PyDeclarative: a declarative Single Page Application framework for Python

Stanislaw Adaszewski[1]

Abstract

The tools and practices for software development have made substantial improvements over the course of the past decades. The subsequent generations of languages, frameworks and associated coding practices have evolved to improve readability, maintainability and to make the code less error-prone. One of the increasingly prominent paradigms in the latest generation of software development ecosystems is the declarative coding style. From Single Page Applications (SPA) with React, through Infrastructure as Code using Terraform, to declarative idioms in Scala and Haskell, this technique has been increasingly successful in enabling fast development of industrial quality software at scale. In this paper, we introduce PyDeclarative[a] – a new SPA framework for Python which inherits the best elements of Shiny/Shiny for Python and Qt Modeling Language (QML) in order to allow inline mixing of UI and logic elements; seamless reactive programming with minimal added verbosity; and compositional syntactic uniformity which delivers unparalleled flexibility in Dependency Injection/Inversion of Control mechanisms and refactoring capabilities. We compare our framework against state of the art alternatives in terms of features and performance.

Introduction

The landscape of declarative programming [1–12] is vast and ranges from theoretical works to practical implementations, the latter including both rigorous Turing-complete systems and more specialized domain-specific languages (DSLs) which mix the declarative and imperative paradigms for the sake of practicality. While a comprehensive overview of the concept of declarative programming

1 s.adaszewski@gmail.com

a https://github.com/sadaszewski/pydeclarative

lies beyond the scope of this paper, we would like to provide certain key references for better contextualization of the present research. Egri-Nagy argues[3] that declarativeness is not a particular style but the core idea of both programming and mathematics and discusses the implications of this viewpoint for the teaching of both disciplines, using Clojure[13] for illustration purposes. Torgersson[2] compares weak and strong declarative programming, discusses the challenges with widespread adoption of the declarative style and proposes the directions for the next generation of definitional languages. Multiple publications discuss successful applications of the declarative languages and frameworks, for example Curry for database programming[14,15], multiple frameworks for AI systems[4–6,16], Terraform[17], CloudFormation[18] and Kubernetes[19] for reconciled deployment, KATENA for blockchain[20]. For numerous recent practical examples consider the Practical Aspects of Declarative Languages series[21–25]. Of particular relevance in the context of this manuscript are the uses of declarative approaches to user interface programming, e.g. React[10], XForms[26], TkGofer[27], QML[28], Streamlit[29], Dash[30], Shiny[31], Shiny for Python[32]. We use some of these examples to position PyDeclarative with regards to the level of declarativeness, compositional strategy, runtime partitioning between the client-side and the server-side and other characteristics, such as – critically – the implementation of reactivity.

Reactive programming[33,34] is a concept central to declarative programming of graphical user interfaces which are naturally event-driven. The reactive approach allows the programmer to interact with the event-based framework in a fully- or semi-declarative manner, often eliminating even an indirect awareness of its existence. Key aspects of reactive programming include propagation of change and dependency tracking. The survey[33] proposes six dimensions in the taxonomy of reactive programming. PyDeclarative is implemented in Python[35] – a language which is not intrinsically reactive – and relies fully on the standard syntax (i.e. PyDeclarative is not an embedded DSL) and therefore we focus on comparisons against similar frameworks. Nevertheless the proposed dimensions – basic abstractions, evaluation model, glitch avoidance, lifting operations, multidirectionality, support for distribution – are as pertinent and can be evaluated as easily in the context of frameworks, as in the case of reactive languages. Implementation details of change propagation can involve several interesting concepts such as topological sort[33], delta evaluation[36], memoization[37], lazy evaluation[38], differential dataflow[39], constraint propagation[40], dataflow analysis[41]. Several prominent reactive programming models deserve an explicit mention. Reactive streams[42] refer to asynchronous stream processing with non-blocking back pressure. Functional Reactive Programming[43] (FRP) combines the statelessness of functional programming with reactive processing of asynchronous data streams. Lack of a mutable state makes it easier to reason about such setups even in the presence of complex

behaviors. Reactive scheduling[11,44] refers to abstracting away the management of threads and concurrency through the use of schedulers which decide automatically when and on which thread a task should be executed. For an interesting example of reactive programming without functions consider Oeyen et al. [45]. Lastly, we feel several practical implementations of reactive programming are worth mentioning – RxJava[46], RxJS[47], Bacon.js[48], MobX[49], ReactiveUI[50], Elm[51], Svelte[52], Vue.js[53].

