Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

An Implementation of Prorogued Programming in Python

Bachelor Thesis

G. Piatti

September 2021

Advisors: Prof. Dr. Z. Su, S. Thorgeirsson
Department of Computer Science, ETH Zürich

Abstract

Prorogued programming is a programming paradigm originally described by Afshari et al. in "Liberating the Programmer with Prorogued Programming". It is a new programming paradigm that better aligns with a programmer thinking process. A prorogued programming language supports three basic principles: proroguing concerns, the ability to defer a concern, to focus on and finish the current concern; hybrid computation, the ability to involve the programmer as an integral part of the computation; executable refinement, the ability to execute any intermediate program refinements. In their paper, Afshari et al. introduce the paradigm and describe an implementation for the language C#, which modifies the C# compiler.

We implemented a minimal prototype of the paradigm for the Python Language by constructing a Python library. Our approach is based on Python's metaprogramming features and does not alter Python's interpreter. We describe the prorogued programming paradigm for Python, its realisation using Prorogued Python, our extension to Python.

We performed a qualitative analysis based on a small survey to investigate how Prorogued Python is used and how it changes the development process. As a result, we discovered that Prorogued Python provides some easing to the development process and reduces what the programmer needs to think about while coding.

Contents

Contents **iii**

1 Introduction **1**
1.1 Prorogued programming 1
1.2 Current implementation of the paradigm 2
1.3 Overview 2

2 Methods **3**
2.1 Background 3
2.1.1 Similar approaches to prorogued programming 3
2.1.2 The Python Language 4
2.2 Description of the language (extended Python) 4
2.2.1 Syntax 4
2.2.2 Semantics 5
2.2.3 Type system 6

3 System description **9**
3.1 Prorogued call integration in Python 9
3.1.1 Class name-space integration 9
3.1.2 Module-level integration 10
3.1.3 General prorogued logic 11
3.1.4 Why local function calls cannot be implicitly prorogued 12
3.1.5 What happens with attributes 13
3.2 Type checking of prorogued calls 13
3.3 Prorogued function's return value 15
3.3.1 Variable binding 18
3.3.2 Cache 19

4 Evaluation **21**
4.1 Use cases and examples 21

4.1.1 Progressive refinement 21
4.1.2 API proxy 21
4.1.3 Interactive control flow 22
4.1.4 Shadowing while debugging 23
4.2 Study 23
4.2.1 Study goals 23
4.2.2 Study setting 24
4.2.3 Programming task description 24
4.2.4 Study results 24
4.2.5 Study conclusions 26

5 Conclusion **29**

A Programmer's manual **31**
A.1 Library purpose 31
A.2 How to enable this behaviour 31
A.3 Specification of the prorogued call behaviour 32
A.4 Prorogued handler - interaction with the programmer 32
A.4.1 User interface 33

B Study handout **37**
B.1 Introduction 37
B.2 Prorogued programming paradigm 37
B.3 Task's description 38
B.4 Survey 39

C Survey questions **41**
C.1 General demographic question 41
C.2 Questions 41

D Study responses **43**

Bibliography **53**

Chapter 1

Introduction

1.1 Prorogued programming

Prorogued programming is a programming paradigm described by Afshari et al. in "Liberating the Programmer with Prorogued Programming" [3]. It is a new programming paradigm that better aligns with a programmer's thinking process.

The prorogued programming paradigm belongs to a family of test-driven methodologies, with other examples such as declarative mocking [15] and the approach by Henkel et al. [8] that also supports testing incomplete code.

Definition 1.1 *Prorogued programming language (PPL) [3]: A prorogued programming language supports three basic principles:*

- *proroguing concerns: the ability to defer a concern, to focus on and finish the current concern;*
- *hybrid computation: the ability to involve the programmer as an integral part of the computation;*
- *executable refinement: the ability to execute any intermediate program refinements.*

One of this paradigm key features is to permit the execution of incomplete programs, allowing the programmer to debug and refine the code during the development process. With the mainstream programming paradigms, which are all based on the descriptive declarative programming paradigm [14], the compiler will not compile code that depends on non-existing functions. Respectively for interpreted languages, the interpreter will abort the execution of code that depends on non-existing functions.

To allow the execution of an incomplete program, the compiler, respectively

the interpreter, needs to be aware of invocations to unimplemented functions[1] and provide a specific handler to glue them to the existing code.

1.2 Current implementation of the paradigm

Only one implementation of prorogued programming for a programming language exists at the state of writing, Prorogued C#. This implementation was made by the creator of this paradigm. In the paper [3], the authors described the principles behind this programming paradigm and presented their design for Prorogued C#, their extension to C#.

Their design choice was to introduce a new keyword in the language's syntax: **prorogue** to label an invocation expression. A call expression labelled with this keyword means that the corresponding function, or more generally, the associated code, is not yet implemented. Thus it cannot be found by the compiler during compilation. To allow the program's execution, they embedded a so-called prorogue dispatcher, whose main purpose is to intercept a prorogued call (marked by the prorogued keyword) and inject a placeholder instance, which returns a prorogued value.

They enable hybrid computation by allowing the programmer to interact with the running code. For example, the prorogue dispatcher asks the programmer to supply a return value when reaching a prorogued call. To alleviate the interaction with the programmer, they cache the responses indexed by the function input's value. Moreover, the evaluation of the prorogued calls follows a lazy pattern: the prorogued dispatcher returns an implicit future, which represents the call with the arguments; only when the return value is needed a user interaction is made.

To collect the values from the user, they used as input JavaScript Object Notation (JSON) serialization.

1.3 Overview

In chapter 2, we describe the behaviour of the extended Python and give the necessary background on standard Python. Then, in chapter 3, we present how the library works and how it is integrated within Python. Finally, in chapter 4, we show some use cases for the developed tool and present the results of a small study.

[1] In this chapter, we refer to function as any callable entity, be it local function or a method.

Chapter 2

Methods

2.1 Background

2.1.1 Similar approaches to prorogued programming

We briefly explore similar approaches to prorogued programming and compare them to prorogued programming.

Programming with Ghosts

In the paper "Programming with Ghosts" [5], O. Callaú and É. Tanter propose an extension to a Java's IDE, in their case Eclipse, to support smooth, incremental programming. They focus on eliminating ineffective error messages from the IDE for undefined entities, which will be coded later.

They achieve this by making undefined entities explicit in the IDE, which they call ghosts; furthermore, they do type inference and refinement on those entities. Later in time, the programmer can "bust" ghosts into actual definition, for example by generating code skeletons.

Compared to Prorogued programming, the Ghosts extension does not allow the programmer to run partially implemented programs; to achieve this, we still need to code or generate each undefined entity. Furthermore, no testing or debugging of modules that depend on ghosts is possible.

Declarative mocking

In the paper "Declarative mocking" [16] S. Freeman et al. describe an extension to test-drive development [4], that allows testing code as if it had everything it needs from its environment.

A fundamental principle is to test each object in isolation by replacing the objects it depends on with objects, that test that they are called according to their specification and return the required behaviour.

Compared to prorogued programming, it is needed to define expectations and stub for each mock object's in code; then, it is possible to invoke the method to be tested. This approach also allows to test code that depends on some undefined entities as in prorogued programming, but on the contrary, it requires the programmer to think in advance and write a mock for it.

Only prorogued programming allows to perform incremental programming and to run the code without requiring more work from the programmer's side.

2.1.2 The Python Language

Python is "an interpreted, object-oriented, high-level programming language with dynamic semantics" [13]. It is a high-level language used mainly for prototyping, website development, scientific computing and education. Python supports object-oriented programming but does not enforce its usage. Moreover, it is a garbage-collected language; objects are never explicitly destroyed; when they become unreachable, they may be garbage-collected; thus, their space is freed.

Python Language paradigm

Python supports multiple programming paradigms, namely object-oriented, imperative and functional programming [17]. In a large program, different sections could be written in different paradigms. Note that Python is an impure functional language since it can maintain state [6].

Python type system

Python type system is based on Duck Typing: "If it walks like a duck, swims like a duck and quacks like a duck, then it is a Duck" [11]. This is better known as shape typing or structural typing. Python does not check the object's type to know if a given object has the right interface. Rather, Python tries to access the method or property directly; an exception is thrown at run-time if it is not present.

2.2 Description of the language (extended Python)

In this section, we describe the extension that we made to language behaviour. All proposed enhancements are implemented by the library `ppl`.

