Turtle Blocks Python Export

A Report on my GSoC Project

Marion Zepf

October 7, 2013

1 Motivation

Turtle Blocks teaches children an important skill in today's world: programming. Its block-based graphical interface makes abstract concepts like loops easy to understand and fun to play with. But it does not yet support the next step in learning: writing code in a 'real' programming language. My project fills this gap by automatically converting block programs to Python code. It enables the children to transfer their knowledge to a text-based language and to focus on acquiring the new syntax.

2 Project Description

Turtle Blocks Python export lets the user export their programming project from the Turtle Blocks activity to a Python script. The generated Python code can be run outside of Turtle Blocks, for example from the command line or in the Pippy IDE.

This tool is designed for users who are already proficient Turtle Block programmers and want to move on to text-based programming. It helps them transfer their knowledge and skills from block-based to text-based programming, as they can see their own creations in a new programming language. Thus, they can focus on the new language rather than the content of the program.

3 Implementation Approach

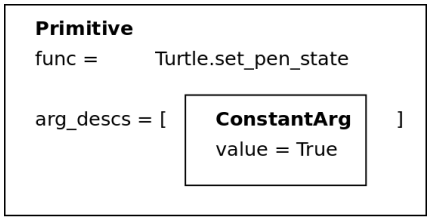
3.1 Execution and Export — Two Sides of the Same Coin

Each block is associated with information on how to execute it in TurtleArt (TA), and information on how to convert it to python code. These two pieces of information are packed together into a single object, an instance of the class `Primitive`. It holds a reference to the function or method to be called when the block is executed in TA.

Figure 1: The *pen down* block, the corresponding python code, and a simplified representation of its `Primitive` object.

a. 

b. `Turtle.set_pen_state(True)`

c. 

This facilitates exporting blocks that correspond directly to a python function. Examples include the *forward* block, which executes the method `forward` of the `Turtle` class, and the *int* block, which resembles the built-in function `int`. When these blocks are exported, the name of the associated function is extracted from the `Primitive` object and used in the python code.

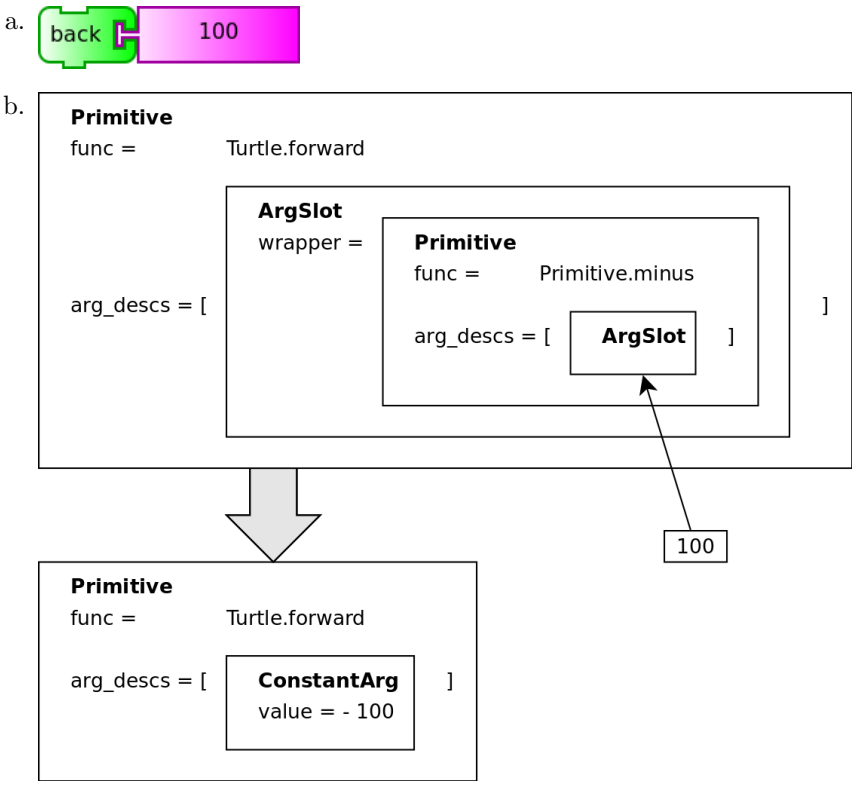
3.2 Constant Arguments

Some blocks, like the *pen down* block (fig. 1a), correspond to a function call with a specific list of arguments. The *pen down* block represents a call to the function `Turtle.set_pen_state` with the argument `True`, as shown in fig. 1b. This argument has to be stored with the function, but it must not be passed to it until the block is executed. This is why such arguments are stored in the `Primitive` object of the block (fig. 1c). From there, they can easily be retrieved and passed to the function during execution of the block, or converted to python code together with the function. Each such argument is encapsulated into a simple wrapper object of type `ConstantArg`.

