

CnC-Python Find Primes Example

First, we need to define the CnC graph:

FindPrimes.cnc

```
// Declarations
// The tag values are the odd numbers in the range [3..n]
<string oddNums>;

// The prime numbers as identified by the compute step
[string->int primes];

// Step execution
// The compute step may produce a prime number (in the form of a tag instance)
(compute) -> [primes];

// Step prescription
// For each oddNums instance, there is
<oddNums> :: (python compute);

// Input from the environment: initialize all tags
env -> <oddNums>;

// Output to the environment is the collection of the prime numbers
[primes] -> env;
```

The CnC graph states that `oddNums` is a Tag Collection with strings as the tags. Currently, strings are the only supported tag types in CnC-Python.

Next, we define the Item Collection `primes` which has strings as tags and integer as its values. These types are defined in the format `tagType->itemType` before the name of the Item Collection.

We define the implementation language of the Step in the Step prescription as in `(python compute)`. In CnC-Python, `python` is the only allowable value. In future releases we plan to expan this value to include other languages such as C, C++, Fortran, Matlab, etc.

`env` is a special keyword representing the environment and in this example we define the environment will put string tags into the `oddNums` Tag Collection and read the results from the `primes` Item Collection.

After defining the CnC graph, we need to run this file using the CnC-Python translator using the following command:
`cnc_t FindPrimes.cnc`

Running the command will generate the following directories: `hj-cnc-api`, `sidl`, `java-client`, `py-lib`, `user-code`, and `hj-main`. In short, these are what the directories represent:

Directory Name	Function
hj-cnc-api	Wrapper classes for the HJ-CnC runtime used by Babel while generating SIDL server/client code
sidl	SIDL files used by the program. This includes the SIDL file for the runtime wrapper classes as well as the Step and Item Collections for the current program
java-client	The java client generated from the SIDL files used by the HJ program to call into the python implementation
py-lib	The generated python files that are invoked from the HJ program
user-code	This is the only directory the user should need to edit. It will contain template files for the python Steps as well as an application class to attach start and end event handlers. The even handlers are used to place values from the environment into Item Collections and read results back from Item Collections.
hj-main	The generated HJ files that manages code to make native invocations into the python implementations using the Babel runtime. The user launches the CnC-Python program using a generated script that invokes the generated HJ main class.

Back to the example, once the translator completes running there should be the following files in the `user-code` directory:

userFindPrimesApp.py.template

```
class Application:
    @staticmethod
    def onStart(args, oddNums):
        # TODO fill out the body of the function
        # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
        # e.g. operation on tag collections: aTagCollection.putTag(aTag)
        pass

    @staticmethod
    def onEnd(primes):
        # TODO fill out the body of the function
        # e.g. operation on input item collections: anItemCollection.get(aTag)
        # e.g. operation on input item collections: anItemCollection.printContents()
        pass
```

and

userComputeStep.py.template

```
class ComputeStep:
    @staticmethod
    def createAwaitsList(tupleContainer, tag ):
        # TODO fill out the body of the function
        # e.g. tupleContainer.add(itemCollection, tagValue)
        # e.g. operation on item collections: anItemCollection.get(aTag)
        pass

    @staticmethod
    def compute(tag , outPrimes):
        # TODO fill out the body of the function
        # e.g. operation on input item collections: anItemCollection.get(aTag)
        # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
        # e.g. operation on tag collections: aTagCollection.putTag(aTag)
        return True
```

Rename both files to `userFindPrimesApp.py` and `userComputeStep.py`. The `userFindPrimesApp.py` provides the `onStart` and `onEnd` functions. The function signatures are determined by detecting the environment interactions in the CnC graph. The `onStart` function also provides access to any command line arguments used while launching the program. The `userComputeStep.py` file provides the file the user needs to edit to provide the Step implementation. A Step needs to implement the `createAwaitsList` and `compute` functions. The `createAwaitsList` function allows the user to specify the input data dependences on Item Collections. Once these dependences have been satisfied the `compute` function will be invoked.

Below are simple implementations for the two python files:

userFindPrimesApp.py

```
import time

class Application:

    startTime = 0
    endTime = 0

    @staticmethod
    def onStart(args, oddNums):
        # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
        # e.g. operation on tag collections: aTagCollection.putTag(aTag)
        if len(args) > 0:
            firstArg = args[0]
            print("py: processing " + firstArg)
            intValue = int(firstArg)

            Application.startTime = time.clock()
            for i in xrange(3, intValue, 2):
                oddNums.putTag(str(i))
            else:
                print("py: usage FindPrimesMain <num_items>")

    @staticmethod
    def onEnd(primes):
        # e.g. operation on input item collections: anItemCollection.get(aTag)
        # e.g. operation on input item collections: anItemCollection.printContents()

        Application.endTime = time.clock()
        elapsedTime = int((Application.endTime - Application.startTime) * 1000)
        print "py: Elapsed time:", elapsedTime, "ms"
        primes.printContents()
```

and

userComputeStep.py

```
class ComputeStep:

    @staticmethod
    def createAwaitsList(tupleContainer, tag ):
        # e.g. tupleContainer.add(itemCollection, tagValue)
        # e.g. operation on item collections: anItemCollection.get(aTag)
        # no dependencies, do nothing
        pass

    @staticmethod
    def compute(tag , outPrimes):
        # e.g. operation on input item collections: anItemCollection.get(aTag)
        # e.g. operation on output item collections: anItemCollection.put(aTag, aValue)
        # e.g. operation on tag collections: aTagCollection.putTag(aTag)
        candidate = int(tag)
        if ComputeStep.isPrime(candidate):
            outPrimes.put(str(candidate), candidate)
        return True

    @staticmethod
    def isPrime(n):
        for k in xrange(3, n, 2):
            if n % k == 0:
                return False
        return True
```

Please refer to the [Partition-String example](#) to see an example of how to implement the `createAwaitsList` function.

Running this program with an input of 100 should produce the following output:

```
Running FindPrimesMain
Starting FindPrimesMain...
...
FindPrimesMain execution time: ... ms.
FindPrimesMain ends.
py: processing 100
py: Elapsed time: ... ms
Contents of py:FindPrimes.PrimesItemCollection [size=24]
'11' = 11
'13' = 13
'17' = 17
'19' = 19
'23' = 23
'29' = 29
'3' = 3
'31' = 31
'37' = 37
'41' = 41
'43' = 43
'47' = 47
'5' = 5
'53' = 53
'59' = 59
'61' = 61
'67' = 67
'7' = 7
'71' = 71
'73' = 73
'79' = 79
'83' = 83
'89' = 89
'97' = 97
```