2.2.1 Syntax

Our implementation of the prorogued paradigm does not introduce any new token to the language; instead, we implicit declare any call to a not

implemented function or method to be prorogued. One issue that we would have if we introduce a new keyword is that we need to modify Python's own interpreter: we need to adapt the parser to recognize the new keyword and then adapt the compiler to our needs.

Instead, we can mark which call is implicitly prorogued by leveraging meta-classes and meta-modules, as described in the semantics subsection.

Next, we describe the syntax for the return value of a prorogued call. An expression is what the programmer can input in the console when the library asks for a prorogued call return value. Basic Python's constructs, like dictionaries, tuples, sets, and lists are supported and basic values.

2.2.2 Semantics

We decided to pursue a fine-grain approach to allow for the coexistence of plain Python and prorogued Python. Therefore, we explicitly mark for which class and module the prorogued behaviour is enabled. This choice can be performed for each Python namespace (at module level or class level).

For a class, which is marked, the following holds: if a method is not declared in the code, then it is implicitly prorogued.

```
1 class Example(metaclass=PPLEnableProroguedCallsStatic):
2     def m1(self, x):
3         pass
4
5 t = Example()
6 t.m1(42)
7 t.m2(42)
8
```

Code Listing 2.1: Example

In example 2.1, the call to `t.m2()` does not correspond to any declared method in the `Example` class and thus is implicitly prorogued. Following the definition 1.1, the library then interacts with the programmer asking for the function's output.

An implicitly prorogued call can be made either inside the class definition, i.e. as part of some code describing the class implementation, or by calling some bounded method on a class instance.

The same applies to modules. For modules, we also need to enable this behaviour, as shown in code 2.2.

```
1 from ppl import enable_module_level_prorogued_calls
2 enable_module_level_prorogued_calls(__name__)
```

Code Listing 2.2: How to enable prorogued functionality at module level

When the prorogued calls are enabled, the following applies: given a call to a function that is a member of a name-space with this feature enabled, which does not correspond to any actual function definition, this is assumed to be implicitly prorogued. For classes, we can specify two different behaviours:

- static prorogued calls; the prorogued calls have the same behaviour for all instances. The prorogue handler considers only inputs passed to the function.
- instance prorogued calls; the called functions could have different behaviours for different instances. The prorogue handler considers inputs passed to the function and instance's attributes.

For modules, we only have one behaviour: prorogued calls involve function's input and do not consider external context.

The return values that a prorogued call can assume are basic types and basic structures like lists, dictionaries, tuples, sets of basic types. Moreover, we support returning constructs based on function's local variables, objects' attributes and class attributes.

2.2.3 Type system

To enforce some meaningful behaviour of prorogued calls, we implemented some type checking rules. The type checker runs at run-time, specifically each time a prorogued call is made.

We first describe the high-level ideas behind the type system and its main objectives, then in a later section, we will describe the type system and its implementation in detail.

Prorogued call arguments

For each occurrence of the same prorogued call, we could proceed as follows. First, we inspect the parameters of the first call, and we save the signature of the function. Then we check if all subsequent calls express a function that is in a subtype relation with the first call, according to Definition 2.1.

Definition 2.1 *Function subtype: A function f is a subtype of a function g if and only if f's arguments are contravariant to g's arguments type, and f's output type is covariant to g's output.*

The idea is that we first compute the initial type of the prorogued function based on the information that we gather on the first call. Then we check if subsequent calls adhere to the specification given by the first seen call.

We now distinguish two aspects. A function signature is given by the number of arguments without a name, by names of named arguments and by their

type. To type check a function, we first check that the number of arguments matches, and then for each argument, we perform a type check. To check that the number of arguments is consistent across calls, we can proceed by case distinction.

Let's focus the attention on a single argument. Assume we have two different occurrences of a prorogued call; then, we can check whether the second occurrence of the argument is a sub-type of the first occurrence. If, for all arguments this relation holds, we have that the first part of Definition 2.1 is satisfied. The main problem with this approach is that typing depends on the program execution path, i.e. different execution paths lead to different first calls to a prorogued method and thus alter the sub-typing relation. By applying this strategy, we do not include the possibility that the first call's arguments are a sub-type of the second call. Since we do not want to restrict Python's dynamic power, we decide to disregard this direction. Moreover, all types are a sub-type of the `object` type, and thus it could well be that a prorogued function takes an argument with any type. This cannot be know a priori.

The approach that we took is based on the previous insights and applies this fundamental principle: we cannot fully type prorogued functions, but we can do some partial typing and warn the programmer that a prorogued function is called with different types.

Prorogued call output

The employed strategy for the output type is relatively simple. Recall that Python employs dynamic type checking; thus, in vanilla Python, the check is done at run-time. In vanilla Python, when a function returns a value, no type check is made; in fact, common checks are only performed when using the returned value as part of some code.

We can leverage this behaviour for our extension; namely, a prorogued function returns a value independent of its type, and then the Python system handles the rest.

Chapter 3

System description

In this chapter, we describe how the proposed library implementation integrates with Python and what Python features we leverage to achieve the behaviour specified in the language specification.

3.1 Prorogued call integration in Python

In subsection 2.2.2, we already described that our implementation of the prorogued programming paradigm is implicit, i.e. when calling a function, if its definition is not present in any name-space; the call is implicitly assumed to be prorogued. Only functions in the class name-space and at module level namespace are implicitly prorogued. In the current implementation, local function calls made inside a function definition are not implicitly prorogued; later, we will explain why this is the case.

3.1.1 Class name-space integration

For the class name-space, the integration is based on functionalities provided by metaclasses. Metaclasses are a way to customize the class creation process; by default, classes are constructed using `type()`. This process can be customized by passing the `metaclass` keyword argument in the class definition line [10].

Class decorators [9] are not an option since they only act on the class where they were directly applied. This means subclasses of the decorated class may not inherit the customization.

What we expose to the programmer are only two different metaclasses `PPLEnableProroguedCallStatic` and `PPLEnableProroguedCallsInstance`, their functionality is quite similar, we focus first on the general cases and then we outline the differences. To enable the prorogued programming paradigm for a given

class, we need to create it by passing one of the two options above as metaclass keyword argument; this allows us to inject some code that allows us to detect prorogued calls and dispatch them to the prorogued handler.

We leverage the following Python behaviour to detect when a function is not present in the current name-space: the function `object.__getattr__(self, name)` is called when the default attribute access fails with an `AttributeError` [1]. Once this is called, we return a function wrapper that expects any number of positional parameters and named arguments to gather the input values of the prorogued call.

Let us demonstrate this behaviour through example A.2. Assume the following code is defined and executed.

```
1 class Example(metaclass=PPLEnableProroguedCallsStatic):
2     def method1(self, x):
3         print(x)
4
5 t = Example()
6 t.method1(42)
7 t.prorogued_method(42, arg1=3)
```

Code Listing 3.1: Example prorogued function

When `t.method1` is called, this is found in the class `Example` and thus the correspective code is executed. When `t.prorogued_method` is called, the function `t.__getattr__(self, name)` is called since, no attribute named `prorogued_method` is found in the instance of `Example` and in the tree class for `Example`, then we return a function that wraps the dispatch to the prorogued handler.

3.1.2 Module-level integration

Prorogued calls are also supported at module level, i.e. when calling a function that is not present in the module, this is assumed to be implicitly prorogued. To enable this behaviour, the code 3.2 needs to be in the module definition.

```
1 from ppl import enable_module_level_prorogued_calls
2 enable_module_level_prorogued_calls(__name__)
```

Code Listing 3.2: example.py

To achieve this goal, we inject a custom implementation to the module's function `__get_attr__` that behaves as in the class case, as already described in the previous section.

We demonstrate this in action with example 3.3. In this example the function `prorogued_fn` is not defined in the code for module `example_module`. Thus, the semantics is implicitly prorogued, and thus, the prorogued handler is executed.

```
import example_module as module
module.prorogued_fn(42)
```

Code Listing 3.3: Example module usage

3.1.3 General prorogued logic

We perform eager evaluation of prorogued calls. At the first call of the prorogued method, we also set the missing attribute (in the example `prorogued_method`) to the prorogue handler, such that subsequent function call with the same name are handled by the same instance of the prorogue handler; this is achieved by using the `object.__setattr__(self, name, value)` function [2].