3.3 Argument Slots and Wrappers

The *forward* block takes as its argument the distance by which the turtle should move. This corresponds directly to the argument to be passed to the function of its `Primitive`, to `Turtle.forward`. This block has one argument slot, and so has its `Primitive`. In addition to a function, a `Primitive` object also stores a list of arguments to be passed to the function. Some arguments are constant (see section 3.2), and some are placeholders for the argument blocks attached to the current block. The latter are represented by `ArgSlot` objects and their number must be equal to the number of argument docks of their 'parent block'. The function of the `Primitive` must accept the total number of all arguments

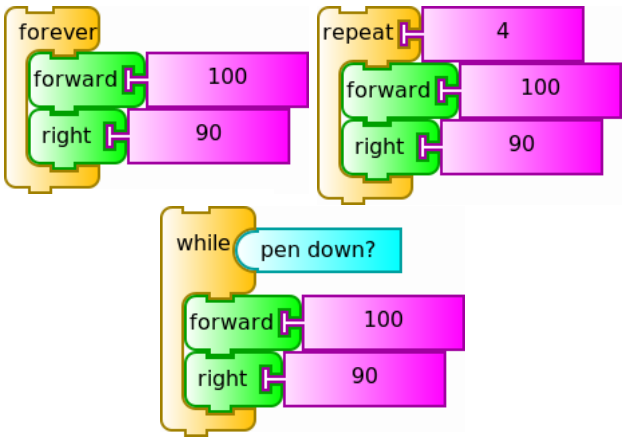
Figure 2: The *back* block and a simplified representation of its `Primitive` object before and after filling the argument slot.



(`ConstantArg`s and `ArgSlot`s). When the block is executed or converted to python code, the blocks attached to its argument docks are evaluated and their return values, wrapped into `ConstantArg`s, replace the `ArgSlot`s.

In other cases, the correspondence between `ArgSlot`s and docked argument blocks is not that simple. The *back* block (fig. 2a) uses the same function for its `Primitive` as the *forward* block, but inverts the sign of the numeric argument before passing it to `Turtle.forward`. The sign inversion step has to be executed after the argument block has been evaluated, but before its return value is passed to the function. It is 'wrapped around' the argument, so it is called a 'slot wrapper'. For these reasons, the callback for the sign inversion step is stored in the `ArgSlot` object under the `wrappper` attribute (see fig. 2b). A slot wrapper is executed (or converted to python code) just before the `ArgSlot` is replaced by a `ConstantArg`.

Figure 3: The *forever, repeat,* and *while* blocks.



3.4 Groups of Primitive Functions

Some blocks correspond to several functions, such as the *clean* block, which resets the plugins, clears the canvas, and stops all playing media, among other things. Such a list of functions to be called can be grouped together by the `Primitive.group` method, which calls all functions one by one. When it is exported, it is converted to several lines of python code, each holding one function call.

3.5 Export to Python Constructs

A number of blocks corresponds to pieces of python code that are not function calls. For instance, the *action* block that defines an action stack corresponds to a function definition in python. The *store in* block corresponds to an assignment statement. The `Primitive` objects of such blocks are handled specially when they are exported, i.e. there is a fixed correspondence between the function stored in the `Primitive` and the python code that it is converted to.

3.6 The Unity of Loops

All loops have in common that they have a body of statements (or blocks), which is executed zero or more times. How often it is executed is controlled by a loop controller that varies between different types of loops. For example, the controller for the *forever* loop block always allows another repetition (fig. 4a). The *repeat* loop controller allows only a fixed number of repetitions (fig. 4b), and the controllers of the *while* and *until* loop blocks allow further repetitions as long as a certain boolean expression does not change value (fig. 4c).