Methods

PyDeclarative Syntax

PyDeclarative is built around the idea of using Python's class declaration syntax to represent a hierarchical structure of components similar to the one found in QML[28]. Differently than in QML, the components do not have an id field and the class names represent identifiers that can be used to reference the respective components. Class attributes are used to represent static properties (unless they are holding an instance of the Binding class, in which case they are lifted to be a dynamic property). Properties defined using the @property decorator represent dynamic properties (i.e. properties whose values are updated whenever the values used to compute them change). Lastly, the LazyBinding class allows a class attribute to be provided with a binding coming from the runtime scope. Since QML is more of a framework than a dedicated language with its own parser, differently to QML, the scoping rules are implemented at runtime only. At declaration time, only the regular Python scoping is possible. LazyBinding allows to delay the initialization until the runtime scope is available. Nested classes represent child components. Methods represent either functions or event handlers if they start with the 'on_' prefix. They are automatically connected to the corresponding signals which by convention end with the 'ed' prefix (e.g. clicked, added, etc.) The runtime scope object (instance of class Scope) is available as the first parameter to functions and event handlers, as well as through the Binding and LazyBinding classes. At runtime, the hierarchy described above gets instantiated so that the components are translated into nodes (instances of the Node class) representing the DOM nodes which are subsequently created in the browser. Components which do not need any visualization in the DOM are represented as empty elements. Each instance of the Scope class is associated with a node. The scoping rules are the following: first the members (properties, functions) of the associated node are searched. If not found, the identifier is assumed to be a component identifier and the corresponding node is searched for, first among all the descendants of the current node (in a breadth-first search), then in subsequent parents and their descendants (in a breadth-first search at each level). This behavior does not mimic the behavior of QML scope very well, as the latter uses individual QML files as scoping

units. Due to the nature of PyDeclarative, this would have resulted in added complexity, therefore in this iteration, we rely on the runtime scope. The built-in library of components is based on Bootstrap[54] and includes text input/output controls, buttons, links, images, tables (using Pandas[55] and DataTables[56]), plots (using plotly.js[57]), tabs, modal dialogs, notifications, file upload/download controls, as well as foundational components such as the Repeater which implements dynamic model-based instantiation of delegates. Furthermore, utility components such as Timer and Connections are available. The latter allows to automatically establish signal-slot connections in components other than the current one.

PyDeclarative Architecture

The central element of PyDeclarative is the DeclarativeEngine class. It is responsible for the creation and the destruction of node hierarchies, along with the triggering of the associated signals. Furthermore, asynchronous tasks can be created through the engine, ensuring that they will be terminated when the associated browser session ends. In addition to that, this class is responsible for sending and reception of the messages to/from the DOM and for notifying the appropriate nodes on the server side. The Node class represents instantiated components and the DOM nodes. It features a uuid attribute which corresponds to the id attribute of the given component in the DOM. It allows for easy two-way communication between a component's client- and server-side counterparts. Nodes use the html property of the underlying components to generate the markup that is then sent to the browser through a WebSocket[58,59] upon the application startup. Each instance of the Node class holds a list of properties along with their corresponding generative expressions (if they are dynamic). The Property class holds references both to all of its dependencies and to all of its dependents. These relationships are managed by the Scope class which updates the property that is currently being computed (the CURRENT_DEPENDENT context variable) whenever it successfully resolves an attribute request to another property. Furthermore, if a static value is assigned to a dynamic property, the lists are cleared accordingly. The htmldiff function is responsible for the computation of the incremental updates of the DOM elements. The logic behind it is to communicate only the minimum subset of changed/added/removed attributes and/or children of a node. Its counterpart on the client side (the applyDiff function), interprets these recipes and updates the nodes to reflect the current state of each element with minimal amount of modifications being executed. Last but not least, the server module is responsible for the delivery of the static content (HTML, JavaScript, CSS), as well as for the WebSocket connections. The Handler class in the server module is responsible for the setup of the connections between the DeclarativeEngine instance and the WebSocket input/output queues. The

format of both the afferent and the efferent messages is designed in such a way, as not to require any translation, therefore the server module essentially functions as a dispatch mechanism.