The prorogue handler procedure does the following high-level operations:

- gathers the arguments and computes the call signature;
- type-checks the input values;
- if the type-check succeeds without errors, then asks the programmer for the return value

To allow more expressiveness and flexibility, there are two different ways of operation of prorogued calls: static prorogued call or instance prorogued call. This difference is achieved by binding the prorogued handler in the former case to the class object and in the latter to the class's instance via `object.__setattr__`. The motivation behind this customization is to enable different behaviour of a prorogued function that is part of two different instances of the same class. In example 3.4 we can see this in action, when using `PPLEnableProroguedCallsStatic` a call to a prorogued function with the same parameters returns a cached value if present because the function was already called even from another instance. On the other hand, when using `PPLEnableProroguedCallsInstance` calls with the same parameters to different instances do not behaves equally; in fact, the programmer is involved and is asked to input a return value for both occurrences.

```
class ExampleStatic(metaclass=PPLEnableProroguedCallsStatic):
        pass
t1 = ExampleStatic()
t1.prorogued_method(42) #returns value v
t2 = ExampleStatic()
t2.prorogued_method(42) #returns cached value v


class ExampleInstance(metaclass=PPLEnableProroguedCallsInstance):
        pass

t1 = ExampleInstance()
t1.prorogued_method(42) #returns value v
t2 = ExampleInstance()
```

```
15 t2.prorogued_method(42) #returns value v'
```

Code Listing 3.4: Example prorogued function binding

In sections 3.2-3.3 we describe the type-checking procedure and how the library gathers the return value.

3.1.4 Why local function calls cannot be implicitly prorogued

We demonstrate through example 3.5 why local prorogued functions calls cannot be implemented in Python.

```
1  class Example(metaclass=PPLEnableProroguedCallsStatic):
2      def method1(self, x):
3          self.prorogued_method(42, arg1=3)
4
5      def method2(self):
6          prorogued_fn(42)
7
8  t = Example()
9  t.method1(42)
10 t.method2()
```

Code Listing 3.5: Example local prorogued function

With the current implementation, the call to `t.method1(42)` succeeds, and the prorogued handler correctly intercepts the call to `prorogued_method` since this function is bounded to an object instance using the keyword `self`. On the other end the call to `t.method2()` does not succeed, in fact by trying to access `prorogued_fn` we get the errror `NameError name 'prorogued_fn' is not defined`.

Supporting this type of function class as implicitly prorogued poses some challenges that clash with the fundamental limits of the Python programming language. In the following paragraphs, we describe the issues given by those limitations.

Let's look at how Python resolves names. Given a name, this name is resolved using the nearest enclosing scope, from the local scope to the global scope. When a name is not found, a `NameError` exception is raised [12]. We could think of a way to exploit this and detect if we try to access a name that is not present during run-time. Then we can assume that we can distinguish functions from variables; we could create a function wrapper that invokes the prorogued handler at run-time.

This approach does not work. The first issue is that when a statement raises an exception, there is no way to re-execute the statement that produced the exception, like in hardware when a fault occurs. Instead, we could execute the function that encloses the prorogued call, but this would work only when the function does not have any side effects beyond the function's scope, which is clearly a big limitation.

We could instead leverage the fact that we can inject global states when executing a function and run the function code in a sandbox until no missing function is found. Finally we execute the function outside the sandbox. One needs to be careful how to handle side effects, especially IO side effects.

This last idea needs some further investigation.

As shown above, there is a fundamental difference that function searched in local or global namespaces raise `NameError` and do not trigger any built-in function call that can handle the missing definition, as we have for missing class's attributes.

3.1.5 What happens with attributes

Since we are currently leveraging the fact that when an attribute is not found, the function `object.__get_attr__` is called, we do not distinguish at first if the attribute is a function or not. Instead, we assign our wrapper to this property, and then only if a call is made by using the operator `()`, then the prorogued call workflow start executing.

Each not found attribute of a class or module is assumed to be a prorogued function. This means that we always return a function wrapper to the prorogued handler for every not found attribute. The code 3.6 demonstrates this behaviour.

```
1 class Example(metaclass=PPLEnableProroguedCallsStatic):
2     pass
3
4
5 t = Example()
6 o = t.attribute_not_defined
7
8 print(o)  # <function PPLEnableProroguedCallsStatic.__init__.<locals
    >.getattr.<locals>.wrapper at 0x7fdc30c2f1f0>
9 print(type(o))  # <class 'function'>
```

Code Listing 3.6: Wrapper to prorogue handler

3.2 Type checking of prorogued calls

In this section, we describe the implementation of the type checking for the parameters of a prorogued function. Recall the basic principle already described in subsection 2.2.3 that we do partial typing and warn the programmer that a prorogued function is called with different types.

To perform this process, we save the signature of the first prorogued call and use this as a reference for subsequent calls. The type checker proceeds as follows:

- check that the number of parameters and named parameters matches;
- check that the name of the named parameters matches;
- check that not more named parameters are passed to the function.

If the above conditions are not satisfied the error `PPLTypeError` is raised.

We then check for the actual type of a single parameter; if they do not match the first call, some warnings are emitted. Those warnings inform the programmer that the current function call signature is a subtype of the first or a supertype of the first or that they are incomparable.

We now present a few examples to illustrate better this system.

```
class Example(metaclass=PPLEnableProroguedCallsStatic):
    pass

t = Example()
t.prorogued_method(42, arg1=10)  # TYPING: ok (first call)
```

Code Listing 3.7: Typing example part 1

When first calling `prorogued_method` the system computes the signature and saves this. Then, the type system performs the same logic for the subsequent function call to the same function. The following code listings are executed each directly after code listing 3.7.

```
t.prorogued_method(42, arg1=1)  # TYPING: ok, different argument
```

Code Listing 3.8: Typing example part 2

Here the call to `prorogued_method` differs from the first only by the value of the second argument, the type remains the same, and thus it passes the type check.

Let's now look at some more interesting cases.

```
t.prorogued_method(42)
# PPLTypeError('prorogued_method() missing 1 arguments')
```

Code Listing 3.9: Typing example part 3

```
t.prorogued_method(42, bad_arg=1)
# PPLTypeError('prorogued_method() missing 1 required named argument
    : arg1')
```

Code Listing 3.10: Typing example part 4

In code listings 3.9 and 3.10 the function `prorogued_method` is called, in both cases, with only one argument, and thus it does not type successfully. According to the specification `PPLTypeError` is raised.

```
t.prorogued_method(42, arg1=True)
# PPLSubTypeWarning(
# FunctionCallSignature: [<class 'int'>, arg1: <class 'int'>],
# FunctionCallSignature: [<class 'int'>, arg1: <class 'bool'>])
```

Code Listing 3.11: Typing example part 5

The last case shows an interesting feature, namely typing warnings. As shown in the example, a warning is issued when the number of parameters match but the function calls do not have an identical signature. They are three different warnings that reflect the possible relationship that two signatures can have:

- `PPLSubTypeWarning`: the following typing relation holds *call signature* $<:$ *first call signature*
- `PPLSuperTypeWarning`: the following typing relation holds *first call signature* $<:$ *call signature*
- `PPLIncomparableTypeWarning`: neither of the above relations does hold

No checking of the return type is made since we can employ vanilla Python behaviour directly.

```
res = t.prorogued_method(42, arg1=10)  # TYPING: ok
print(res + 3)
```

Code Listing 3.12: Typing example part 5

Assume the `prorogued_method` returns a `string`, then Python's typing system takes care of the rest. So when executing line 2, the statement raises an error. Assume instead the `prorogued_method` returns a `int`, then Python's typing system takes care of the rest. So when executing line 2, the statement does not produce any error. This behaviour is the same that happens with vanilla Python when returning a value from a function.

3.3 Prorogued function's return value

In this section, we describe how we gather the prorogued function's return value. What values are accepted, and how this integrates with vanilla Python.

Recall that by the semantics defined in section 2.2.2, we know that, if a function is implicitly prorogued, it is the programmer's responsibility to define a return value for a given set of inputs. The prorogued handler manages this process.

We distinguish 2 different cases. The first case occurs when a function was previously called with the same inputs before; then, the prorogued handler returns the value present in cache. The second case occurs on newly seen

inputs, is such case we prompt the programmer with a message containing the current function name, its inputs and ask him to input the return value; as shown in figure 3.1

To allow for a pleasant programming experience, we decided to handle the library's user interaction via an external window, such that no interference with the program IO is made in the command line.

