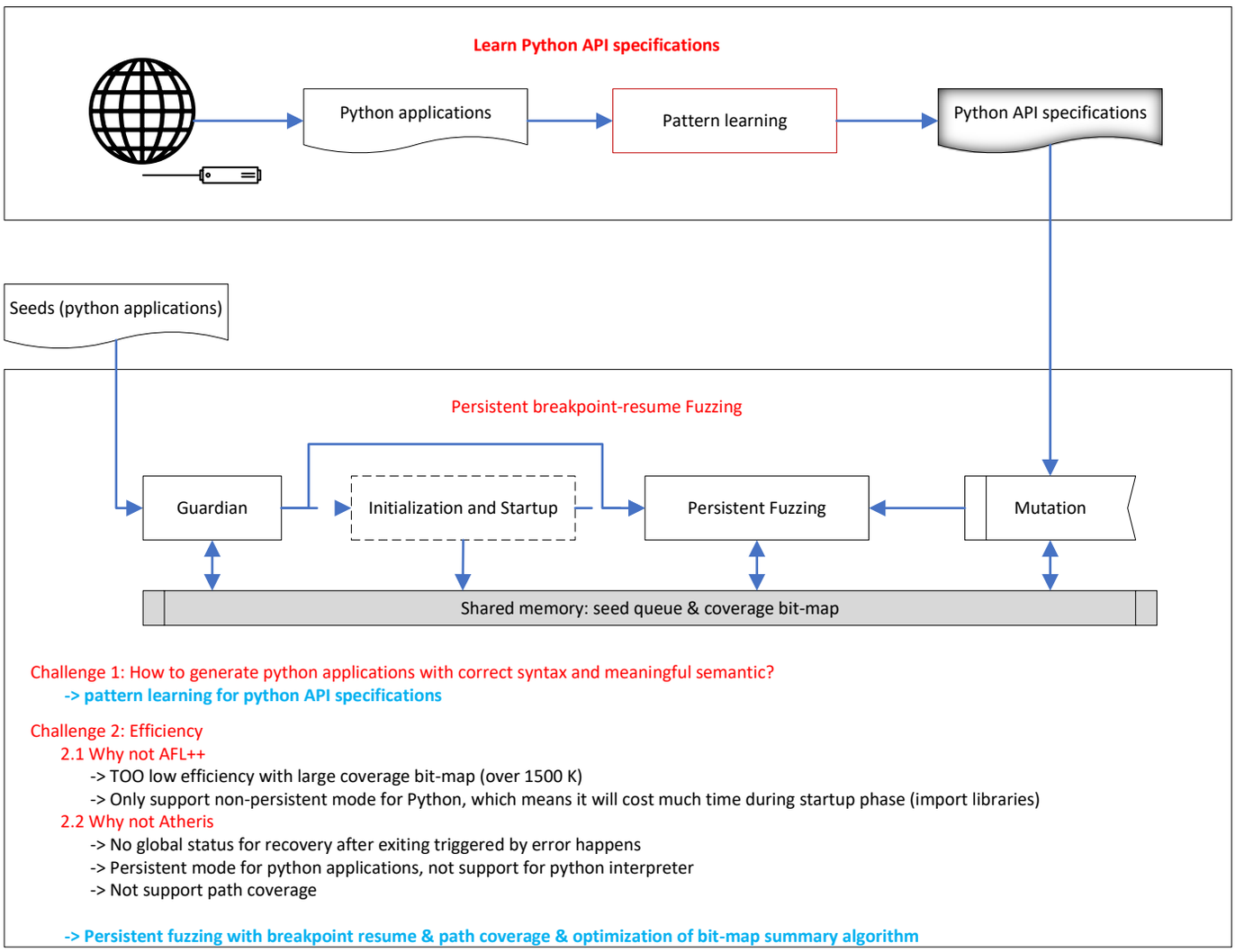
Overview



Challenge 1: How to generate python applications with correct syntax and meaningful semantic?

