How to implement a new problem type (How2TPL)

<Live Google Doc>

In this document, we explain the process for creating a TPL file in order to implement a new problem type. **This is the ideal first stop** for those looking to expand Practicum either on a small scale (e.g., for a particular classroom), or on a larger scale (e.g., a comps group). For the former group, **you may choose to skip to steps relevant to you**; for example, to only 4.2 and 4.3 if you just want new or edited problems. For the latter, we recommend running through this process as a tutorial, but **look to Step 2 to understand the front-end** as it connects to the TPL, **and to Step 3 for the Python back-end**.

Table of Contents

1. Writing out the Thought Process

1.1. Clarifying your ‘problem type’

- **Ask** yourself:
    - Does it have a **clear and consistent thought process**? What is it?
    - Do **intro/returning CS students struggle with its application**? Is it worth putting in Practicum?
    - Can you **vary and iterate on this problem type**? How broad is it?
    - **Red flag** — Is this **a feature or a type of problem**?
        - For example, break statements are available in Practicum, but do not receive their own set of problems
        - Features can be **implemented independently in Step 3**
- Devise some **example problems** that you would include
    - Consider what problems **will and will not be included**
        - The **narrower** you get, the **easier** it will be to implement, but it may be **less useful**
        - For example, *For Loop Investigations* currently **support range, string and list iterables**, but **do not support dict iterables**
    - **Save these** for later

1.2. Structuring the thought process

- Write out **all the scenarios a user might face** when solving these problems
    - **Refer to your examples** from the previous step
    - For example, in *For Loop Investigations*, we **assume that only variable declarations**, not, say, an if statement, **will appear before the for loop**
        - This **simplifies the thought process algorithm** (TPA; the `.tpl.txt` file) by reducing the number and types of checks we have to do before entering the for loop
    - **Make separate cases** for branching logic, such as whether an if statement is true or false or whether a loop has more iterations or not
        - You will need to be prepared for **every possible type of line and line outcome a user can experience** at each stage of the problem

1.3. Filling in the prompts and supporting actions

- Write out the **prompt text (in plain English) for the scenarios** in the previous step
    - Think through how you want to **explain the problem walkthrough to the user**
    - Every case will need some explanation for what is going on
- Note at each stage of the process if any **highlighting, variable bank edits, user questions, or any other extra actions** will need to be taken
    - For example, in if statements of the *If/Else Investigations*, note that **the conditional is highlighted** and the **user is presented with a yes/no question** they need to answer

2. Converting into TPL Code

2.1. Structuring the TPL file

- **Begin writing the TPL code** into a .txt file within Practicum, **using an existing file such as [forLoop.tpl.txt] as reference**
    - TPL files start by **creating the AST node, laying out the variables, and creating the variable bank**, given below
        - ```
          let ast;
          [no_step]
          ast = state.ast;
          ```
        - ```
          [no_step]
          state.vars = helper.copy_args(state.args);
          ```
        - ```
          let variables: VariableBank;
          [no_step]
          variables = helper.create_new_variable_bank();
          ```
    - Next, there should probably be some **introductory text and the addition of the parameters to the variable bank** before the TPL begins reading in and processing new lines
    - Depending on your problem type, copying the overarching for or while loop structures from their respective TPLs may save a good amount of time
- No matter what the structure of your problem is, you will almost certainly have **one or more loops based on reading in lines of the problem** and reacting to their type
          ```

- The TPL parser can parse **while, do/while, and if(/else) statements**
    - If more complicated structures are necessary you may find it necessary to make additions to the TPL parser and simulator
- Look at the existing examples to **see how setting up loops and loop variables works** with the AST node
- The **following helper functions (in tplHelper.js) will likely be important** for the structure
    - `get_the_next_line_in_this_block_to_execute`: **For some parent statement, retrieves the next line in its child block**, or null if none; supports method (the overarching AST node in present problem types), for loop, while loop, and if/else types
    - `is_there_another_line_in_this_block_to_execute`: Boolean of the above, for conditional checking

2.2. Implementing prompts in TPL code

- With the exception of let lines (initial variable declaration), **every contentful line of the TPL code by default produces a prompt**
    - This can be **suppressed with the `[no_step]` tag**
    - **Variable assignments will use the variable's name** (in snake case) for the prompt
        - For example, **`update_the_list_element`** `= helper.assign_the_new_value_to_the_list_element(...)` will provide the user with the prompt **"Update the list element."**
    - Non-assigning **function calls (i.e., as conditionals of if statements or loops) will use the function name** (in snake case) for the prompt
        - For example, `do {...} while (helper.`**`is_there_another_item_in_the_loop_sequence(...)`**`)` will provide the user with the prompt **"Is there another item in the loop sequence?"**
- You can also **use the `[prompt]` tag to provide a string (of English text) as a prompt directly**
    - The tag and string should be **placed after the related action**
    - Usually, you should **use the `[no_step]` tag to suppress the automatic prompt for the related action**

- Use a combination of `[prompt]` tags and automatic generation to **include your plain-English prompts from Step 1**
    - Generally, **shorter prompts, especially repetitive ones, should use the automatic generation**; longer or one-time prompts should use `[prompt]` and `[no_step]`
    - Reference existing examples, such as forLoop.tpl.txt, as a guide

