Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Technology

InPUTpy
InPUT for Python

Christoffer Fink
MID SWEDEN UNIVERSITY
Department of Information Technology and Media

Examiner: Ulf Jennehag, ulf.jennehag@miun.se
Supervisor: Felix Dobslaw, felix.doblaw@miun.se
Author: Christoffer Fink, crfi0400@student.miun.se
Degree programme: Programvaruteknik, 180 credits
Main field of study: Computer Engineering
Semester, year: HT, 2013
Abstract

There are many problems where the solution depends heavily on parameter tuning and therefore on configuration. Examples include Hill Climbers and various Evolutionary Algorithms. The Intelligent Parameter Utilization Tool (InPUT) uses a cross-language format for describing configurations of computer experiments, thereby aiding in documentation and communication. It is implemented in Java, and a C++ version is being developed. This means that only implementations in statically typed languages currently exist. A Python port would thus greatly increase the diversity of implementation languages. The goal of this project was to create an InPUT implementation in Python, and to explore the suitability of a dynamically typed language in this context. A de facto specification was discovered by creating a suite of learning tests, and the new implementation was developed using a test-driven approach. Several features of Python proved advantageous by simplifying the development process.

Keywords: InPUT, Python, dynamic typing, functional programming.
Terminology

Parameter

A named set of possible values. May either refer to a parameter in general or a Param instance. When the distinction is important, ”Param” will be used.

Param

The representation of a parameter as an object.

NParam

A numeric parameter (integer, floating point and boolean values).

SParam

A structural parameter (arbitrary user-defined types).

InPUT

The language-independent configuration framework.

InPUT4j

The Java implementation of InPUT.

InPUTpy

The Python implementation of InPUT.
## Contents

1 Introduction ................................................. 4  
   1.1 Background and Problem Motivation ....................... 4  
      1.1.1 Intelligent Parameter Utilization Tool ............... 5  
      1.1.2 An InPUT Implementation in Python ................... 5  
   1.2 Overall Aim .......................................... 6  
   1.3 Scope ................................................ 6  
   1.4 Concrete Goals ...................................... 6  

2 Theory .................................................. 7  
   2.1 InPUT ................................................. 7  
   2.2 Dynamic and Static Typing .............................. 8  

3 Methodology .............................................. 13  
   3.1 Porting InPUT4j ....................................... 13  
   3.2 Identifying Useful Language Features .................... 14  

4 Design and Implementation .................................. 15  
   4.1 Design ............................................... 15  
   4.2 Implementation ...................................... 18  
      4.2.1 eval ............................................. 18  
      4.2.2 Factories and kwargs .............................. 20  
      4.2.3 Runtime Type Loading .............................. 21  
      4.2.4 Meta Parameters .................................. 22
## 5 Results

### 6 Discussion

- 6.1 Open Questions ........................................ 28
  - 6.1.1 Numeric Types in Java and Python ............... 28
  - 6.1.2 Enums in Python .................................. 28
  - 6.1.3 Type Checking in InPUTpy .................... 29
- 6.2 Justifying DocTest .................................. 29
- 6.3 Ethical Implications ................................ 31
- 6.4 Conclusion ............................................. 32
- 6.5 Future Work .......................................... 32
  - 6.5.1 InPUT Specification ............................ 32
  - 6.5.2 InPUTpy ........................................ 32
  - 6.5.3 InPUThs .......................................... 33

### A InPUTpy github repository 


3
Chapter 1

Introduction

1.1 Background and Problem Motivation

There are many problems for which an optimal and/or complete algorithm is unknown. Examples include various constraint satisfaction problems and different classes of optimization problems. There also exist many techniques for solving those problems approximately, within some resource limit. Such techniques include Particle Swarm Optimization, Simulated Annealing, and various Evolutionary Algorithms, such as Genetic Algorithms and Differential Evolution[1]. All of these techniques perform a form of parameter tuning in order to achieve some level of optimization. By extension, this means that the process of generating the solution, as well as the solution itself, relies heavily on configuration. A key part of the process is trying a large number of configurations semi-randomly. Some heuristic is then used to determine what constitutes a good configuration, and an algorithm determines how to generate new candidates. The result is a configuration that (hopefully) solves the given problem within a defined set of constraints.
1.1.1 Intelligent Parameter Utilization Tool

The Intelligent Parameter Utilization Tool (InPUT)[2, 3] enables researchers to design and document computer experiments in a programming language independent way. Configurations define design spaces in a declarative style, and generated designs can be imported and exported in various formats.