-> pattern learning for python API specifications

Challenge 2: Efficiency

2.1 Why not AFL++

-> TOO low efficiency with large coverage bit-map (over 1500 K)

-> Only support non-persistent mode for Python, which means it will cost much time during startup phase (import libraries)

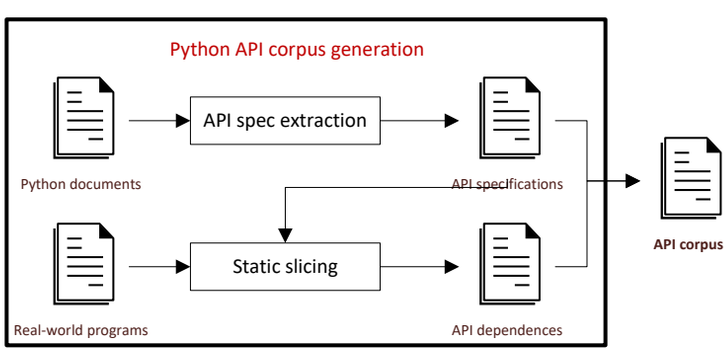
2.2 Why not Atheris

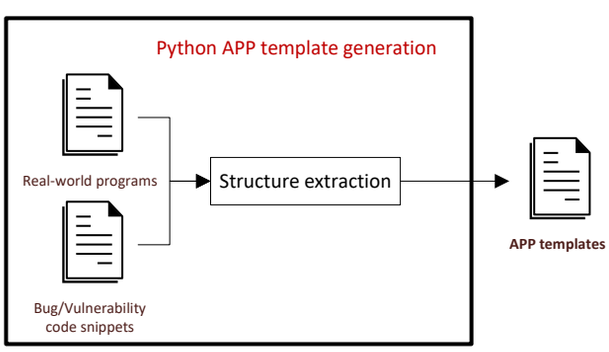
-> No global status for recovery after exiting triggered by error happens

-> Persistent mode for python applications, not support for python interpreter

-> Not support path coverage

-> Persistent fuzzing with breakpoint resume & path coverage & optimization of bit-map summary algorithm



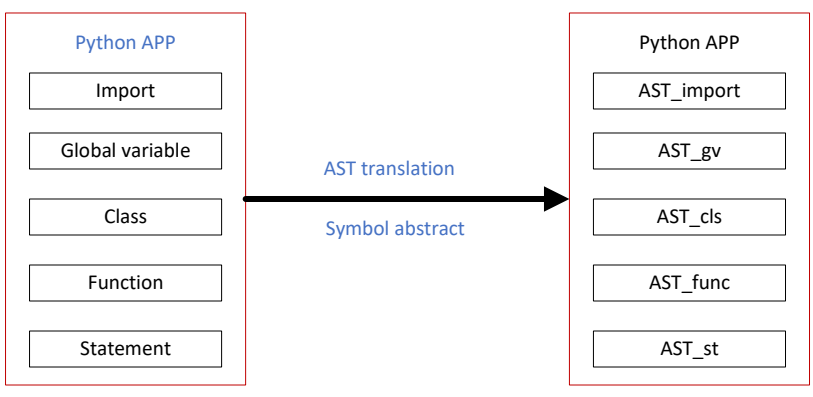


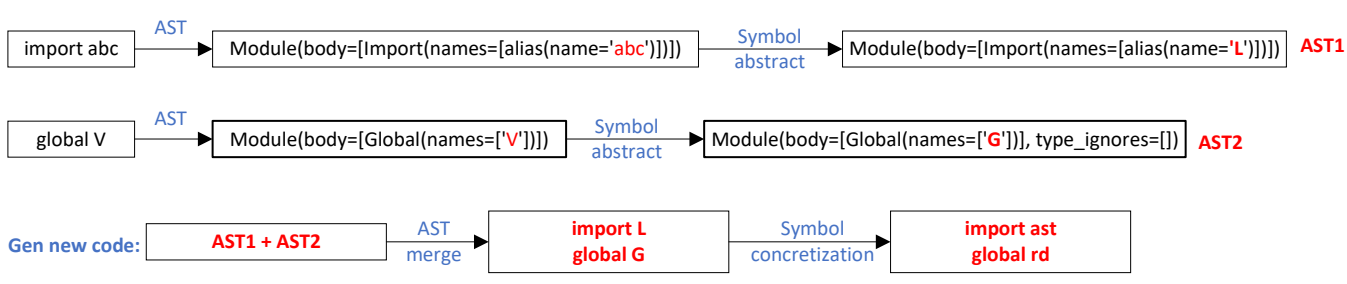
Python APP generation

1 **General python APP generation**

1. Learn the grammar from Python applications
   - translate APP into ASTs
   - for each type of element, **learn** the patterns and normalize the format through abstract symbol