Work in progress...

Comparisons and Benchmarking

We compare the characteristics of PyDeclarative to Shiny for Python[32], Streamlit[29], Dash[30], Gradio[60], Anvil[61], Panel[62], QML[28] and SwiftUI[63] in the following categories: Programming Language, In-place Logic, Declarative Values, Compositional Syntactic Uniformity, Syntax Resemblance to UI, Scoping Rules and Dynamic Callbacks. In brief, Programming Language refers to the notation used for building the user interface, as well as for implementing the logic governing the interface's behavior and its interactions with the other elements of the programming ecosystem. In-place Logic designates the support for code fragments acting in different roles (e.g. initializers, event handlers, dynamically computed values), interspersed with the user interface description syntax, so that the logic stays as close as possible to the elements that it operates on. Declarative Values describe computed values (and their hierarchies), which are automatically updated whenever the values of their dependencies change due to user interactions, timed events, input/output or other changes in the state. Compositional Syntactic Uniformity is the name we have coined for the ability to use the same syntax for Component specifications irrespective of whether they appear as single-use definitions nested within the hierarchy of another component or if they are declared outside of hierarchies, for reuse as building blocks. Importantly, this results in tremendous refactoring flexibility, where entire chunks of UI and logic syntax can be redistributed at will without changing their content or the way they communicate with the remaining structures. Syntax Resemblance to UI aims to convey the idea of how the UI will look and behave by merely looking at the code. This can be achieved for example by syntax-level support for nesting components, eliminating the need for any boilerplate code and enabling concise expression of both dynamic and static content, attributes and behaviors. Scoping Rules determine where variables are declared and accessible. Well thought-out scoping rules can simplify collaborations between components, as they minimize the necessity to explicitly pass around the references. Lastly, Dynamic Callbacks refer to the ability to manipulate callbacks at runtime using the same or similar declarative syntax as for the other tasks. It is an important prerequisite for convenient dynamic component creation and linking of the newly created components to the existing ones.

As the next step, we evaluate the above frameworks in the six dimensions of reactive programming proposed by Bainomugisha[33] - Basic Abstractions, Evaluation Model, Lifting Operations,

Multidirectionality, Glitch Avoidance and Distribution support. In brief, Basic Abstractions refer to the declarative primitives of the given framework, which usually differ in their naming, function and interactions. Evaluation Model determines how changes are propagated across the dependency graph of values and computations[33]. Lifting Operations define the way normal functions and operators can be modified in order to act on reactive values and expressions. Lifting strategies are divided into implicit, explicit or manual. In implicit lifting, regular functions and operators can act on reactive expressions. In explicit lifting, functions and operators need to be transformed using a lifting combinator before they can act on reactive expressions, whereas in manual lifting, the current value has to be extracted from its reactive wrapper before it can be acted on. Multidirectionality refers to being able to automatically both update the current value of a reactive expression when one of its dependencies changes, as well as to appropriately update its dependencies when the expression is explicitly set to a new value. We employ it in a narrower sense here – of being able to pass the data to and from the UI, using a single reactive variable updated in turns. Glitch Avoidance is a method employed with some push-based evaluation models in order to ensure such an order of updates that the dependent values are always recomputed using only up-to-date values of their dependencies. In other words – invalidated dependencies are always recomputed before their dependents. This avoids the dependents momentarily holding inconsistent values (computed partly using updated and partly using old dependencies) – the so called glitches. In push-based frameworks without a guaranteed order of updates, glitches are admissible and can be eliminated only by avoiding the creation of diamond-shaped dependency graphs. Lastly, Distribution Support refers to the capability of the reactive syntax to describe the interactions of elements living on multiple independent computing nodes and of the framework to ensure the consistency of the dependency graph in such a setup.