The only implementation of InPUT that is currently available is written in Java (InPUT4j), and an implementation in C++ is being developed. However, these two languages are quite similar in many ways. In particular, they are both statically typed. As a result, InPUT is currently limited to only two implementations and a single paradigm. InPUT would be more useful if it were available to a more diverse set of users. The narrow range of implementation languages also leaves open the question of how well a dynamically typed language would be suited to the task.

1.1.2 An InPUT Implementation in Python

Python would be an appropriate language with which to create another implementation of InPUT. It is a popular language[4, 5] that is dynamically typed and supports a range of programming styles:

- Plain imperative programming
- Object-Oriented programming (OO)
- Functional programming (FP)

In addition, Python also has extensive support for meta programming. The dynamic typing and support for multiple programming paradigms makes Python a suitable choice for a new InPUT implementation.

It is expected that a dynamic language such as Python will simplify the design and implementation, by offering greater flexibility[6, 7]. After all, "Life in a dynamically typed world is fundamentally simpler"[8].
1.2 Overall Aim

The goal is to port InPUT4j to Python, and to investigate where Python, as a dynamic language, offers advantages when creating the design and implementation.

1.3 Scope

The implementation is limited to a subset of the core features of InPUT4j. Evolutionary algorithms and other optimization problems are merely example application domains and will not be explored.

1.4 Concrete Goals

The ported program should implement the core functionality of the original Java version, which means that it should pass the existing tests [9]. Additionally, the process of developing the clone should yield specific examples of features of dynamic languages in general, or of Python in particular, that made development easier.
Chapter 2

Theory

2.1 InPUT

At its core, InPUT is about generating designs from design spaces by initializing parameters based on implementation language agnostic configurations.

A design consists of a set of names mapped to values. Designs are created by initializing parameters according to their definitions. A design space represents all the different possible combinations of parameter values, and hence all the possible designs for that set of parameters.

There are two primary kinds of parameters: numeric parameters (NParam) and structural parameters (SParam). NParams are analogous to primitive data types in some programming languages. They represent various numeric types as well as boolean values. NParam elements cannot be nested inside each other in the configuration, but they can depend on each other by defining their intervals relative to other NParams. Continuing the analogy with programming languages, SParams are objects. They can represent values of an arbitrary user-defined type and can contain other parameters nested in them. Nested parameters correspond loosely to instance variables.

When SParams are initialized, the program must create an instance of a
previously unknown type. Information about the type of the parameter is specified in a separate code mapping configuration.

There are also choice parameters (SChoice), which effectively represent different alternatives for how to initialize an SParam. That is, when their parent SParam is initialized, the details of what kind of value will be created can be randomized. An SChoice can thus be viewed as a potential SParam.

### 2.2 Dynamic and Static Typing

With dynamic typing, the type of an object is associated with that object. Only certain operations are supported by the object, depending on its type. Because the type information resides with the object, and not with the name bound to that object, a variable could potentially be of any type. Type checking can then only be performed at runtime (code example 1).
```python
def f(someVar):
    print(3 + someVar)
f(4)    # Prints 7
f('4')  # Error
```

**Code example 1:** With dynamic typing, any error shows up at runtime.

In Lua, the second example would also work, but an argument like 'hello' would fail. The name someVar does not have an associated type, so any object can be passed in as an argument to the function, and depending on the type of that object, the operation may or may not succeed. The looming threat introduced by this approach is that errors such as the third case are undetectable until runtime (code example 2).

```python
if wantToSucceed:
    # The program will work fine
    # as long as wantToSucceed is true.
f(4)
else:
    # Otherwise, the program is guaranteed to fail.
f('hello')
```

**Code example 2:** This program may or may not work, depending on the execution path.

A statically typed language would require that a type be declared for someVar, and that any arguments to the function f have a compatible type. For example, a C++ compiler would refuse to even compile a program like the one in code example 3.