In the top panel, we show all messages from the library; in the middle, there is information regarding variables of object's instances, such that when `PPLEnableProroguedCallsInstance` is enabled. On the bottom, there is a input field for the function's return value.

The UI expects the user to insert in the input field the return value of the prorogued function call displayed on the top panel. Then, using the enter key to confirm the input and communicate to the library to continue the program's execution using the provided return value.

Figure 3.1: PPL dialogue window

As possible return values, we allow returning any Python basic type and basic containers (dictionary, set, lists) of basic types and local function variables, object attributes and class attributes.

We employ a parser that parses the programmer output choice according to the grammar described in Grammar to achieve this 3.2. We then convert the resulting AST to a Python expression that is then returned by the prorogued function.

⟨*expression*⟩ ::= ⟨*tuple*⟩
| ⟨*list*⟩
| ⟨*dict*⟩
| ⟨*set*⟩
| ⟨*singleton*⟩
| ⟨*atom*⟩

⟨*atom*⟩ ::= ⟨*float*⟩
| ⟨*int*⟩
| True
| False
| None
| ⟨*escaped_string*⟩
| ⟨*local-var*⟩
| ⟨*instance-var*⟩
| ⟨*class-var*⟩

⟨*singleton*⟩ ::= '(' ⟨*expression*⟩ ',)'

⟨*expression-list*⟩ ::= ⟨*expression*⟩ ',' ⟨*expression-list*⟩ | ⟨*expression*⟩

⟨*tuple*⟩ ::= '(' ⟨*expression-list*⟩ ')'

⟨*list*⟩ ::= '[' ⟨*expression-list*⟩ ']'

⟨*set*⟩ ::= '{' ⟨*expression-list*⟩ '}'

⟨*kvpair*⟩ ::= ⟨*atom*⟩ ':' ⟨*expression*⟩

⟨*kvpair-list*⟩ :: = ⟨*kvpair*⟩ ',' ⟨*kvpair-list*⟩ | ⟨*kvpair*⟩

⟨*dict*⟩ ::= '{' ⟨*kvpair-list*⟩ '}'

⟨*local-var*⟩ ::= ⟨*identifier*⟩

⟨*instance-var*⟩ ::= 'self.' ⟨*identifier*⟩

⟨*class-var*⟩ ::= ⟨*identifier*⟩ '.' ⟨*identifier*⟩

Grammar 3.1: Programmer's input return value grammar specification

3.3.1 Variable binding

In the grammar the rule `<identifier>` correspond to any Python identifier described by Python's syntax. We distinguish the 3 different cases.

⟨*local-var*⟩ ::= ⟨*identifier*⟩

⟨*instance-var*⟩ ::= 'self.' ⟨*identifier*⟩

⟨*class-var*⟩ ::= ⟨*identifier*⟩ '.' ⟨*identifier*⟩

Grammar 3.2: Grammar variables

The first case is the easiest; we look up the identifier directly in the local dictionary.

```
class Example(metaclass=PPLEnableProroguedCallsStatic):
    pass

e = Example()
e.prorogued_method(a0, a1, a2)

#return value:$ a
```

In the second case, we want to access attributes in the same instance of the object in which the prorogued function lives. This is only possible when we are using `PPLEnableProroguedCallsInstance`; otherwise, we can only access static variables. In this case, we look up the identifier in the object.

```
class Example(metaclass=PPLEnableProroguedCallsInstance):

   def init(self):
        self.a = 42

e = Example()
e.prorogued_method()

#return value:$ self.a
```

For the last case, we want to allow access to static variables declared via class definition. This case is a bit subtle; the first identifier needs to match the current object class name, as we would do in vanilla Python. If the name corresponds, then we look up the second identifier in the class. This way of accessing attributes works in both `PPLEnableProroguedCallsInstance` and `PPLEnableProroguedCallsStatic`.

```
class Example(metaclass=PPLEnableProroguedCallsStatic):
   a = 42

e = Example()
e.prorogued_method()

#return value:$ Example.a
```

3.3.2 Cache

As mentioned before, we want to be able to cache return values of prorogued calls, such that on subsequent calls with the same input, we can avoid asking the programmer for the return value. We assume that prorogued functions are deterministic and do not contain randomness. That is, for a fixed input, the result is the same.

Python built-in function cache functionality is based on hashing, and such cannot be fully used for our purposes since, in Python, mutable containers (like lists, sets) cannot be hashed. Instead, we employed cache functionality provided by the package `cachetools`,[1] which allows us to provide a custom function that maps the function's argument to a key value.

Caching follows both the instance call behaviour and the static call behaviour. We illustrate this through example 3.13

```
class ExampleStatic(metaclass=prorogue.PPLEnableProroguedCallsStatic
    ):
        pass
t1 = ExampleStatic()
t1.prorogued_method(42) #returns value v
t1.attribute = 1000
t1.prorogued_method(42) #returns cached value v
t2 = ExampleStatic()
t2.prorogued_method(42) #returns cached value v


class ExampleInstance(metaclass=prorogue.
    PPLEnableProroguedCallsInstance):
        pass

t1 = ExampleInstance()
t1.prorogued_method(42) #returns value v
t1.attribute = 1000
t1.prorogued_method(42) #returns value w
t2 = ExampleInstance()
t2.prorogued_method(42) #returns value k
```

Code Listing 3.13: Example prorogued function caching

[1] https://pypi.org/project/cachetools/

As described in the section 2.2.2, we have two different operation modes for proroguing calls in a class namespace; the caching system also considers this aspect when caching values.

From example 3.13, we see that when a variable local to a class instance is modified, and we are operating with `PPLEnableProroguedCallsInstance`, calling `prorogued_method` again with the same inputs does not return necessary the same value. So the programmer is involved again and is asked to provide a return value.

In the implementation, we consider all instance attributes as part of the cache's key, so modify an attribute results in a new key being computed.

Chapter 4

Evaluation

In this chapter, we present a few examples to demonstrate the capabilities of prorogued programming in Python. We also present a qualitative analysis investigating how programmers use Prorogued Python; to outline its weaknesses and strengths.

4.1 Use cases and examples

We present different use cases for Prorogued Python that demonstrate how the library can facilitate the development process.

4.1.1 Progressive refinement

The first use case is key to a prorogued paradigm language. We can leverage proroguing concerns to code an application focusing only on one component at a time by assuming the implementation of other components where a component can be either a function or a module.

4.1.2 API proxy

The first example supposes that we already have an external API to use within our application. Imagine we are developing an application that uses some API to gather some data. This API usually runs on a server, and creating a test server, either on the internet or on the local machine, is time-consuming and costly.

We would like to have more flexibility than simply gather constant values. We now leverage Prorogue Python to build a proxy almost instantaneously to run the application without deploying any server.

A typical usage example would be the following, where in code listing 4.1 we show an example code that uses the API `myapi`.

```
import myapi as api

api.init()

def process_data(user):
    data = api.get_data(user, 'profile')
    #...
    user_book = api.get(data.user_books, 'details')
```

Code Listing 4.1: Example API usage

We now substitute the reference `api` from `myapi` to `prorogued_api` in code listing 4.2 , now all module level calls are prorogued, and thus the programmer can mock the API functionality.

```
import prorogued_myapi as api

api.init()

def process_data(user):
    data = api.get_data(user, 'profile')
    #...
    user_book = api.get(data.user_books, 'details')
```

Code Listing 4.2: Example API usage - PPl

The only step that is needed to achieve the goal is to create a new file `prorogued_myapi.py` and enable the module level prorogued calls, as shown in code listing 4.3.

```
from ppl import enable_module_level_prorogued_calls
enable_module_level_prorogued_calls(__name__)
```

Code Listing 4.3: prorogued_myapi.py

Compared to standard mocking, this approach does not require the programmer to write code before running the application to mock the API functionality, so the programmer does not think ahead for possible values.