This unity of all loop types is captured by the function `LogoCode.loop`, which is used by the `Primitive` objects of all loops (fig. 5). It gets two argu-

Figure 4: Loop controller functions of the three loop blocks from fig. 3 in simplified form.

a.
```
def controller_forever():
    while True:
        yield True
```

b.
```
def controller_repeat(num):
    for i in range(num):
        yield True
    yield False
```

c.
```
def controller_while(condition):
    while condition():
        yield True
    yield False
```

ments, a loop controller in the form of a generator object or function, and the loop body as a list of primitive names to be resolved to `Primitive` objects later. All loop controllers, except for that of the *forever* block, require an argument which corresponds to the argument required by the block. This is why these loop controllers are attached as slot wrappers to the first `ArgSlot` of the loop block's `Primitive` object. The loop controller for the *forever* block does not require any arguments, and therefore it is attached as a `ConstantArg`.

3.7 Automatic Type Conversion

Most blocks accept only a certain type of arguments. E.g., the *minus* block only accepts two numbers, not strings. Even the *plus* block, which accepts any combination of numbers and strings, needs information on the type of its arguments to decide whether to add them mathematically or to concatenate them to a big string.

In addition, strings that represent a number must automatically be converted to the corresponding number, colors have an associated numeric value that can be used as a number, and even characters can function as the number representing their unicode value. These 'special' type conversions are already present in some blocks, but not used consistently across blocks.

The new type system addresses all these issues at once. It requires every `Primitive` object and every value of a value block to belong to a certain type. In the case of `Primitive`s, the type represents the return type of the `Primitive` object. This makes it possible to use the output of one block as the input of another and apply all necessary type conversion automatically.

The types form a network together with their converters. A converter is a function that converts a value from one type to another. The simplest example

Figure 5: Simplified representations of the `Primitive` objects of the three loop blocks from fig. 3.

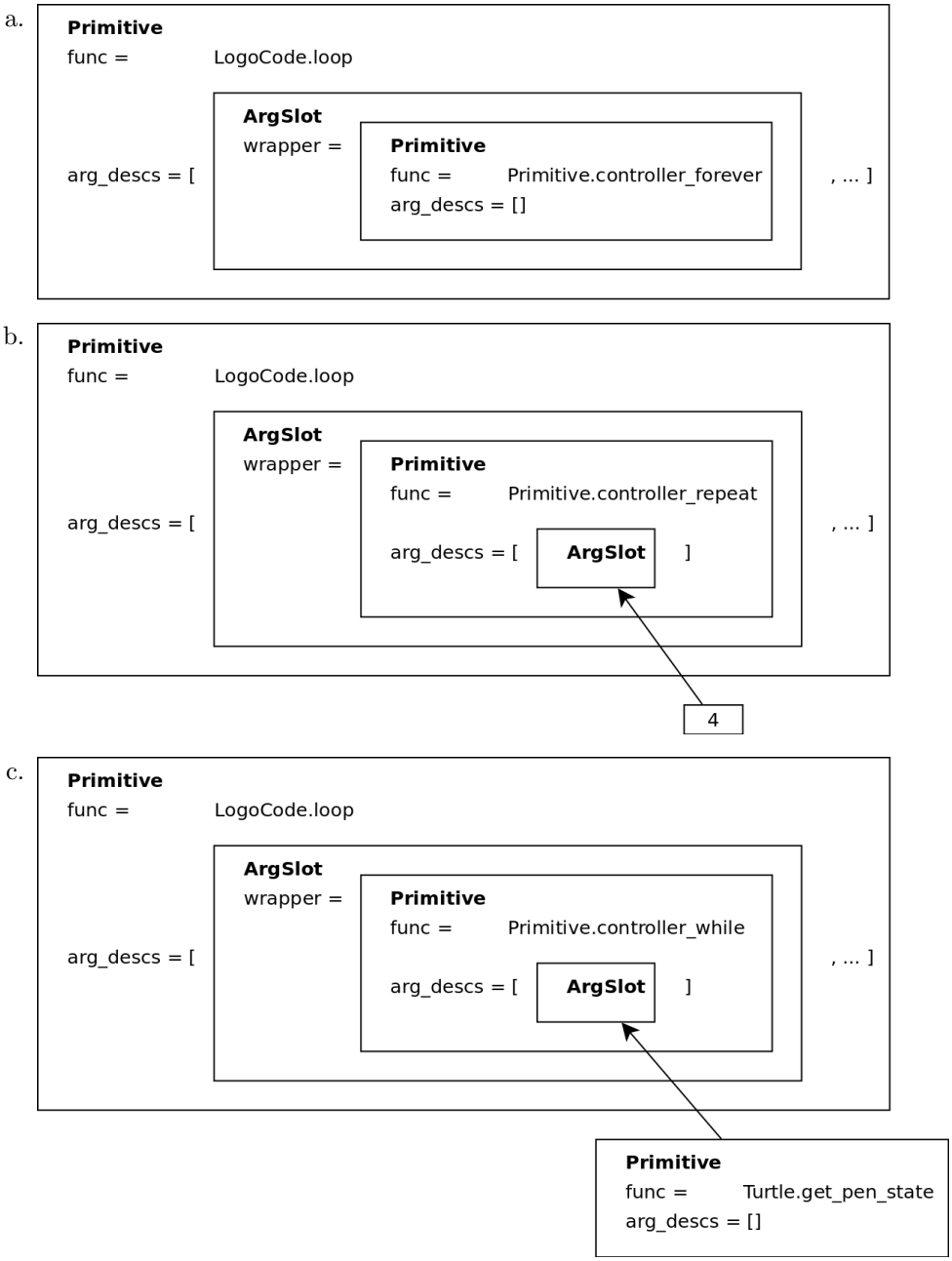
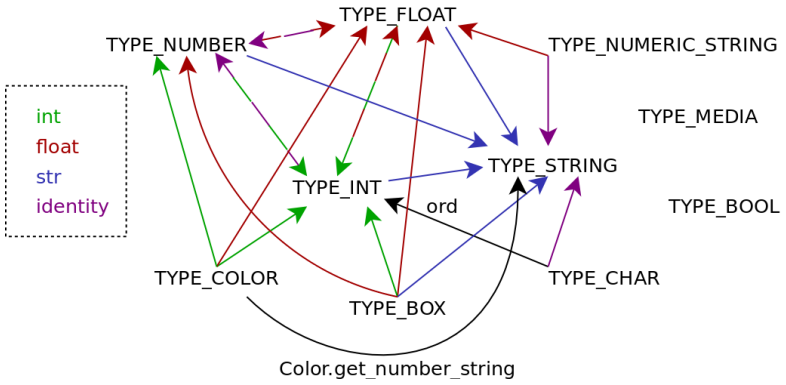