Many dynamic languages employ a concept called duck typing. This technique is based on the familiar saying: "if it walks like a duck and quacks like a duck, then it’s probably a duck." It allows any object that supports an operation to be used as an operand in that operation. In other words, it
```c
void f(int someVar) {
    printf("%d", 3 + someVar);
}
```

```c
if (wantToSucceed)
    f(4);
else
    f("hello");
```

**Code example 3:** The type error shows up during compile time. Even though there is no guarantee that the offending statement will be executed, it doesn’t matter whether it in fact is a duck. As long as it can do the things a duck is expected to be able to do, it can be treated like a duck.

With duck typing, any inheritance hierarchy can be ignored at will. The JavaScript snippet in code example 4 demonstrates the principle more clearly. In fact, the dynamic nature of this way of handling types can be made even clearer by making only a slight alteration to the previous example (code example 5).

The opName could even be input by the user. A trade-off exists between finding errors at compile time and having flexibility in solving a problem.
// The function expects some object
// that can be accelerated.
function accelerate(obj, value) {
    obj.accelerate(value);
}

obj = {}  
// This will fail, since obj
// doesn’t support the operation.
accelerate(obj, 13);

obj.accelerate = function(value) {
    alert('accelerated by ' + value);
}
// This will work
// because obj now supports the operation.
accelerate(obj, 13);

Code example 4: Support for a new operation can be added at run-time.
// The same accelerate function as in
// the previous example is assumed to exist.
function f(value) {
    alert('accelerated by ' + value);
}

if(wantToSucceed)
    var opName = 'accelerate';
else
    var opName = 'some other operation';

obj[opName] = f;
// May or may not work, depending
// on the value of wantToSucceed.
accelerate(obj, 13);

**Code example 5:** Adding a method to an object under one of two possible names.
Chapter 3

Methodology

3.1 Porting InPUT4j

InPUTpy was developed using Test-Driven Development (TDD)\cite{10}. The first step in the development process was porting the existing tests. However, the tests were exercising very high-level functionality. So the existing tests had to be put to one side, and lower-level tests had to be written in accordance with TDD practices.

Before implementing any substantial parts of the program, DocTest\cite{11} was created as a new sub-project of InPUT4j. This project served as learning tests\cite{10}. The tests were used to discover details about the behavior of InPUT4j, and thus formed a kind of specification.

A set of testing tools was also developed. The InPUT4j test project includes some classes that are used in the test configuration. These were ported together with the tests, but additional classes representing more concrete and meaningful concepts were also created, such as points and shapes. As test code was refactored, some common code was extracted to form a small testing language\cite{12}.
3.2 Identifying Useful Language Features

As InPUTpy was being developed, language features that were particularly useful in this context were noted. One important criterion for inclusion was that the feature must be lacking in Java. Apart from this, the development process itself served as a test for discovering such features.

For a feature to be useful, a minimum requirement is that it is functional (fit for purpose)\textsuperscript{13}. Functionality is demonstrated by the fact that the feature solves the problem it was applied to. Beyond that, determining usefulness is inherently subjective\textsuperscript{13}. It is assumed that a useful feature is one that, in addition to merely accomplishing the desired goal, enables a convenient and concise solution. Examples of solutions to specific challenges in InPUTpy show that the features highlighted in those examples fit this description.
Chapter 4

Design and Implementation

While InPUTpy is largely OO, some FP principles were also incorporated into the design. InPUTpy makes only modest use of higher-order functions and lambda expressions, but it tries to use pure functions as much as possible. The design also leans toward the philosophy of applying functions to data, rather than bundling functions with data. This can be seen in some of the abstractions.

The implementation section details solutions to problem-specific challenges, thereby highlighting important language features.

4.1 Design

One key decision that was made early on was to let Param objects represent a definition. Rather than being generators of values, as they are in InPUT4j, they are blueprints for how to create values. This means that Params are data structures that can be passed to a function that maps such information to values. It also means that Params never take on values. When a parameter is initialized, what is really happening is that a suitable value is generated, based on the definition of the parameter.