4.1.3 Interactive control flow

Instead of simply using the library to develop software by progressing refinement, we can use the library to interactive change the control flow of a given program by inserting some prorogued calls at control flow instructions. Let's demonstrate this by means example 4.4. Where depending on the returned prorogued value, we branch in code 1 or 2. Since we are explicitly not caching any call, this prorogued call is interactively executed each time we reach the `if` statement.

```
class ControlFlowExample(metaclass=PPLEnableProroguedCallsStatic,
    cache=False):
    pass
```

```
#Some where in code

if ControlFlowExample.control_point_1():
    #branch 1
else:
    #branch 2
```

Code Listing 4.4: Control flow example

4.1.4 Shadowing while debugging

The library gives the power to temporarily unbind method names with their implementation already written in code. This allows us to treat each call to a function as prorogued; this is achieved by employing the decorator `prorogued_method` as showed in code listing 4.5. This allows the programmer to temporarily exclude some code from the execution, returning the expected value directly. This is helpful when trying to locate the source of an error.

```
class ShadowingExample(metaclass=PPLEnableProroguedCallsStatic):

    @prorogued_method
    def foo(self):
        #some code
```

Code Listing 4.5: Shadowing example

4.2 Study

To study how the prorogued paradigm influences the development of software, we performed the following study.

4.2.1 Study goals

This study inquires how programmers use prorogued Python, how they interact with the library, and whether this extension to the Python language helps reduce the programmer's mental strain. In addition, we want to know whether using prorogued programming in Python makes it easier to solve programming tasks than vanilla Python. We formulate the following study hypothesis.

Study hypothesis: Prorogued Python makes it easier to develop software in Python, reducing the mental strain of the programmer and renders the process more efficient.

Study methodology: We perform a qualitative analysis based on a small survey, filled after participants have been exposed to Prorogued Python. We expose participants to Prorogued Python through a small programming task.

4.2.2 Study setting

We select 9 participants who have proficient knowledge of Python and followed at least a programming course at university level. This ensures that participants are somewhat homogeneous.

We provided participants with the programmer manual and a small introduction that provides the basics of prorogue programming.

Participants then need to program one programming task based on the game tic-tac-toe.

After completing the task, we ask participants some questions (see appendix C). We ask some multiple-choice questions and mostly open-ended questions. For each question, we also specify what information we are trying to collect; this is described in the appendix.

We keep the possibility to ask participants follow up questions depending on the participants' responses.

Data collected for analysis included the code written by the participants and the survey's responses. We analyze open questions with Inductive Categorization [18], all text parts related to specific phenomenon, e.g. debugging, are marked; and analyzed.

4.2.3 Programming task description

For the programming task, we already provide a template containing some program functionality, mainly the user interaction and game initialization. The skeleton already contains some calls to functions that need to be implemented. Moreover, we provide a test suite that tests the implementation of all functions that we expect participants to solve, and the skeleton also depends on. Finally, participants are free to add any function or class that they retain necessary for the task.

Tic Tac Toe

For the Tic-Tac-Toe task, we already provided a skeleton that allows displaying the board on the terminal. The task of the participants of the study is to implement the code that handles the game logic.

4.2.4 Study results

We presented an overview of the collected responses and drew some conclusions based on the inductive categorization analysis [18]. All collected responses can be found in appendix D.

Most of the study participants found that prorogued programming benefits the development process, but not all of them found a benefit concerning the

study's task. However, a few participants think that our tool is helpful when programming bigger programs.

We now highlight the study response by area. Double quotes enclose specific quotations in the rest of the section. They may seem anecdotal, but they provide a semantic unit that concertizes and illustrates the participants' responses, as described in [18].

Prorogued advantages

Participants have identified as advantages the following points.

First, prorogued programming allows compartmentalizing the development process by "focusing on the current problem and not spending time on future ones". This thought is also underlined in the following quote "do not need to stop implementing a method to code some subroutine, which could make [me] forget some things about the original method".

Moreover, an identified benefit is that prorogued programming helps in understanding the general structure of the programs, either because the code is complicated or because the code was partially written by someone else already.

The paradigm and its implementation of the library do no complicate the development process; as participants pointed out, a programmer can write all functions before running the code as they would do in vanilla Python.

Only 2 participants think it would be easier to write a stub function instead of leveraging the prorogued paradigm.

Debugging and testing

Regarding the execution of a prorogued program, participants found it helpful to test the program by modifying returning values on the fly. Furthermore, experienced programmers stake that the library "grants fine control of the values of the variables at run-time, allowing to dynamically modify control flow and check edges cases" of the program.

Only a small influence on the debugging process where noted; some participants found an indirect help to debug the program since some program values are printed at each prorogued call.

UI

The interaction with the library presents some negative opinions. The majority of the participants found that the interaction could be smoother. Two main issues can be identified. The first problem is given by using an external window to interact with the library, which shifts programmer's sight to an

external view and does not spatially integrate with the code location that generated the prorogued call.

The second problem is the return value; writing a complex data structure via a text field is a bit tedious, for example, writing an array in an extensive form. No suggestion to improve this is made.

IDE integration

Most of the participants agree that integrating the interaction with the library directly into the IDE "would be beneficial because it would make more seamless the injection of return values in the code". Additionally, this integration would help programmers identify the program point where the prorogued call was made and line identification missing in the current status.

Improvements

Three improvements were suggested by at least two different participants each. Comments about these suggestions can be found later. One is to show what the prorogued function expects as output, like some typing information, by analyzing the code, which calls a prorogued function. The second suggestion is to have some functionality that allows proroguing already implemented methods such that it is possible to override their implementation; this suggestion was implemented after the study was over. The last suggestion is to add the functionality to inject multiple return values simultaneously by specifying an input-output mapping via a table.

Being able to show what the prorogued function expects as output is a complex problem in Python; since Python is a dynamically typed language, no information is natively available regarding the actual function type. Therefore, to implement this feature, we should perform some analysis regarding how the output of the prorogued function is used; typing annotations could help the process, but still, those need to be enforced. Although further work needs to be done, a starting point is given by the paper of M. Hassan et al. [7].

4.2.5 Study conclusions

Recall that the study hypothesis was to validate whether Prorogued Python makes it easier to develop software in Python and whether it reduces the mental strain of the programmer and renders the process more efficient.

The first thing to notice from the participants' response is that no additional burden is given by using Prorogued Python since it can fall back to vanilla Python. From the responses analyzed in subsections, *prorogued advantages* and *debugging and testing*, we can infer that Prorogued Python provides some

easing to the development process and reduces by a bit what the programmer needs to think about while coding.

No conclusion can be drawn from this study regarding whether the development process with Prorogued Python is more efficient than vanilla Python. We can state that Prorogued Python does not hinder productivity; making a statement about efficiency by reasoning over a small task is too prone to errors; some participants wrote that it is difficult to judge with such a small task.

Chapter 5

Conclusion

This thesis aimed to create a prototype implementation of prorogued programming in the Python programming language. To support prorogued function calls, the implementation did not change the Python interpreter but instead uses Python's support for metaprogramming via metaclasses. We implemented the core functionality of a prorogued programming language, adhering to the three basic principles described by Afshari et al. in "Liberating the Programmer with Prorogued Programming" [3]: proroguing concerns, the ability to defer a concern, to focus on and finish the current concern; hybrid computation, the ability to involve the programmer as an integral part of the computation; executable refinement, the ability to execute any intermediate program refinements.

Instead of explicitly mark what function call should be prorogued, as done in Prorogued C# [3], we do that implicitly; we leverage Python's internal attribute resolution mechanisms to label each undefined function call as prorogued call. This is an analogous approach as done by O. Callaú and É. Tanter, in "Programming with Ghosts" [5], where they label each undefined entity as ghost.

We showed how this implementation provides some benefits when programming with Prorogued Python. From the responses analyzed in subsections, *prorogued advantages* and *debugging and testing*, we infer that Prorogued Python provides some easing to the development process and reduces by a bit what the programmer needs to think about while coding.

Still, this implementation remains only a prototype, so it is not ready to be used as a daily tool. More work needs to be done, such as implementing an IO store to cache prorogued return values between different program executions or lazy evaluation of prorogued calls. Both functionalities are present in Prorogued C#, and we think they can improve the user experience.

There are two main limitations of the current implementation. The first

is a consequence of the design choice of not making any changes to the Python interpreter: our implementation cannot support local prorogued function calls as described in 3.1.4. The second limitation comes from the user interface; currently, the interaction with the library is made through an external window; this is sub-optimal and does not allow a cohesive programming experience.

Possible improvements to this project would be to develop an IDE extension to directly communicate with the library in the editor while executing the code. Some participants in the survey also suggested this improvement. In addition, this would better represent the information spatially, for example, by overlaying the dialogue box directly at the caller point in the code.

