transferability
- **Clawback functionality**: Asset recovery mechanisms

DeFi Primitives

- **Atomic swaps**: Trustless exchange of assets

- **Escrow accounts**: Conditional fund release
- **Time-locked contracts**: Scheduled operations
- **Automated market makers**: On-chain liquidity provision
- **Yield farming**: Incentive distribution mechanisms

Oracles and Bridges

- **Data feeds**: External information injection
- **Cross-chain bridges**: Inter-blockchain communication
- **State proofs**: Verifiable blockchain state