Python’s lack of immutable data notwithstanding, most objects are in-
Figure 4.1: Generators have the capability to generate values for certain types of parameters. A parameter is merely input to a generator.

Figure 4.2: DesignSpace, ParamStore and Param UML.
tended to be as immutable as possible, at least by convention. A user may be able to change attributes of objects, but that would violate the way the objects were intended to be used. For example, Params do not change once created. All Params are collected into a parameter store, which is also mostly immutable. Lastly, a design space contains a parameter store and is also immutable. There is an exception that runs all the way from a design space down to a parameter: the possibility of setting a parameter to a fixed value.

Fixed values could be seen as a violation of the idea that parameters are mere blueprints and do not generate values. An alternative interpretation is that this is a special case of providing instructions to a generator. Instead of an instruction such as ”generate a value between 1 and 10”, a fixed parameter might supply the instruction ”generate the value 3”.

Code example 6: Both B and Point depend on other parameters, but in different ways.

There are two different ways in which parameters can be dependent (code example 6). Numeric parameters may refer to other parameters in its inter-
val definition. Because endpoints must be evaluated before a value can be generated, the parameter depends on the referenced parameters. Structured parameters can depend on nested parameters that must be passed to a constructor or setter method during initialization. In that case, the dependency is a function of the code mapping. Both cases are solved by always initializing dependencies first, recursively.

Throughout the program, whenever a dependency must be resolved, the dependent component expects to receive some way of resolving it. This means that the information about dependencies is separate from the mechanism for resolving them. The latter usually takes the form of a dictionary.

### 4.2 Implementation

#### 4.2.1 eval

InPUT4j allows limits for numeric parameters to be defined using arbitrary mathematical expressions involving references to other parameters as well as mathematical functions. For example, an endpoint may be defined like this: \( \text{minIncl} = "2 + \text{Math.sqrt}(X) - Y" \). In InPUT4j, expressions are evaluated using a JavaScript engine. This is certainly one possibility, but Python presents another alternative. Unlike Java, Python has a built-in `eval` function that will evaluate strings and return the result. The math library in Python also matches the JavaScript math library quite closely. These factors make the `eval` function in Python a valid alternative to a third-party engine.

The potential benefit of using `eval` is that dependencies on other libraries can be minimized. Both alternatives pose two challenges. One problem is that the expressions can contain references to other parameters. These references must somehow be resolved to actual values. Another problem is one of security. Because `eval` truly evaluates arbitrary code, using the function to execute a user-supplied string can be dangerous. There is nothing to stop a user from creating a design space containing a parameter that is
defined with a minimum value that, when evaluated, will remove all files from the hard drive.

Both problems can be solved by the same mechanism. The eval function takes an optional name space argument. The name space is a dictionary, and any name that occurs in the expression being evaluated must be mapped to a value in that dictionary. An exception is the \_\_builtins\_\_ module, which, if missing, will automatically be included in the name space. References to a math function can be resolved simply by including the math library in the dictionary, using an appropriate key. Similarly, parameter names that occur in the expression can simply be mapped to their values, and the expression can then be evaluated by eval as-is. No transformations are necessary (code example 7).

```python
import math
namespace = {'Math': math, 'X': 4, 'Y': 1}
# Evaluates to 3
eval("2+Math.sqrt(X)-Y", namespace)
```

**Code example 7:** Satisfying dependencies with a namespace dictionary.

There are two minor issues remaining. Any impact on the real world requires the \_\_builtins\_\_ module. Even just printing something to the console requires the print function, which exists in \_\_builtins\_\_. As does opening files or importing other libraries. The fact that the module is automatically added to the passed-in name space re-introduces the original security problem. The problem can be finally solved by explicitly mapping the module name to an empty dictionary, as shown in code example 8.

The second problem is that the math functions exist under slightly different names in JavaScript and Python. The biggest difference is that the library is called Math (upper case) in JavaScript and math (lower case) in Python. This problem can be solved almost completely by just using the appropriate key, as shown in the example above.
# Prints 'Hello World'
eval('print("Hello World")')
# Prints 'Hello World'
ns = {}
eval('print("Hello World")', ns)
# NameError: name 'print' is not defined
ns = {'__builtins__': {}}
eval('print("Hello World")', ns)

Code example 8: The built-in functions must be be explicitly replaced with an empty dictionary.