Appendix A

Programmer's manual

This manual describes the functionality of the prorogued programming library in Python. A key feature is to permit the execution of incomplete programs, allowing the programmer to debug and refine the code during the development process.

A.1 Library purpose

This library purpose is to enable developers to work following the prorogued programming paradigm in Python.

To do so, we allow running the program already when not all functions are implemented; when a function call to a not implemented function is made, we capture this and ask the programmer to specify a return value of the given set of inputs.

A.2 How to enable this behaviour

This behaviour can only be enabled for selected Python namespaces (at module level or class level). The following snippets specify this.

For classes you need to pass either `PPLEnableProroguedCallsStatic` or `PPLEnableProroguedCallsInstance` as metaclass attribute.

```
1 class Example(metaclass=PPLEnableProroguedCallsStatic):
```

Code Listing A.1: Enable prorogued behaviour for class Example (static case)

```
1 class Example(metaclass=PPLEnableProroguedCallsInstance):
```

Code Listing A.2: Enable prorogued behaviour for class Example (instance case)

For modules, you need to call `enable_module_level_prorogued_calls` at the beginning of the Python file.

```
from ppl import enable_module_level_prorogued_calls
enable_module_level_prorogued_calls(__name__)
```

Code Listing A.3: Enable prorogued behaviour at module level

A.3 Specification of the prorogued call behaviour

When the prorogued calls are enabled, the following applies. When calling a function, if its definition is not present in the name-space and this feature is enabled; the call is implicitly assumed to be prorogued. So no error is trowed, but instead, the prorogue handler intercepts the call.

- `PPLEnableProroguedCallsStatic`: sets that prorogued calls have the same behaviour for all instances. The prorogue handler considers only inputs passed to the function.
- `PPLEnableProroguedCallsInstance`: sets that prorogued calls could not have the same behaviour for all instances. The prorogue handler considers inputs passed to the function and instance's attributes.
- module-level: the prorogued handler considers only inputs passed to the function.

A.4 Prorogued handler - interaction with the programmer

The prorogued handler prints on a window that a prorogued call was made and asks the programmer to provide a return value for the given inputs.

Informally, we allow returning any Python basic type and basic containers (dictionary, set, lists) of basic types and local function variables, object attributes and class attributes.

Formally, allowed output need to respect the rule `expression`.

$$
\begin{aligned}
\langle expression \rangle &::= \langle tuple \rangle \\
&\mid \langle list \rangle \\
&\mid \langle dict \rangle \\
&\mid \langle set \rangle \\
&\mid \langle singleton \rangle \\
&\mid \langle atom \rangle \\
\\
\langle atom \rangle &::= \langle float \rangle \\
&\mid \langle int \rangle
\end{aligned}
$$

| True
| False
| None
| ⟨*escaped_string*⟩
| ⟨*local-var*⟩
| ⟨*instance-var*⟩
| ⟨*class-var*⟩

⟨*singleton*⟩ ::= '(' ⟨*expression*⟩ ',)'

⟨*expression-list*⟩ ::= ⟨*expression*⟩ ',' ⟨*expression-list*⟩ | ⟨*expression*⟩

⟨*tuple*⟩ ::= '(' ⟨*expression-list*⟩ ')'

⟨*list*⟩ ::= '[' ⟨*expression-list*⟩ ']'

⟨*set*⟩ ::= '{' ⟨*expression-list*⟩ '}'

⟨*kvpair*⟩ ::= ⟨*atom*⟩ ':' ⟨*expression*⟩

⟨*kvpair-list*⟩ :: = ⟨*kvpair*⟩ ',' ⟨*kvpair-list*⟩ | ⟨*kvpair*⟩

⟨*dict*⟩ ::= '{' ⟨*kvpair-list*⟩ '}'

⟨*local-var*⟩ ::= ⟨*identifier*⟩

⟨*instance-var*⟩ ::= 'self.' ⟨*identifier*⟩

⟨*class-var*⟩ ::= ⟨*identifier*⟩ '.' ⟨*identifier*⟩

In the grammar, the rule `<identifier>` correspond to any Python identifier described by Python's syntax.

We distinguish the 3 different cases for variable binding.

- `<local-var>`: local function variable
- `<instance-var>`: variables part of object that contains the prorogued function. This is possible only when we are using `PPLEnableProroguedCallsInstance`.
- `<class-var>`: variables part of class that contains the prorogued function. This way of accessing attributes works in both `PPLEnableProroguedCallsInstance` and `PPLEnableProroguedCallsStatic`.

A.4.1 User interface

To not interfere with command-line programs, we handle the IO of the library via a separated window.

In the top panel (red rectangle), you can find all messages from the library; in the middle (green rectangle), there is information regarding variables of

object's instances, such that when `PPLEnableProroguedCallsInstance` is enabled, you can access the internal values of the prorogued function context. On the bottom (blue rectangle), there is the input field for the function's return value.

The UI expects the user to insert in the input field the return value of the prorogued function call displayed on the top panel. Then, using the enter key to confirm the input and communicate to the library to continue the program's execution using the provided return value.

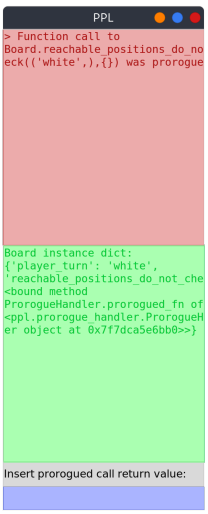

Figure A.1: PPL dialog window PPLEnableProroguedCallsInstance

It is best used alongside the current editor, as showed in figure A.2.

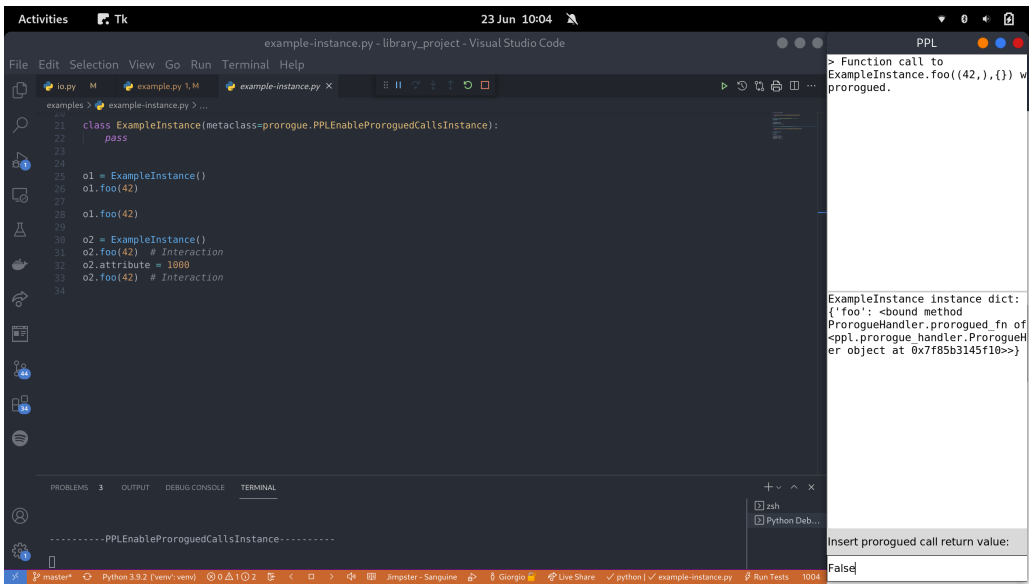

Figure A.2: PPL dialog window alongside editor

Appendix B

Study handout

B.1 Introduction

We want to perform a qualitative analysis on prorogued programming in Python and compare it to vanilla Python. This study is part of G. Piatti's Bachelor Thesis at ETH: "An Implementation of Prorogued Programming in Python".

Data will be anonymized after the study is over. No personal information is collected during this process; only questions regarding your experience and code proficiency are asked. In case we need further information about your responses, we kindly ask you to leave us your email address.

We ask you to read the description about prorogued programming and then to code one programming task and fill a survey at the end.

B.2 Prorogued programming paradigm

A prorogued programming language supports three basic principles:

- proroguing concerns: the ability to defer a concern, to focus on and finish the current concern;
- hybrid computation: the ability to involve the programmer as an integral part of the computation
- executable refinement: the ability to execute any intermediate program refinements

One of this paradigm's key features is to permit the execution of incomplete programs, allowing the programmer to debug and refine the code during the present development process.