The benchmark we employ is loosely inspired by the uibench[64] battery of tests for measuring the performance of browser-based UI frameworks (React[10], imba[65], etc.). The battery consists of: adding and removing to the Document Object Model (DOM) tables with a variable number of rows and columns (100x4, 50x4, 100x2, 50x2), updating the tables from baseline to different states (sorted, filtered, with every n-th row activated), adding/removing boxes and updating their style attribute, adding/removing trees with a variable number of nodes at each level (500, 50x10, 10x50, 5x100, 2x2x2x2x2x2x2x2x2x2x2) and updating trees from baseline to states with added/removed/moved items. We skipped the 4 tests designed specifically to test the worst case performance of Javascript frameworks Kivi[66], snabbdom[67], React[10] and virtual-dom[64]. We implement 92 tests in total and run 10 iterations of the entire battery in order to compute the mean and standard deviation of the

measurements. Last but not least, we use a variant of the tree test where no update is performed to calibrate an additional delay we introduce in the preparation step, before proceeding to the update step. We do this in order to let the browser settle after the initialization step. This delay was set to 250ms, which let us measure the expected values of around 0ms for the “no change” test. Otherwise, for unspecified reasons we saw variable amounts of unexpected time (ranging from 7ms to 200ms) used to execute the “no change” updates. We suspect that the reason for this were garbage collection cycles or the layouting/rendering engines operating in the background and slowing down how fast the browser could report back to the backend that a step has been executed. The lowest observed delay of 7ms on the other hand could likely be attributed to the refresh rate of the monitor used for testing which was set to 144Hz, which translates to 1000ms/144=6.94ms frame duration. This delay could be present due to the mechanics of the window.requestAnimationFrame() (rAF) call which executes the passed callback as soon as the browser is ready to draw a frame but attempts to synchronize its subsequent calls to the refresh rate of the monitor. By introducing a delay before the update step we ensure that the callback passed to rAF is executed immediately in case no updates to the DOM are made and at most with 7ms extra delay in addition to the time used for updating, reflowing and preparing the DOM for painting. Any further delays caused by the internal mechanisms of the browser would be impossible to identify or eliminate in the update step measurements without extensive modifications to the Javascript and layouting/rendering engines, therefore we did not investigate them. We use the Brave[68] browser 1.66.110 with the Chromium[69] engine 125.0.6422.60 and Python 3.9.1 to run all tests. We implement the benchmark for PyDeclarative and Shiny for Python, since the latter is the most closely related competitor in terms of language and features.

Results

In Table 1 we have captured the results of the comparison of the frameworks in the six reactive programming dimensions. In terms of basic abstractions, PyDeclarative has many commonalities with QML, most importantly it shares the signals and slots mechanism and orients its reactive logic around the concepts of properties and bindings. Shiny for Python has a more fine-grained concept of reactive constructs and differentiates between reactive values, calculations and effects. The two former correspond essentially to PyDeclarative’s properties – static and dynamic, respectively; whereas effects would be the closest match to slots, with events being the equivalent of signals. Dash operates on the concepts of inputs, outputs and callbacks, where both inputs and outputs refer to the properties of UI elements, not to general purpose expressions. The data exist in parallel to Dash’s UI and are