4.2.2 Factories and kwargs

Keyword arguments are convenient in their own right, but the real value, at least in the context of this project, lies in Python’s expansion mechanism. Python provides two features related to keyword arguments. One is the ability to take a dictionary and expand it in a function call:

```python
f(**d)
```

The keys in the dictionary d will be matched against the keyword arguments of f. The corresponding values in the dictionary are then assigned to the matching arguments. A function can also be on the receiving end and define it’s arguments in a similar way:

```python
def f(**d):
    # Use d[key] inside this function.
    # ...
```

Here the keyword arguments that are passed to the function are gathered into a dictionary for use within the function.

Keyword arguments turned out to be very useful, both when creating parameters explicitly and when importing configurations.

The testing package includes a set of mock objects for importing preset configurations. These serve two purposes: testing and testing. By dupli-
cating the configuration in code in addition to the XML document, both versions can be imported and compared. This serves to verify that the XML import works reliably. The other benefit is that an efficient version of the configuration can be used during testing. The tests can thus avoid parsing an external document multiple times. The tests in InPUTpy are pretty fast anyway, so there is no immediate risk that they would be too slow\[12\], but faster tests are always a nice bonus.

When creating the configurations in code, the representation should be compact. Another fact to take into consideration is that XML element attributes are conveniently accessed as dictionaries when using the Element-Tree library. These two premises taken together suggest a simple solution. A dictionary is already available from ElementTree, and a dictionary is also a compact way to organize arguments. The solution is to use dictionaries in combination with kwargs when creating parameters (or any other XML element counterpart).

### 4.2.3 Runtime Type Loading

Loading and instantiating user-defined types at runtime is almost trivially easy with Python. Initializing values with custom setters is only a minor complication. The code that is required is so small that all the important parts can be included here.

Given a module name and a type name relative to that module, code example 9 shows the complete code necessary to return the corresponding class object.

```python
module = importlib.import_module(moduleName)
return module.__dict__[typeName]
```

**Code example 9:** Importing an arbitrary type.

Let C be such a class object. Creating an instance, given a list of argu-
ments args, amounts to a single line (code example 10).

```python
return C(*args)
```

**Code example 10:** Creating an instance of some type C using some sequence of arguments.

Given the name of a setter function and an argument, setting an attribute requires one more line of code (code example 11).

```python
obj.__getattribute__(setterName)(arg)
```

**Code example 11:** Initializing an attribute using a setter name and an argument.

It would certainly be possible to encapsulate all the steps inside some class or similar construct, but in InPUTpy (even more so than in InPUT4j) the majority of the work is dedicated to dealing with configuration-specific tasks. For example, deciding which attributes are set by constructor or by setter involves much more code than instantiating the object once that information is available.

### 4.2.4 Meta Parameters

There are two kinds of parameters that are only slight variations on other parameters: arrays and SChoice parameters. Both cases are handled by meta parameters in InPUTpy. This is one clear case where a dynamic language and the specific features of Python were particularly useful.

#### Array parameters

Compare the two parameter definitions in code example 12. Both look like unbounded numeric parameters of type integer. The only difference is that initializing B should create not just one such integer, but several.
could be said that B is of type array rather than integer, but an array of what? The type attribute in the definition can be read as defining B to be an array of two elements. Each element is an array of three elements, each of which is an integer. This description suggests that array parameters can be defined recursively. In InPUTpy, array parameters are defined by a size, which is the number of elements, as well as a parameter, which is the type of each element. The type parameter can either be an array parameter or any other non-array parameter.

It should be possible to use an array parameter like any other parameter in most cases, but conceptually, it is more accurate to say that an array parameter has another parameter, rather than that it is a parameter. Inheritance therefore seems to be an unsuitable mechanism for achieving polymorphism. The array parameter works more like a wrapper, and most calls should be forwarded to the wrapped parameter. The meta parameter is not a true decorator, but in a sense, it is enhancing, or “decorating”, the regular parameter with the quality of being an array.