2.3. Applying interactivity via TPL types and tags

- The **object type you declare the variable as (`let`) also influences its visuals** and interactivity; **use these as appropriate to match the actions in your outline** from Step 1
    - **`Parameter`**: Highlights the **function arguments**
    - **`Variable`**: Visualized in the variable bank, used for **variables in the problem code**
        - Interactive version: the user adds the values of the variables themselves
    - **`ArrayElement`**: Highlights **the array element at a specific index**
        - Interactive version: the user clicks the element themselves
    - **`AstNode`**: Highlights the AST node, which can be **a loop call, an if statement, etc.**
        - This highlighting is a lighter blue color and is more explicit than the regular line highlighting
    - **`Line`**: Highlights **the specific line of the code**
        - In interactive mode, the user clicks on the line themselves
    - `ArrayIndex`: For back-end use in the scratch area
    - `ArrayIndices`: For back-end use in the scratch area
    - `ScratchAstNode`: For back-end use in the scratch area
- The **class of interactive TPL tags**, below, enables interactivity in the back-end; **correlate these with the object type for the effect you want**
    - `[interactive(“`**`next_line`**`”)]`: before a **variable assignment; when moving to a new line**
        - Expects a **`Line`**

- `[interactive("`**`add_variable`**`")]`: before a **variable assignment; when adding an item to the variable bank**
  - Expects a **`Variable`**
- `[interactive("`**`update_variable`**`")]`: before a **variable assignment; when updating the value of an item in the variable bank**
  - Expects a **`Variable`**
- `[interactive("`**`conditional`**`")]`: before an **if(/else) statement; when evaluating the conditional of an if(/else) statement**
  - Expects an **`AstNode`**
- `[interactive("`**`list_element_click`**`")]`: before a **variable assignment; when updating the value of an element of a list in the variable bank**
  - Expects an **`ArrayElement`**
- `[interactive("add_array_index")]`: For back-end use in the scratch area
- `[interactive("array_element_get")]`: For back-end use in the scratch area
- `[interactive("evaluate_expression")]`: For back-end use in the scratch area

2.4. Using helper functions

- The `helper.` functions in the TPL refer to `tplHelper.js`; **use these as appropriate to get the data you need from the (Python) AST**
  - This can range from simple checking of AST tags, as in the function `is_if`, to doing Python simulation work, as in `execute_the_loop_increment`
  - See Appendix A for a **list of useful, currently existing helper functions**
  - See Step 3.1 for **instructions on creating new helper functions**

3. Implementing New Features *(if necessary)*

3.1. Creating new helper functions

- You may find that you **need the TPL to be able to manipulate or fetch data from the Python AST** in a new or different way; this can be achieved by **adding new functions to `tplHelper.js`**
  - Functions in `tplHelper.js` should have **descriptive names in snake case** for the automatic prompt conversion mentioned in Step 2

  - Helper functions can take **any number and type of parameters**, but **do not support default values**
    - You can, however, have a parameter be last in the order, such that it defaults to `undefined` when undeclared in the TPL
  - `tplHelper.js` is one of the best places to **use `console.log`s or inspect-tool breakpoints** for bug fixing
- It may at some point become relevant to **understand the object structure of the AST node(s)** you are working with
  - Here is an **overview of the elements** of a node
    - `id`: A unique identifier produced by `new_id()`; can be useful during bug fixing
    - `position`: The position of the statement in the code; determines highlighting and can help during bug fixing
    - `tag`: Almost always present, and **declares the type of node**; if absent, assumed to be `"literal"`
    - `type`: Present only and always for `"literal"`s, **declares the type of Python object** associated with the node
    - `value`: For `"literal"`s, **contains the literal value**
    - `expression`: **Contains the dependent expression node** for certain types of nodes, such as declarations
    - `args`: **Gives the arguments, as a JavaScript array of nodes**, of the expression
    - `body`: **Contains, as a JavaScript array of nodes, the lines in the block** parented by the statement
    - *others*: See below
  - The object declarations, in full, can largely be found in **`python/parser.js`**[1]
  - This is another place where you may find it easiest to **use `console.log`s or inspect-tool breakpoints**

[1] Not to be confused with `tpl/parser.js`.

3.2. Adding Python parser/simulator functionality

- While you are creating new helper functions, you may find that you **need the back-end to process new types of Python statements or expressions**; this will require **work in `python/parser.js`, `python/simulator.js`, and/or `python/ast.js`**
  - **`python/parser.js`** contains the **core of Python functionality**; you will **definitely need to do some work here**
  - **`python/simulator.js`** handles **automated evaluation of (only) expressions**; bear in mind that most Python simulation happens in real-time with the TPL code
  - **`python/ast.js`** handles the **treeing of nodes**; you will need to work here for **statements with a `body` or `expression` element**
- For **new types of statements**, starting in `python/parser.js`:
  - **Add any new Python keywords** to the `keywords` dictionary
  - In the `match_statement(...)` function, **add a case for the new statement** using the initial keyword
  - Modeling off of `match_forloop`, `match_ifelse`, etc., **add a `match_` function that handles the content of the statement and returns a node object**
    - Note that `match_block` is used to **match a child code block**, if needed
  - If the statement has a body (i.e., if you used `match_block`), **add a case to `children_of` in `python/ast.js`**