Figure 6: The type network. An arrow marks a converter from one type to another. The typical converter functions `int`, `float`, `str`, and the identity are indicated through color, and unusual converter functions are attached to the arrows. All types can also be converted to themselves or to the type `TYPE_OBJECT` using the identity.

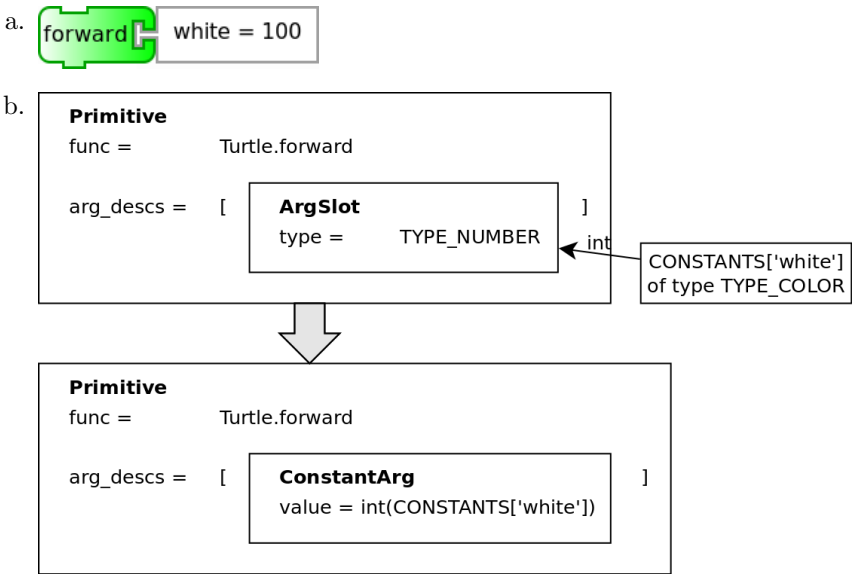


is the built-in python function `int`, which converts e.g., a numeric string like `"100"` to the integer `100`. The network of types and converters is stored as a two-level nested dictionary in the constant `TYPE_CONVERTERS` (in the module `tatype`). Its layout is simple — `TYPE_CONVERTERS[TYPE_A][TYPE_B]` holds the converter from `TYPE_A` to `TYPE_B`. A graphical representation of the type network is given in fig. (6).