fundamentally non-reactive. Reactivity of the UI is achieved through the mechanism of callbacks, however this reactivity does not propagate across the non-UI elements. In other words, the programmer is responsible for implementing any additional required reactivity. Panel, on the other hand does provide general purpose reactivity which can cross the boundaries of UI. The key concepts in Panel are reactive parameters (similar to PyDeclarative's and QML's properties) and expressions. Importantly, Panel's reactive expressions can live outside of the object hierarchy which is a different paradigm than the one employed in PyDeclarative and QML, where even non-UI objects have to belong to an object (typically rooted in an UI element) hierarchy. SwiftUI employs state variables, computed variables, bindings and events to provide similar functionality to that found in PyDeclarative and QML. Streamlit, Gradio and Anvil do not currently provide reactive primitives, with Gradio team discussing such a possibility for the future. We omit Streamlit, Gradio and Anvil in further reporting in this section, as due to the absence of reactive primitives, the remaining dimensions do not apply to these frameworks. Among the reactive frameworks, the evaluation models are almost equally represented, with PyDeclarative, Dash and QML implementing push-based models; Shiny for Python and SwiftUI supporting pull-based models; while Panel employs both push- and pull-based evaluation models depending on whether a reactive expression is watched. We can identify two types of lifting operations in the frameworks under consideration. PyDeclarative, QML and SwiftUI rely on implicit lifting, whereas Shiny for Python and Panel resort to manual lifting. With that said, the manual lifting in these frameworks is minimal in the sense of requiring very little additional syntax (e.g. calling a reactive value using round braces in Shiny for Python in order to retrieve its current value). In fact, PyDeclarative relies on the Scope object which replaces the self argument in methods to handle the lifting. The classification of this behavior could be subject to debate, however we choose to classify it as implicit as it is indistinguishable from what accessing regular values through the self argument would look like in regular Python methods. Since Dash does not provide universal reactive primitives, it does not need lifting operations. We will omit Dash in further reporting in this section, as due to the character of its implementation of reactivity, the remaining dimensions do not apply to this framework. Multidirectionality is a relatively rare capability among the compared solutions, with only QML and SwiftUI featuring variants of this functionality. In both cases, the behavior is not implicit, subject to special considerations and limitations and/or implemented by third party libraries. Glitch avoidance is meaningful only as related to push-based evaluation models, since pull-based models are inherently immune to this behavior. Among the push-based frameworks, none implements proper Glitch Avoidance, which would typically entail a topology sort of the dependency graph before executing the updates. As mentioned, pull-based frameworks benefit from implicit glitch avoidance. Lastly, none of

the projects under consideration, supports distributed execution. With that said, browser-based frameworks could be considered as distributed in the sense that their frontends and their backends may operate on separate nodes. However, we consider this special case to be too specialized and limited in order to be considered as a true distributed execution. We suggest that in these cases the litmus test should be based on the distributed reactive execution of the backends, which is not implemented by any of the solutions.

In Table 2 we have captured the results of the comparison of the frameworks in the general categories. Most of the solutions we considered are based on the Python programming language. This was a deliberate choice, as we wanted to compare PyDeclarative to as many direct competitors as possible. The only non-Python solutions are QML, which uses its own syntax combined with Javascript; and SwiftUI, based on the Swift programming language. We have included the non-Python alternatives, since – as we demonstrate further – they bear stronger resemblance to PyDeclarative in terms of their capabilities and the associated coding style than their Python counterparts. For example, in-place logic support is present only in PyDeclarative, QML and SwiftUI. Strictly speaking, these frameworks admit code fragments in most places throughout the syntax describing the UI and are flexible enough to enable connectivity between one another and to the UI elements regardless of their placement. Of particular importance is the fact that the user is not forced to write event handlers, declarative expressions or utility functions in a different style, outside of the UI hierarchy. Given the convenience of modern text editors, in particular the powerful code folding and navigation facilities, having the ability to define everything where it makes most sense to the developer is paramount. This place is often close to the place where the code is used and almost exclusively there if the code is not reused elsewhere. The latter is a frequent occurrence in UI programming. Declarative values is a feature shared among the aforementioned frameworks and additionally by Shiny for Python and Panel. While Dash offers limited support in this respect with reactivity being limited to UI elements, the remaining frameworks do not employ reactivity principles at all, although most of them (bar Anvil) can be described as declarative in their UI building syntax alone. Compositional syntactic uniformity is a characteristic we deem particularly impressive and important. PyDeclarative, QML and SwiftUI were the only solutions under consideration to share this trait. Syntax resemblance to UI is shared across the spectrum with PyDeclarative, Shiny for Python, Dash, QML and SwiftUI demonstrating strong resemblance, Panel – a middle one, whereas Streamlit, Gradio and Anvil employ syntaxes which don’t resemble UI in the sense of reflecting the hierarchy of components, however (bar Anvil) feature alternative ways to describe UI in an intuitive fashion. PyDeclarative and QML were the only frameworks to feature custom scoping rules, which result in very flexible, dynamic scopes, further