Python makes the implementation of this solution very powerful and convenient. Its meta programming support makes it extremely easy to automatically forward calls to the wrapped parameter. A one-line method is all that is required (code example 13). Duck typing also makes the meta parameter completely generic and transparent. It is generic because any parameter type can be wrapped. In fact, any object at all can be wrapped. The meta parameter is transparent because it automatically supports all the operations that the wrapped parameter supports.
Figure 4.3: The recursive structure of an array meta parameter.
def __getattr__(self, attr):
    return getattr(self.__param, attr)

**Code example 13:** Calls to the wrapped object are easily forwarded.

**Figure 4.4:** SParams containing SChoices are translated to Choice meta parameters.

**Choice parameters**

SParams that contain nested SChoices are converted to another meta parameter of the Choice class. Each SChoice is converted to an SParam, while inheriting all the necessary information from the parent SParam. In other words, a Choice object contains a collection of SParams, each of which represents one of the possible versions of the original SParam. The new Choice object then takes the place of the original (figure 4.4). Once this transformation has been performed, it is quite straightforward to initialize a Choice. The process amounts to choosing one of the choices and then initializing that choice just like any other SParam.
Chapter 5

Results

This list summarizes the features that were highlighted in the previous chapter as being particularly important, together with their implications and application in InPUTpy.

Duck typing

- Meta Parameters
- Easy prototyping and program evolution

Meta programming

- Meta Parameters
- Easy access to attributes such as accessors

Functional programming

- Lambdas
- Class objects are functions
- Simplified testing

eval
• No need for external evaluation engine

**Keyword arguments**

• Easy XML parsing
Chapter 6

Discussion

6.1 Open Questions

There are a few open questions that represent missing features and undefined behavior in InPUTpy for now.

6.1.1 Numeric Types in Java and Python

Python only has one (built-in) integer and one floating point type, while Java has several. This presents some challenges in how values are handled. In particular, integers in Python have arbitrary size, so they must be limited artificially.

6.1.2 Enums in Python

Enums do not exist natively in Python. It is unclear what the best way would be to represent them. Resolving this involves finding out which uses of Python are idiomatic. It may also become a question of how much InPUTpy can be allowed to intrude upon the user. This is not nearly as much of an issue for regular user-defined classes.
6.1.3 Type Checking in InPUTpy

A Python programmer will be familiar with the dynamic typing of Python. It would be nice if a user could expect the same behavior while working with InPUTpy. However, configurations must be compatible with other InPUT implementations. In the interest of interoperability, some type checking may be necessary. The real question is how stringent such checks should be.

6.2 Justifying DocTest

The work for this thesis really consists of three parts: discovering a specification, implementing that specification, and analyzing the implementation. However, the main goal was to port InPUT4j to Python, not to write additional tests for InPUT4j. It therefore seems worthwhile to reflect on the first part of the methodology and assess whether spending time on this additional DocTest sub-project was wise.

This poses somewhat of a philosophical conundrum. In the absence of a complete specification, what does it mean for a program to be a "port" or "clone" of another program? In this case, there is a test suite to fall back on. Does this mean that a program that passes the tests can be considered a "clone" for our purposes? It is not clear that this definition is satisfactory.

One way to use the tests as the basis for an implementation would be to hard-code the expected results, essentially applying the Fake It strategy to production code. Such an implementation would pass the tests, but it would obviously not be adequate. It is clear that test cases are meant to be instances of some general rule. In the absence of sufficient documentation, this suggests that tests must be comprehensive enough to allow such rules to be inferred with reasonable confidence. The following argument is an attempt to show by contradiction that this was not the case with InPUT4j.

First, it should be acknowledged that attempting to specify exact behavior in all possible situations is often infeasible and unnecessary. Therefore,
extremely contrived and irrelevant edge cases are not compelling reasons to invalidate a purported clone. Such cases would only serve to attack a straw man\cite{15}, and they should thus be ignored.

If legitimate (not contrived or irrelevant) cases can be found, where the tests pass, but where the clone and the original program nonetheless differ significantly in their behavior, then it would seem inappropriate to conclude that the software under test is a successful clone. Finding such examples would show that the assumed definition must be insufficient. In fact, many such examples can be found. In the interest of brevity, two examples related to arrays will have to suffice.