When an `ArgSlot` is filled with an argument, TA automatically finds and applies the right converter based on the type the `ArgSlot` requires and the type of the filler. This way, attaching the color block *white* to the *forward* block (fig. 7a) has the same effect as attaching the *number* block with the value `100`. This is because the `ArgSlot` of the *forward* block requires type `TYPE_NUMBER`, and the value of the *white* block is of type `TYPE_COLOR` (fig. 7b). The converter function from `TYPE_NUMBER` to `TYPE_COLOR` is the python built-in function `int`, which, when applied to a `Color` object, yields its associated numeric value.

The type system is the same for execution and export, thereby ensuring that the exported python code behaves exactly like the original block program. When a block with an arugment is exported, the `ArgSlot`'s type requirement is compared to the type of the argument, and the appropriate type converter is wrapped around the argument.

Figure 7: The *forward* block with a color argument and a simplified representation of its `Primitive` object before and after filling the argument slot.



4 Difficulties and their Solution

4.1 Execution of *While* and *Until* Loops

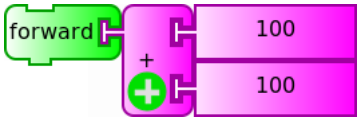
The *while* and *until* loop blocks are different from the other loops in that they require the boolean expression to be re-evaluated before every iteration step. This is not required for other loop types because the *forever* loop's controller takes no argument, and in the case of the *repeat* loop, it is sufficient to evaluate the numeric argument once before the loop starts. Its value can be saved inside the generator that allows or disallows further iterations.

This solution is not applicable to the *while* and *until* loops because each iteration step has the potential to change the value of the boolean expression. Therefore, the boolean expression has to be stored in a way that makes it possible to re-evaluate it. This is achieved by passing arguments to their 'parent' `Primitive` not as the result of evaluating them, but as `Primitive`s themselves. The 'parent' `Primitive` then decides whether to call its argument immediately, or to store it.

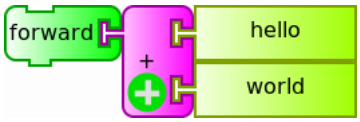
The `Primitive`s of the *while* and *until* loops do not call their first argument, but pass it on to the loop controller. The controller calls the argument before each iteration step to determine whether to execute the step or to break out of the loop.

Figure 8: The *plus* block with two numeric or two string arugments and a simplified representation of its `Primitive` object.

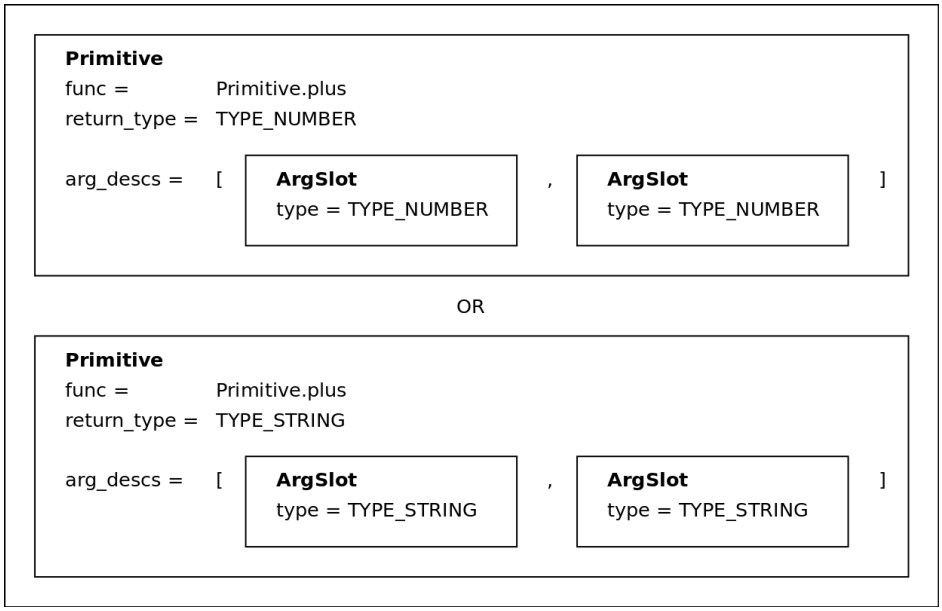
a.



b.



c.