2. Gen new APP through operation (ADD/DEL/MERGE/RECURSIVE...) of ASTs





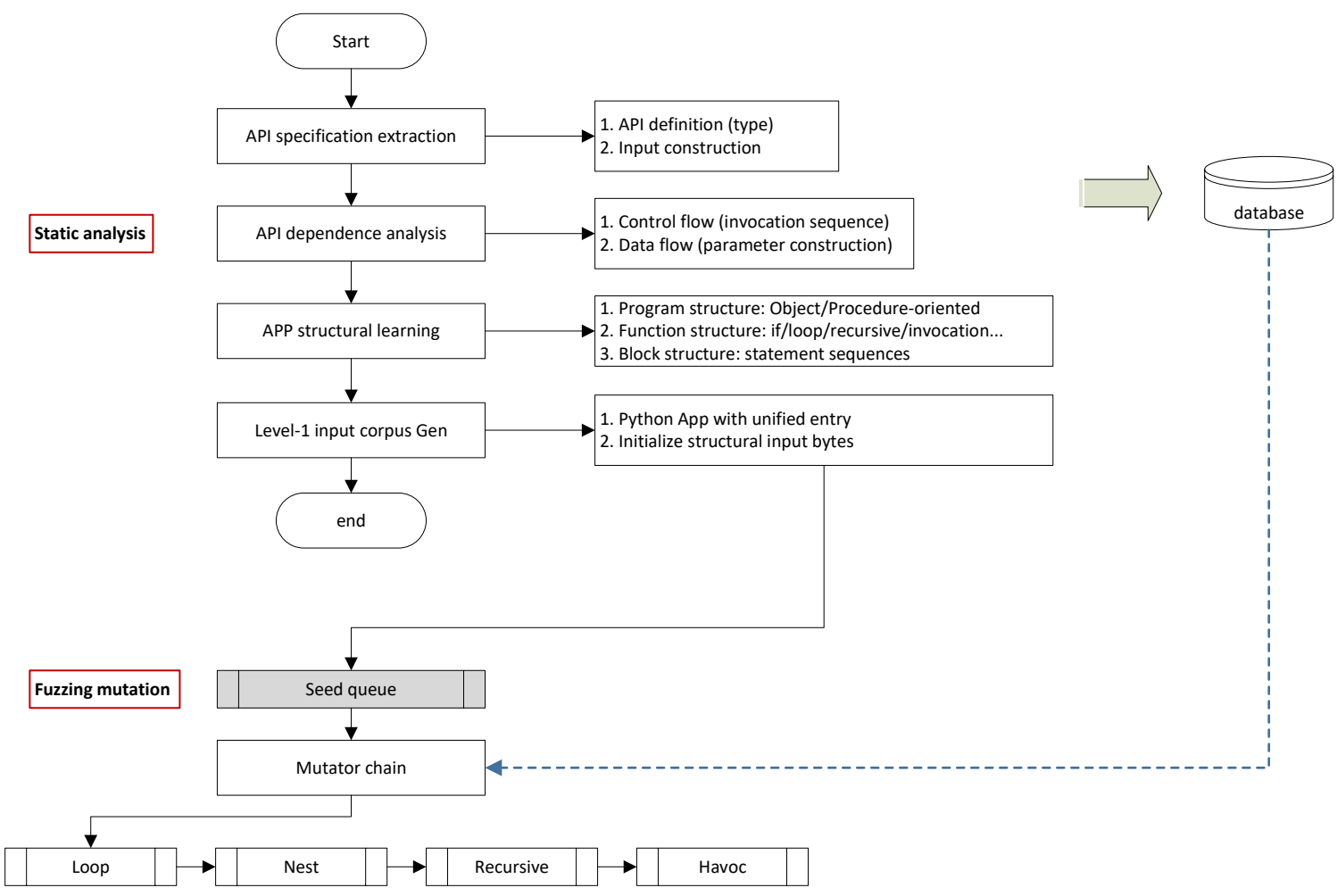
2 **Python APP generation focusing on usage of runtime libraries**