To do so, we allow running the program already when not all functions are implemented; when a function call to a not implemented function is made, we capture this and ask the programmer to specify a return value of the given set of inputs.

You may read the programmer manual to gain insights into the implementation. We already enabled the prorogued programming paradigm, so you don't need to add any metaclass attribute.

B.3 Task's description

You need to develop a command line program based on the provided skeleton that allows 2 players to play the game tic-tac-toe.

Game rules: Each player places a marker on the board for each turn. The player who successfully places 3 markers in a vertical, horizontal or diagonal row is the winner.

Instructions Open the template `tic-tac-toe-ppl` and start the task by gain help from the prorogued programming paradigm. First, you will find a README file that explains some details regarding the provided template. After you can play tic-tac-toe on the command line and pass the tests, please zip the files.

The game board is represented by a list of length 9, where elements are from left to right, top to bottom. Elements of the list can be either numbers from 1 to 9, or `'X'` and `'O'`; numbers represent an empty cell, they act as a placeholder.

The main routines `play` is implemented and calls the following functions:

- `board_is_full`
- `get_player_marker`
- `turn`
- `check_win`

The function `play` prints the board, performs a game turn and checks if someone won.

We expect `turn` to perform a game turn; given a player marker, it should ask on the command line in which cell the new marker should go (via standard Python input), and if the cell is free, put the marker, otherwise, ask again. We suggest the usage of the following functions, which also need to be implemented:

- `put_marker`

- `is_cell_free`

To summarize, the following function are expected to be implemented:

- `put_marker(self, board: 'board', cell: int, marker: 'O|X') -> 'board'`
- `is_cell_free(self, board: 'board', cell: int) -> bool`
- `get_player_marker(self, player_id: '1|0') -> 'X|0'`
- `check_win(self, board: 'board') -> 'X|O|None'`
- `board_is_full(self, board)`
- `turn(self, board, xo) -> 'board'`

B.4 Survey

After you solve the task, please fill the survey and upload the zip file when asked.

Survey link: https://forms.gle/hMdFRcJjLvLThHja7

Appendix C

Survey questions

C.1 General demographic question

1. Please indicate the level of experience in Python:
   - beginner
   - intermediate
   - advanced
2. Please indicate the level of experience in languages different from Python
   - beginner
   - intermediate
   - advanced
3. How many years of coding experience you have?
4. What is your level of education? Please specify degree and title
5. How many years did you professionally worked as a programmer (or related jobs, where programming was part of the job)?

C.2 Questions

7. Do you think that prorogued Python made it easier to program this task, considering the overall experience?
8. Please explain why it easier with a few sentences.
9. Did you find it helpful to execute partially written programs and inject return values at run-time? Please explain with a few sentences. *We want*

to address if being able to run partially written programs helps the development process.

10. Do you think that prorogued Python allows you to focus better? Please explain with a few sentences. *We want to investigate if prorogued programming facilities the development process.*
11. Do you think that using prorogued Python renders the development process more complicated? Please explain with a few sentences. *We want to identify if there is any drawback.*
12. Did you find easy injecting return values at run-time? Please explain with a few sentences. *We want to gather some opinions regarding the user experience with the library.*
13. Did you think it is easier to debug a program with prorogued Python? Please explain with a few sentences. *We want to investigate whether prorogued Python helps to debug the code.*
14. Would you change something in the user interface for prorogued Python? Please explain with a few sentences. *We want to gather some opinions regarding the user interface of the library.*
15. Assuming there is better integration with the IDE with which you can directly see which line the prorogued call was made and directly write the return value. Do you think this interface would be beneficial? *We want to identify possible extensions to the current implementation.*
16. What extension to the implementation do you think would be useful, if any? *We want to identify possible extensions to the current implementation.*

Appendix D

Study responses

Participant A

1. intermediate
2. intermediate
3. 4
4. BSc Mathematics
5. 0
6. No
7. The task was quite an easy one, one could solve new problems quite fast, so the prorogued programming was not needed, on a harder task I can see it being very helpful
8. yes, it's useful to be able to simulate a function result even when the function has not been implemented yet, one is able to focus on current problem and not spend time for the future ones, especially if the code is written by more than one person, in this way one doesn't have to wait until other people solve their part
9. yes, it allows to focus on the current problem and not to think about the future ones
10. it doesn't, if one wants he could still develop in the same way as usual and writing all the functions before running the code
11. yes, the UI is very intuitive
12. Yes, it is easier to see what is happening when a function is called and what the inputs are
13. being able to customize the positions of the UI elements and changing their size would be a good feature

14. yes, but I would still show the inputs of the function and not only the value to insert. (Note: overlay also with input value w
15. -

Participant B

1. beginner
2. intermediate
3. 0
4. BSC CS
5. 0
6. yes
7. Its very helpful to be able to run the whole program before having implemented all of it, for example to test if a functions works properly
8. Yes, it allowed me to make sure the program was working correclty at all points
9. Yes you don't have to worry about functions that you haven't implemented yet
10. It was a bit annoying to have to re run the environment setup if I started the program from a new terminal window, but there are workarounds to avoid this
11. Yes the GUI for injecting values is very self explaining on how to use it
12. Not necessarily, works the same way as normal debugging I would say
13. it is not aesthetically pleasing but it is pretty easy to use, maybe add something that tells you what kind of values that function is expected to return
14. it would be more comfortable to use but probably not necessary
15. Maybe to inject multiple different return values in one go

Participant C

1. intermediate
2. beginner
3. 5
4. finishing Bachelor CS
5. 0

6. Yes
7. Because I suck at other languages
8. It helps to get a feeling of the main program (play) before starting actually coding (like a debugger, one can see which function calls are made). So as to get a quick idea about the general structure of the program and, without the function it would probably just crash. It did not help me to implement the functions themselves.
9. .

   It was of no use in this tiny example, because I was able to keep track of the only four functions I needed to implement (for which I used the Handout, last page). I don't know for larger tasks.
10. It was of no use in this tiny example, because I was able to keep track of the only four functions I needed to implement (for which I used the Handout, last page). I don't know for larger tasks.
11. No, why would it. Just don't look at the second screen if you don't want to M
12. You gotta first find out the return type (look at the handout) and then understand what the arguments mean (which should be explained in the handout), once you figure that out, it's easy
13. Well technically yes, but only because it has an GUI and I suck at using a debugger M
14. I mean, could be prettier, but does it's job.
15. Sure yes. Refer to my answer about the general benefits of this prorogued programming; it helps in understanding the general structure of the program.
16. -

Participant D

1. intermediate
2. intermediate
3. 6
4. BSc CS
5. 0
6. yes
7. I could test parts of the code without needing to already have all methods implemented

8. I find it helpful, it allows me to experiment many different possible behaviours of the program without needing to implement all methods.
9. Yes because it allow to compartmentalise my coding process and I don't need to stop implementing a method to code some subroutine (which could make me forget some things about the original method).
10. No, on the contrary it makes it somewhat simpler in the sense that I can focus on implementing a specific method without having to worry about sub-routines and similar (analogue to what said before about focus).
11. Generally I find it quite easy, however in some cases it could be a bit convoluted, especially when some unimplemented method returns some complex data structure.
12. Yes because it grants me fine control of the values of the variables at run-time, allowing me to dynamically modify control flow and/or check edge cases. To that end it would be useful to have the ability to prorogue already implemented methods to override their return values so that we have even more control on the run-time value of variables.
13. It would be nice to have indicated the expected return type of the prorogued method.
14. I think it would be beneficial because it would make more seamless the injection of return values in the code, given that having to tab between different windows could be a bit cumbersome sometimes.
15. The ability to prorogue already implemented methods.