4.2 Disjunctions

The *plus* block does two different things — mathematical addition or string concatenation — based on the types of its arguments. If both arguments are numbers, they are added up (fig. 8a), otherwise both arguments are converted to strings and concatenated (fig. 8b). This makes it necessary to define two different `Primitive` objects for the *plus* block (fig. 8c), each with different type requirements in its `ArgSlots`.

The *set color* block always uses the same function to set the pen color, but it accepts both color value blocks and numeric values. It is important to note that a color argument must not be converted to a number before being passed to the function `Turtle.set_pen_color` (the primitive function of the *set color*

block), since that would not update the pen shade and pen gray values to the color's shade and gray values. So, it is not possible to rely on automatic type conversion from `TYPE_COLOR` to `TYPE_NUMBER` in this case. Instead, the *set color* block's `Primitive` must be able to accept either a value of type `TYPE_COLOR` or one of type `TYPE_NUMBER`.

These are only two examples that show the necessity of disjunctions in the system of `Primitives` and types. In order to make disjunctions easier to instantiate for block inventors, the utility function `or_` (from the module `taprimitive`) is provided. It takes all disjuncts of a disjunction as its arguments and returns an object of the appropriate type that represents a disjunction of several types, `ArgSlot`s, lists of arguments, or entire `Primitive` objects.

During execution or export, the disjunctions are tried in order, and the first disjunction that leads to matching types is chosen. Two types are considered to match for this purpose if there is a converter function registered for them.

4.3 What is the Type of Boxes with Unknown Content?

At the time of execution of a block program, values can be stored in boxes or retrieved from them. The retrieved values can also serve as arguments to other blocks as their type can be identified at runtime. This is not the case during export. It is impossible to predict what type of value a box will hold at the time the exported code is executed, as the value and its type may depend on user input or be subject to chance. Yet the exported code must behave the same way as the block program.

The solution to this problem is the special type `TYPE_BOX`. It is the return type of the *box* block, as well as of several user input blocks. During export, this type is handled specially by the type system. Converters are not applied directly to value of `TYPE_BOX`, but the type system's utility function `convert` (from the module `tatype`) is used instead. It accepts a value and a target type, and converts the value to the given target type. E.g., if the value of a box must be converted to an integer, the resulting python code is not `int(BOX['my box'])`, but rather `convert(BOX['my box'], TYPE_INT)`.

5 Developer Documentation

Two tutorials explain how to define a new block and a new type in the type system, respectively. They are intended for TurtleArt developers and plugin writers who wish to add new blocks to TurtleArt or to make existing blocks exportable to Python code. They are available as markdown files under `doc/` in the root directory of the Turtle Art repository.

A large number of documentation strings and comments in the code explain the purpose of classes, methods, attributes, and constants. As part of this project, I also added in-code documentation to classes and methods which are not directly related to the Python export tool, such as the `Block` class. Such

documentation is useful for other TurtleArt developers, especially new developers who want to get an overview of the existing code base.

6 Achievements

As a result of this project, TurtleArt users can export their block programs to python code. The export functionality is restricted to those blocks with a `Primitive` object — yet this includes more than half of all blocks. The blocks of the *Turtle, Pen, Pen Colors, Numeric Operators, Flow Operators, and Variable Blocks* palettes are exportable, and so are the most important blocks from some other palettes, like e.g., the *show* block.

The implementation of the export functionality is modular and documented extensively, so that other developers can continue to make the rest of the blocks exportable as well. In addition, inventors of new blocks can use the system of `Primitive`s and types to implement their blocks and make them exportable at the same time.

7 References and Further Reading

- Git repository for development of the Python export tool: `https://git.sugarlabs.org/~mzepf/gsoc-python-export`
  The most recent version of the code is in the branch `type-system`.
- Git repository for design documents and sample code: `https://github.com/outofthecave/ta-python-export-dev`
- Project wiki page: `http://wiki.sugarlabs.org/go/Summer_of_Code/2013/Turtle_Blocks_Python_export_project`
- Project proposal wiki page: `http://wiki.sugarlabs.org/go/Summer_of_Code/2013/Turtle_Blocks_Python_export`
- Google Melange page: `http://www.google-melange.com/gsoc/project/google/gsoc2013/mzepf/9001`