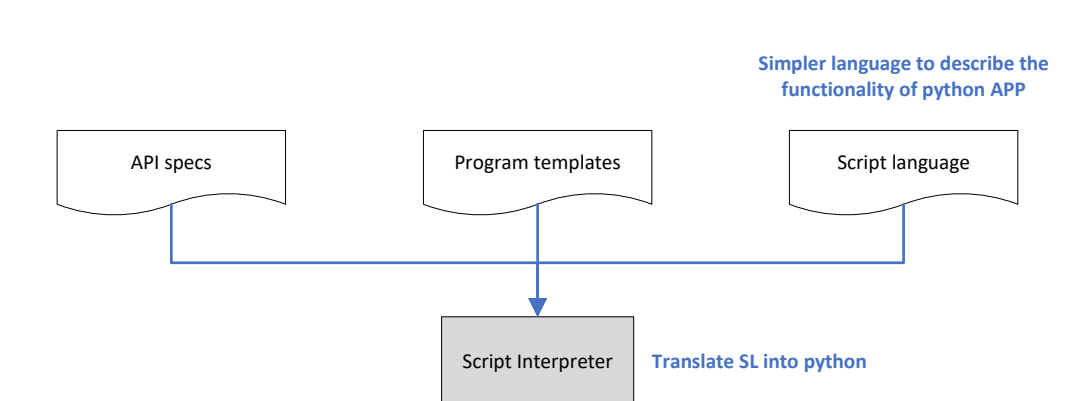
The usage of API is in form of call statement, hence we only need to learn all the possible call statements of these APIs and store them as ASTs

3 **Possible dependencies of runtime library APIs**

For possible dependencies among APIs, we can learn from the real world programs through static slicing