contributing to their compositional syntactic uniformity, which eclipses that of SwiftUI. Lastly, PyDeclarative, Gradio, Anvil, Panel and QML offer direct support for dynamic callbacks, whereas Dash resorts to pattern-matching callbacks, while Shiny for Python and SwiftUI employ a similar technique of storing a reference to the current callback in an auxiliary variable. Streamlit, due to its continuous reevaluation of the entire script on input changes, is not callbacks-oriented.

The benchmark results are captured in Figure 1. Broadly speaking, we observed similar levels of performance across all of the tests between Shiny for Python and PyDeclarative. Shiny for Python does not feature a htmldiff-like function on the server-side. Unlike PyDeclarative, it does not compute partial updates of the DOM which are then transferred to the browser. Its components are defined as explicitly consisting of two layers – a server-side written in Python and a client-side – written in TypeScript/JavaScript using React. Shiny front-end and back-end exchange data rather than representations and it is the role of the front-end elements to decide about the presentation details and compute the pertinent deltas to the DOM nodes that have already been created. The latter part is a key capability of React and implemented very efficiently. This allows for higher theoretical performance but complicates the creation of components, making it a multi-step procedure, requiring expertise in React and making it incompatible with the rule of compositional syntactic uniformity, which effectively mandates the ability to declare a component in a single block that can be moved between the code units without modification. In order to emulate the PyDeclarative syntax in Shiny, we have defined the necessary Shiny helpers using a Shiny module and a ui.HTML renderer rather than using the two-layered approach described above. This solution, coupled with the lack of a HTML diff-ing engine, gave Shiny an advantage in performance since the entire HTML was simply replaced at every update. The second variant of UIBench for PyDeclarative, eliminated the use of the diff-ing engine and resorted to full HTML replacements akin to the Shiny implementation. In this test, PyDeclarative was faster than Shiny in virtually all tasks except for table[100,4]/render, tree[50,10]/render and tree[10,50]/reverse. Importantly, all tests both for Shiny and PyDeclarative had an average execution time below 100ms delivering plenty of performance in these artificially big update scenarios. Unsurprisingly, both client-server frameworks were 1 to 2 orders of magnitude slower than the Javascript-only browser-only frameworks like Imba[65] or React in executing UIBench[64].

Discussion

Compared to other frameworks

Among the Python frameworks under consideration, PyDeclarative demonstrated the most complete implementation of the features captured by the selected general categories, including support for in-place logic, declarative values, strong syntax resemblance to UI, powerful dynamic scope and crucially – its most important feature in our opinion – compositional syntactic uniformity. The latter is to large extent a product of the remaining characteristics rather than a completely independent trait. In particular, without in-place logic and dynamic scoping rules, compositional syntactic uniformity would not be feasible. Without in-place logic, the existence of a component in a single block would be rendered impossible; whereas without a dynamic scope, moving a block would likely require providing additional linking mechanisms between the components. The tree-like syntax resembling a UI hierarchy provides further incentive and visual distinction, which ensures that maintaining components in-place feels like a natural choice and facilitates their modularization later. Lastly, declarative values help to keep up-to-date the part of a component's state which depends on its ascendants or descendants without the need to explicitly manage their creation order.

The topic of syntactic uniformity has been given some attention before[70,71], although not in the exact same sense as we are employing the term. The credo coined by Mark Lewis – "Similar things should be done with similar syntax" – captures the essence of our value proposition but, in order to keep its generality, is – necessarily so – so abstract, that it is difficult to immediately imagine what it could mean in the context of an UI framework. Furthermore, it is not clear if defining a component in-place, as opposed to declaring it in a separate code unit for reuse would be considered similar and what degree of similarity of the syntax that would imply. We would like to put across a working definition of the compositional syntactic uniformity as follows: the two things – declaring a component in-place or in a separate code unit – are identical (barring the location) and the syntax for the two should be literally identical. We mean to say that it should be possible to copy/paste a code fragment out of the UI tree and into a separate code unit and immediately obtain a program maintaining full functionality without any further code changes. With this said, it becomes obvious that the step-wise and continuous modularization of a program as it grows, is a very straightforward action in this paradigm, and therefore becomes a high reward/low overhead choice for the developer, in the end significantly improving the coding culture and practices.