Let X be an array parameter of three integers greater than zero (defined using type=\texttt{"integer[3]"} and exclMin=\texttt{"0"}). A design space or design containing X will support four parameter IDs: \texttt{"X"}, \texttt{"X.1"}, \texttt{"X.2"}, and \texttt{"X.3"}. The tests in InPUT4j show that arrays, as well as their elements - be they sub-arrays or leaves - can be accessed using those IDs. Consider two different code snippets operating on such a design (code examples 14 and 15). The assertion that the following assumptions are reasonable will serve as an additional premise.

- After the code in example 14 is executed, the value of \texttt{"X.1"} should be 1.

- Executing the code in example 15 should fail, because 0 is not greater than 0.

Both assumptions would be erroneous. If those are indeed reasonable assumptions, that would make the argument sound, but then passing the tests cannot be a sufficient criterion for qualifying as a successful clone.

These two examples are especially compelling, because it is not at all clear that the behavior reveals bugs, which would complicate matters even more. Without a specification or other documentation as arbiter, how can any unspecified behavior ever be classified as a bug to begin with? This is
int[] arrayOfOnes = { 1, 1, 1 };
design.setValue("X", arrayOfOnes);

**Code example 14:** Setting the array to an array of legal values.

int[] arrayOfZeroes = { 0, 0, 0 };
design.setValue("X", arrayOfZeroes);

**Code example 15:** Setting the array to an array of illegal values.

an additional problem that DocTest was intended to at least mitigate, but which will not be elaborated on further here.

While a program could be created by an honest attempt to pass the tests, it is not clear that the program could meaningfully be called a clone or port. Some additional effort to establish actual and desired behavior was therefore necessary.

However, in retrospect, it also seems that priorities could have been more sensibly assigned. It is clear that some tests in DocTest are needlessly specific, and the time that went into writing those could have been better spent focusing on the main goals.

### 6.3 Ethical Implications

The InPUTpy program is intended to be useful in ways that are implicit in the problem motivation. It is a tool both for answering questions and supporting communication between researchers. Because it is such a general tool, it is hard to imagine any direct negative effects it may have. Ethical concerns related to the use of InPUT are thus limited to the ubiquitous danger associated with the use or misuse of knowledge in general.
6.4 Conclusion

All the goals were not achieved. At the moment, the biggest contribution may turn out to be the DocTest project. Not only did the tests reveal several bugs in InPUT4j, but by demonstrating the behavior of InPUT4j in great detail, they will also be useful for any future work related to InPUT, regardless of implementation language.

Some features of InPUT4j were not implemented, so all of the ported tests do not pass. However, the implementation is complete enough to reveal several useful features provided by Python.

Duck Typing may have been the most useful language feature overall. Python’s meta programming and FP support, combined with the fact that classes are essentially factories\[16\], made runtime-type type loading almost trivial.

6.5 Future Work

6.5.1 InPUT Specification

A significant portion of the work of porting InPUT4j to Python was an exercise in reverse engineering, in order to discover a specification to implement. This work has branched off into a separate project\[17\] to create and maintain a specification of InPUT that is separate from any specific implementation.

6.5.2 InPUTpy

There is still quite a bit of work to be done on the Python implementation of InPUT. Some features are missing, and the design can certainly be improved. It is also likely that there is room for significant performance optimizations.
6.5.3 InPUThs

By enabling some functional programming (FP), InPUTpy adds important diversity to the family of InPUT implementations. Unfortunately, Python’s FP support is limited. Python is functional in the same sense that JavaScript, Lua and Ruby are functional; functions are first-class values. Python thus supports higher-order functions. However, Python only has limited support for lazy evaluation, which is an important FP feature\cite{18}. Iterators and generators offer basic lazy evaluation, but it is not built into the language as in Scala or Haskell, for example. Python also has fairly weak support for immutable data and concurrency.

An implementation in a "true" functional language is still lacking. Once a stable specification of InPUT exists, a Haskell implementation would probably be the next logical step.
Bibliography


Appendix A

InPUTpy github repository

The code for the implementation is stored in the InPUTpy github repository\[19\].