4. Adding the Problems to Practicum

4.1. Integrating the new TPL file

- **Add the new problem type** (whose name must correspond to the TPL file; e.g., "`nested`" for `nested.tpl.txt`) **to `index.js`**
  - First, add it to **both elements of the `problem_types` dictionary**
  - Second, add it to **the `numProblemsByCategory` and `problemIdsByCategory` dictionaries** (each of which appears twice) **in `setupLogging(...)`**

4.2. Writing the Python problems

- Following the model you declared in 1.1, **write out a set of Python programs** in your text editor or IDE of choice

  - **Use tabs instead of spaces** for indentation: one tab per indent
  - Remember **not to deviate significantly from your set structure**, or you may need to add more to parse other cases
  - It is **simple to go back and edit or add to these problems**, as you see fit to best teach students

4.3. Integrating the problems as a set

- **Add a new problem set to `categoryConfig.json`** following the model of other problem sets
  - For the `name` element, **prefix `"default-"` onto the problem type name** from the previous step
  - For the `category` element, **use the problem type name** from the previous step
  - **Problem IDs need to be unique** but it otherwise does not matter what you set them to; we just numbered them
  - For each problem, **copy the problem text from the original into the JSON file**
    - If you are using WebStorm (or most other IDEs that handle JSONs), **newlines and tabs should automatically get replaced appropriately** (make sure, per the last step, that all indentation is done with tabs)
    - **Remove the final newline** from the problem text string, if it copied in
  - Under variants, **every problem needs at least one set of arguments** as a base; more are optional

A. Currently Supported Python Features

A.1. For loops

- Can iterate over a **string, list, or range call** with 1-3 integers
- For loops that run for **less than two iterations are not currently supported**
- See the **for and nested for loop TPLs** for implementation

A.2. While loops

- Currently supports simple arithmetic conditions for variables: `<`, `>`, `==`, `!=`
- See the **while loop TPL** for implementation

A.3. If statements

- The TPL also **supports elif and else statements**
- While multiple elifs are possible, they require **further nesting (hardcoding) in the TPL**
- See the **if/else TPL** for implementation

A.4. Nested for loops

- As with elifs, multiple nested for loops require **further nesting (hardcoding) in the TPL**
- See the **nested for loop TPL** for implementation

A.5. Break and continue

- Search in the **for, nested for, or while loop TPLs** for their respective implementations

A.6. Range and len

- Evaluated in the the Python simulator

A.7. Variable assignment

- Practicum supports **assigning new or existing variables to integers, floats, or simple equations** using +, -, *, /, %, and their corresponding += style operators
- Search **`add_variable` or `update_variable` in any TPL** for the implementation

A.8. Updating lists

- Any list in the variable bank can have its value updated
- Search **`does_this_line_update_array` in the while loop TPL and check the accompanying else block** for the implementation

A.9. And/or

- Evaluated in the Python simulator

B. Useful Helper Functions

B.1. Adding to the variable bank

- `helper.add_other_parameters_to_the_variable_bank`: **adds the function parameters** at the start of the problem
- `helper.add_local_variable`: **adds a newly created variable** to the variable bank
- `helper.add_the_loop_array_to_the_variable_bank`: adds the loop sequence for a **for loop iterating over a list or string**
- `helper.add_this_to_the_variable_bank`: adds **the loop variable for any kind of loop**; also adds the loop sequence for a **for loop iterating over an integer range**

B.2. Getting the next line

- `helper.get_next_line`: used **before entering the main loop of the TPL** to grab the next line of the problem, **only for use on the initial variable declarations** before any for/while/if lines or similarly simple situations
- `helper.get_the_next_line_in_this_block_to_execute`: used **within the main loops of the TPLs to grab the next line of the problem to execute**, capable of more complex operations such as skipping else statements or returning to the beginning of a loop

B.3. Handling loops

- `helper.get_loop`: takes the ast as its parameter and returns **the loop line of the problem**
- `helper.is_loop_called_without_range`: returns **true if the for loop iterates over a list or string**; false if it iterates over an integer range
- `helper.loop_sequence_index`: returns **the index of the loop sequence that the for loop is currently on**, typically right after incrementing the for loop
- `helper.execute_the_loop_increment`: returns **the value at the index of the loop sequence that the for loop is currently on**
- `helper.get_loop_init_variable`: returns the **iterating variable a for loop is assigning into**
- `helper.get_iterable_sequence`: returns a **list containing the loop sequence** of a for loop

B.4. Updating lists

- `helper.does_this_line_update_list`: returns **true if the line updates an element of a list** but false if, for example, it updates the value of a standalone integer
- `helper.select_the_list`: returns which **array in the variable bank is having a value updated** by the line being parsed
- `helper.select_the_index`: returns **the index of the array having its value updated** by the line being parsed