# RTM Python Coding Style Guidelines

A basic coding style convention for the RTM scripts is desired to allow the various users of the RTM to produce python scripts to a common standard. This will improve the ability to communicate the model structure to new users and make it easier for existing users to review changes to the RTM procedures

## Basic Structure

The python procedures in the RTM should be formatted using 4-space indenting and Windows line endings (CRLF). A basic header at the top of each python script should identify it as part of the RTM and the toolbox path it can be accessed at. A basic description of the purpose of each script can also be provided.

Example Header:

##---------------------------------------------------------------------

##--TransLink Phase 2 Regional Transportation Model

##--

##--Path: translink.emme.stage1.step0.runall

##--Purpose: Full model run

##---------------------------------------------------------------------

It is desirable to keep line length below 80 characters, but it is recognized that some expressions will due to complexity have to be longer that this for readability. One specific case is the import of tools from the toolbox, the long toolbox names can be provided on the same line as the .tool() lookup rather than using a separate variable.

Example Tool Use:

util = \_m.Modeller().tool("translink.emme.util")

Example Tool Use with separate namespace variable (discouraged):

NAMESPACE = "inro.emme.matrix\_calculation.matrix\_calculator"

compute\_matrix = \_m.Modeller().tool(NAMESPACE)

## Reserved Variable Names

Each Tool should provide the \_m name as follows

import inro.modeller as \_m

Other reserved names:

eb – This variable or argument should always contain an emmebank object reference

util – This variable should always contain a reference to the translink.emme.util tool.

specs – A list of matrix calculation expressions being built to pass to a matrix calculator compute\_specs method.

Example using reserved names:

def method(self):

util = \_m.Modeller().tool("translink.emme.util")

eb = \_m.Modeller().emmebank

util.initmat(eb, “mo10”, “ones”, “A matrix containing 1 in all values”, 0)

specs = []

specs.append(util.matrix\_spec(“mo10”, “1”))

util.compute\_matrix(specs)

# Tool Methods

Each script should provide at least the following methods and should be provided in the following order, consult the EMME documentation for the details of what is available in these methods:

1. def page(self): – The UI page builder method
2. def run(self): – If this is an interactively runnable tool, pass arguments from the UI to the main \_\_call\_\_ method.
3. def \_\_call\_\_(self): – All functional code should be in, or called from this method

All methods should be instance methods (taking a self argument). All RTM tools should be imported using the Modeller().tool() method and not the direct .module() method.

Example of tool import:

util = \_m.Modeller().tool("translink.emme.util")

Example of module import (discouraged):

utilities = \_m.Modeller().module("translink.emme.stage3.step5.utilities")

## Page() Method

This method creates the graphical interface in Modeller. Each page method should start with this basic form:

def page(self):

pb = \_m.ToolPageBuilder(self)

pb.title = "Full Model Run"

pb.description = "Performs a full model run"

pb.branding\_text = "TransLink"

Optionally if the tool is not runnable directly, it can be explicitly noted:

pb.runnable = False

If the tool is marked not runnable in this way, no run() method is required. For example, the auto ownership module is not runnable outside of household segmentation and no run method is provided for that module.

## Run() Method

Any arguments set in the user interface should be processed and fed to the \_\_call\_\_ method. No other model code should appear in this method other than argument marshalling.

## \_\_call\_\_() Method

This method should always be tracing to the logbook to allow the model run process to be adequately traced. Tracing this method will capture calls directly from other python tools as well as interactive use in the Modeller intrerface. If the tool does receive arguments, consider saving them with the optional save\_arguments parameter.

@\_m.logbook\_trace("Read settings from settings.csv", save\_arguments=True)

def \_\_call\_\_(self, file\_name):

# Matrix Calculation Examples

A compute\_matrix method has been provided in the util tool that performs the matrix calculations using the parallel computation option using the level of parallelism defined in the settings file. The matrix calculations are more efficient when multiple calculations are passed in one call to the compute\_matrix method. This avoids the fixed overhead of initializing the matrix calculator each time. Each method doing matrix calculations should look like the following, importing the util tool at the start of the method and using the matrix calculator.

@\_m.logbook\_trace("Example Tool - Major Section")

def calculation\_example(self):

util = \_m.Modeller().tool("translink.emme.util")

The general guideline is to try and write ‘paragraphs’ of code beginning with a new list of matrix specifications in a ‘specs’ variable a middle section where specifications are appended to this list and finally a call to compute\_matrix().

specs = []

for i in range(1, 500):

specs.append(util.matrix\_spec("mo%d" % i, "mo500"))

… Perhaps many more lines appending matrix specifications

# Note that compute\_matrix appears at the same indent level as ‘specs’

util.compute\_matrix(specs)

# Start a new calculation - using matrix constraints

specs = []

for i in range(1, 10):

spec = util.matrix\_spec("mo%d" % i, "mo99")

# see the details of the constraints schema in the EMME documentation

spec["constraint"]["by\_value"] = {"od\_values": "mf158", "interval\_min": 0, "interval\_max": 0, "condition": "EXCLUDE"}

spec["constraint"]["by\_zone"] = {"origins": "GY1", "destinations": None}

# If performing a calculation that aggregates

spec["aggregation"] = {"origins": "+", "destinations": ".max."}

specs.append(spec)

util.compute\_matrix(specs)