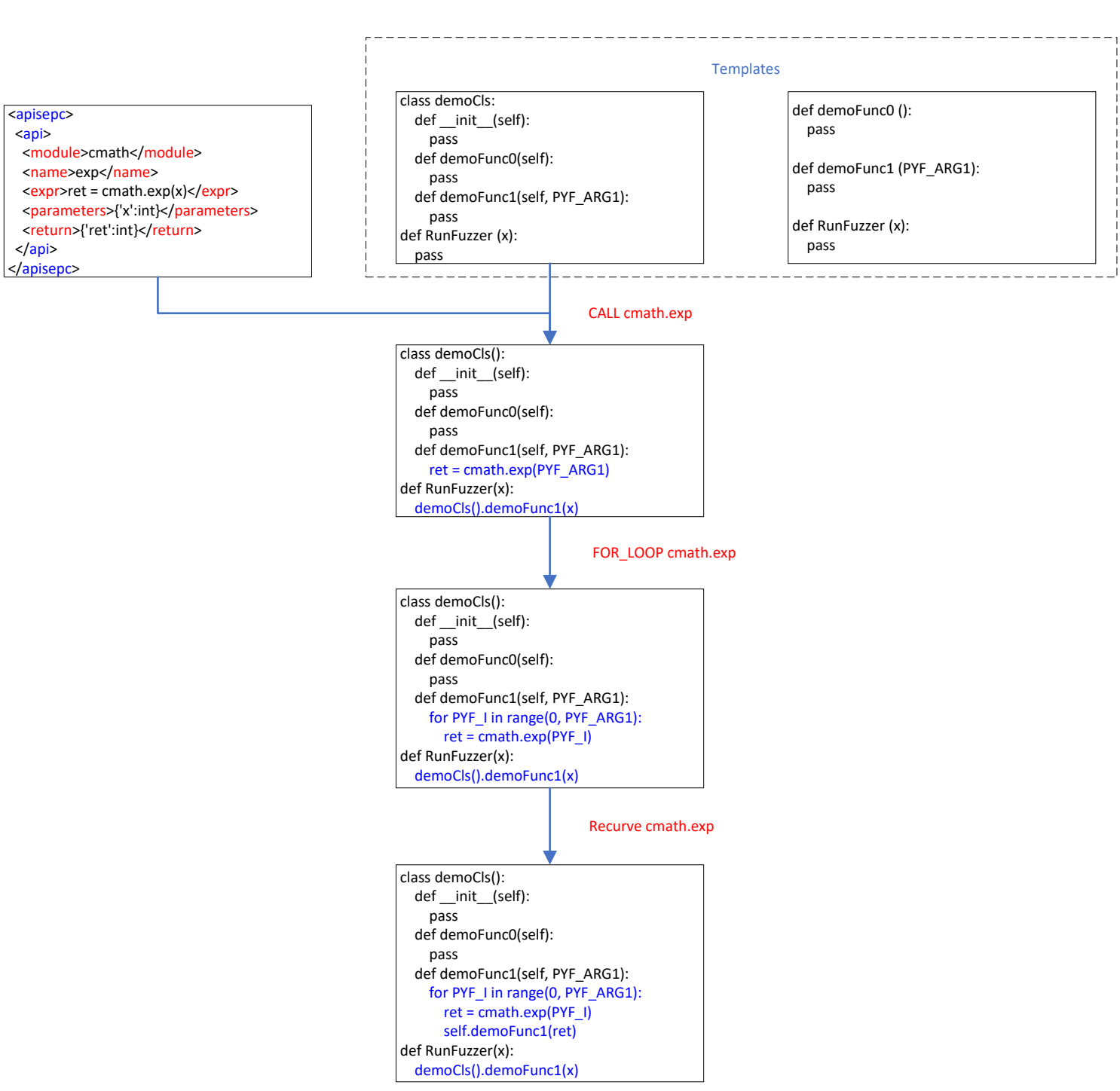
Software design



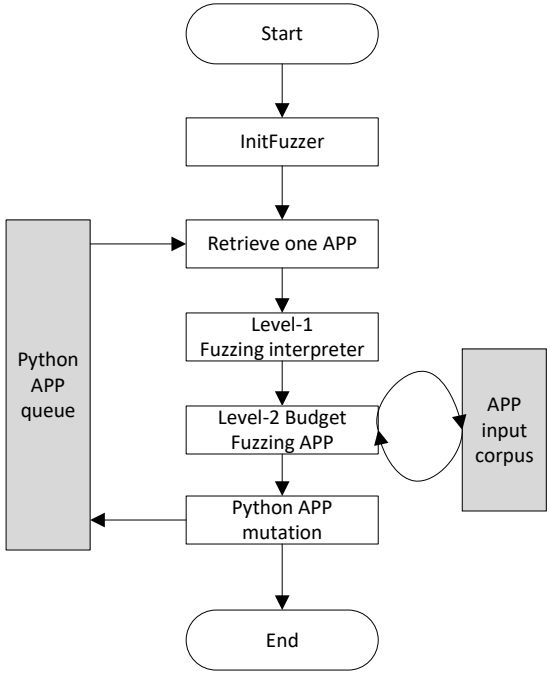


Level-1 Fuzzing type: Generation-based / Mutation-based → **Generation-based fuzzing**

Example



Two-level Fuzzing



Level-1 Fuzzing: Targeting interpreter Core. We use infinite budget at level-1.

Level-2 Fuzzing: Targeting runtime libraries. We use finite budget at level-2 until no favored path/block/feature found.

```
<apisepc>
  <library email>
    <module charset>
      <class Charset>
        <api>
            <name>get_body_encoding</name>
            <expr>ret = get_body_encoding()</expr>
            <parameters>{}</parameters>
            <return>{'ret':base64|7bit}</return>
            <dependences> </dependences>
        </api>

        <api>
            <name>header_encode</name>
            <expr>header_encode(str)</expr>
            <parameters>{'str':string}</parameters>
            <return>{}</return>
            <dependences> </dependences>
        </api>
      </class>
    </module>

    <module contentmanager>
      <api>
        <name>header_encode</name>
        <expr>header_encode(str)</expr>
        <parameters>{'str':string}</parameters>
        <return>{}</return>
        <dependences> </dependences>
      </api>
    </module >

    <errors>
      <exception>MessageError</exception>
      <exception>MessageParseError</exception>
      <exception>BoundaryError</exception>
      <exception>MultipartConversionError</exception>
    </errors>
  </library>
</apisepc>
```

Object-oriented

Procedure-oriented

Program structure

```
class demoCls:
   def __init__(self):
      pass
   def demoFunc0(self):
      pass
   def demoFunc1(self, arg1):
      pass

def RunFuzzer (x):
   pass
```

```
class demoCls:
   def __init__(self):
      pass
   def demoFunc0(self):
      pass
   def demoFunc1(self, arg1):
      pass

def RunFuzzer (x):
   pass
```

```
class demoCls (PY_LIB.CLS):
   def __init__(self):
      pass
   def CLS_func0(self):
      pass
   def CLS_func1(self, arg1):
      pass

def RunFuzzer (x):
   pass
```

Function structure

```
For intV in range (x, y):
      pass
```

```
For obj in iterObj:
      pass
```

```
if x:
   pass
```

```
if x:
   pass
Else:
   pass
```