Participant E

1. intermediate
2. advanced
3. 3-4
4. CSE BSC 2nd year
5. 0
6. yes
7. Possibility to skip function implementation and implement later on (when for example we have a better idea for the implementation)
8. Yes, so we can test each function separately without re-running the program every time
9. I don't have enough experience to say something about it. But I think the vertical bar next to the editor allows a better focus experience. (Note:

it's a comment regarding how to arrange the ide and the prorogued window)

10. No, I don't think so. The programmer doesn't have to put much effort to use this paradigm (it's only written in class parameter)
11. Yes, there is nothing easier as a simple text input field in the PPL window ;)
12. I have never used a debugger for Phyton, but compared to other debugger I think that prorogued Python is easier. This because it's not required to set any breakpoints as it's used in other debuggers.
13. No. I think that the idea of the three different parts is good
14. Yes, it can be useful, but I think it's not a necessary thing. The negative point may be the presence of a lot of text when there are multiple prorogued calls that can distract the user.
15. -

Participant F

1. intermediate
2. advanced
3. 5
4. BSc CS
5. 0
6. yes
7. I could run not yet complete programs, this was an nice point since usually you need to wait till the end of the coding process to test something.
8. Yes. As I already mentioned I find quite helpfully not to wait till all the functions were coded to be able to test the program.
9. I don't have any take on this.
10. No it does not. There's is a fallback to normal Python in any case.
11. It's a bit cumbersome, the UI could be a bit more integrated in the editor.
12. Yes, interactively changing return values of functions is very helpful.
13. No.
14. Yes. This is a nice idea. This could really improve the development process and provide a more cohesive programming experience.

15. -

Participant G

1. beginner
2. beginner
3. 1
4. CS BSc 1st year
5. 0
6. No
7. I felt very similar to the test phase, having already tested parts of the programme I did not encounter any real difficulties.
8. Yes, I found it very comfortable to work this way, because the code was more understandable and easier to model.
9. It could certainly give me advantages depending on what I'm doing. for example, if the project was very big, I would need it.
10. no, in the end it can simplify and that's it, if you really don't want to use it just work as if it wasn't there
11. yes, I find it good, it makes process more efficient and simpler
12. yes, we can work on small parts of the project with precise values that we want and therefore it is easier to identify and correct a bug
13. no
14. in my opinion it would be more useful because if you have the line it is easier to relate it to the code
15. I don't know

Participant H

1. intermediate
2. intermediate
3. 3
4. in progress BSc CS
5. 0
6. No
7. I think it's a great feature but only when you need to input a small amount of unknown values in the whole program otherwise it becomes

tedious to switch between the console where you give one input and the window where you give the prorogue solution when you need multiple values but not sequentially. Could be crucial at the beginning to test single functions but there has to be a way to insert all the values that will be asked in the prorogue window at the same time and not having to constantly switch between the windows. I think it would work best if you could test just a single function with predetermined values instead of working with the whole program at the same time otherwise the testing may be faster by running the part of the program with fixed values in a new file as a testing ground.

8. I think executing partially written programs is essential however this is already partially possible in python with a jupiter notebook where you can run parts of a program as well as fix values. It would help a lot in other programming languages though where such a thing isn't possible yet.

9. not really for me, but it may be because I'm already used to the way programming requires me to give it values to test the program before I have written the sections so in a way I'm already doing prorogued programming without the add-on window, though the problem with this is that I have to fix every value before the runtime and can't just decide the value when I need it.

10. Not really, I believe it to be a nice to have but not necessary for the program function.

11. Yes, but it would be nicer to be able to inject multiple values at the same time especially if the program requires you to give a lot of console input like the tic tac toe. An example where it was tedious is when I was checking if the player turns were working and I had to constantly switch windows after I put in the cell the player chooses I had to give in if it was free or not (easy) and then if a player won that turn (tedious because I had to switch from when I put True in prorogued to the console and then back to the prorogued to give in False/ None that no player won yet before I could select a cell for the next player)

12. sure, but the way I program i usually try to pre-implement all the necessary functions with fixed return values as a skeleton before I start implementing them so I ended up not needing it for the most part and when I did need it, it felt kind of tedious even if it helped (wished I could fix some values directly without having to reenter them every loop, this can be hardcoded but then the whole point of prorogued programming kindof falls away)

13. Yes I feel like the program should try to see what variables/ inputs it will need and suggest you to give values for them and decide if you

need them to stay the same (either for the same execution or even when you rerun the whole) and the ones you leave empty will be the only ones asked of you every time so you can accurately test what you want to test without having to reenter the values that will stay the same for most of the execution every single loop iteration

14. Sure, sounds good/ useful but it would still be good it you can fix certain values like mentioned above
15. -

Participant I

1. intermediate
2. advanced
3. 6
4. BSc CS
5. 1
6. No
7. Task didn't really require it, was tedious to write out an array of the whole board, it just appeared blank after I had implemented part of the program. Debugging with print()s was faster and easier
8. I could imagine that it would be useful, but it makes more sense in a debug mode. Again, I don't think this task was hard (or non-linear) enough to really require it.
9. Yes, then I don't need to make dummy functions that e.g always return True. However, if I'm writing a large program, it would be easier to make these dummy functions than to have to manually input the return value.
10. Yes, I though it isn't necessarily a bad thing (of course adding a feature is more complicated). In particular it was a pain to jump in and out of the window.
11. With an unfamiliar program, would be helpful to see what kind of value it's expecting. Even just bool or int. Not really possible in Python I guess
12. Yes, as I mentioned you won't need dummy functions.
13. Probably integrate it into the command line like a proper debug tool. Not nice to jump in and out of vscode to do it. Otherwise it's fine.
14. Yes, much better. Like I mentioned, the main problem is having to jump in and out of the IDE.

15. Breakpoints, a shell. Basically to integrate it into a normal debug environment.

Bibliography

[1] `__getattr__`. URL: https://docs.python.org/3/reference/datamodel.html#object.__getattr__ (visited on 05/03/2020).

[2] `__setattr__`. URL: https://docs.python.org/3/reference/datamodel.html#object.__setattr__ (visited on 05/03/2020).

[3] Mehrdad Afshari, Earl T. Barr, and Zhendong Su. "Liberating the Programmer with Prorogued Programming". In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2012. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 11–26. ISBN: 9781450315623. DOI: 10.1145/2384592.2384595. URL: https://doi.org/10.1145/2384592.2384595.

[4] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[5] Oscar Callau and Éric Tanter. "Programming with ghosts". In: *IEEE software* 30.1 (2012), pp. 74–80.

[6] *Functional Programming HowTo*. URL: https://docs.python.org/3/howto/functional.html (visited on 03/25/2020).

[7] Mostafa Hassan et al. "MaxSMT-Based Type Inference for Python 3". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 12–19. ISBN: 978-3-319-96142-2.

[8] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. "Developing and Debugging Algebraic Specifications for Java Classes". In: *ACM Trans. Softw. Eng. Methodol.* 17.3 (June 2008). ISSN: 1049-331X. DOI: 10.1145/1363102.1363105. URL: https://doi.org/10.1145/1363102.1363105.

[9] *PEP 3129 Class decorators*. URL: https://www.python.org/dev/peps/pep-3129/ (visited on 05/03/2020).

[10] *Python - Data model*. URL: https://docs.python.org/3.9/reference/datamodel.html (visited on 05/03/2020).

[11] *Python Glossary - Duck Typing*. URL: https://docs.python.org/3/glossary.html#term-duck-typing (visited on 03/26/2020).

[12] *Resolution of names*. URL: https://docs.python.org/3/reference/executionmodel.html#resolution-of-names (visited on 05/13/2020).

[13] Guido van Rossum. *What is Python? Executive Summary*. URL: https://www.python.org/doc/essays/blurb/ (visited on 03/25/2020).

[14] Peter Van Roy. *Programming Paradigms for Dummies: What Every Programmer Should Know*.

[15] Hesam Samimi et al. "Declarative Mocking". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ISSTA 2013. Lugano, Switzerland: Association for Computing Machinery, 2013, pp. 246–256. ISBN: 9781450321594. DOI: 10.1145/2483760.2483790. URL: https://doi.org/10.1145/2483760.2483790.

[16] Hesam Samimi et al. "Declarative mocking". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, pp. 246–256.

[17] KR Srinath. "Python–the fastest growing programming language". In: *International Research Journal of Engineering and Technology* 4.12 (2017), pp. 354–357.

[18] Josh Tenenberg. *Qualitative Methods for Computing Education*. Ed. by Sally A. Fincher and Anthony V.Editors Robins. Cambridge Handbooks in Psychology. Cambridge University Press, 2019, pp. 173–207. DOI: 10.1017/9781108654555.008.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

An Implementation of Prorogued Programming in Python

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Piatti	Giorgio

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 02.09.2021	G. Piatti

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.