In conclusion, among the frameworks under consideration, PyDeclarative presented the most complete implementation of the feature set (see Table 2) we argue to be essential for the development of declarative user interfaces characterized by high expressiveness, scalability and ease of refactoring. Lastly, we consider the compositional syntactic uniformity to be both the most important trait and the final added value on top of the other traits, which, in practice, are its prerequisites.

Limitations and Improvements

One of the traits of PyDeclarative that could potentially be improved is its eagerness in pushing changes through the dependency graph. While it avoids the accumulation of deferred computations and allows for straightforward handling of all the callbacks reacting to change along the update path, we believe that an approach similar to the one in the Param library used by the Panel framework could be an interesting alternative. In fact, we could combine the advantages of the current implementation with the benefits of a lazy approach by evaluating the expressions eagerly only if change callbacks were detected in the relevant portions of the dependency graph. Furthermore, we could avoid the accumulation problem by periodically forcing a re-computation of invalidated expressions, for example in fixed intervals or each time when the processing settles after a message is received from the DOM. Lastly, lazy DOM updates would be amenable to be regrouped together towards the end of the processing cycle just before they are sent to the browser, therefore avoiding eager re-computation each time one of the dependencies changes. Depending on the declarative expressions used and the pattern of updates, this could bring measurable performance gains.

Another potential performance improvement concerns the computation of the DOM updates. Currently, the contract between the framework and the components is such, that the components are required only to deliver the HTML representing their current state, without taking into account their previous DOM representation, nor participating in any way in the computation of the delta. While this is an extremely straightforward and convenient approach, the potential benefits of delegating the update computations to the components are hard to overlook. A satisfactory compromise could consist of a more granular use of node identifiers inside of the component, which would allow the *htmldiff* function to – still automatically – identify when it was dealing with filtering, sorting, removing or adding of internal nodes of a component and when the changes were more extensive. This could be achieved without requiring too much additional work from the component authors. On the other hand, one might argue, that the current implementation simply assumes, that the HTML representations are opaque, and already provides the possibility to arrange components in hierarchies, therefore already enabling all of

the aforementioned optimizations. While this is true, in order to find the optimal solution, the complexity of implementing a more intelligent *htmldiff* algorithm would have to be weighted against the complexity of writing components consisting of dynamically created and destroyed sub-components.

Declarative programming in general and reactive programming in particular present interesting opportunities for implicit parallelism. Since the order of the updates to the reactive values is dictated only by their dependency graphs, updates within independent sets could be computed in parallel. The Python frameworks showcased in this manuscript serve frequently as a base for building dashboards, often producing interfaces consisting of many plots, tables and images. Typically, those elements are updated in sequence; however, with the use of implicit parallelization and graph partitioning, they could be generated simultaneously. Similarly, long-running asynchronous I/O operations would benefit from being executed in parallel. In practical terms, when using a framework such as PyDeclarative or Shiny for Python, this method would require that the functions defining reactive expressions were allowed to operate asynchronously. This could add a lot to the flexibility and to the performance despite coming with its own set of potential challenges, such as the need for synchronization, resource locking and circumventing the Global Interpreter Lock (GIL) in Python. Maintaining the asynchronicity optional could lead to a framework that is both more powerful and adaptable.

Native Language Support

One interesting general consideration is whether a native support for reactivity at the level of the Python interpreter would allow the existing Python declarative/reactive UI frameworks to improve the syntax, the robustness, the speed and the programming experience they currently offer. Let’s address these questions step by step. It is clear, that unless the language syntax were to become completely reactive, some notation would still be required to differentiate between the classical expressions executed instantly and the reactive ones. In QML, all expressions living within the QML syntax are reactive, whereas the contents of the Javascript functions are executed according to the regular Javascript rules. This is the opposite of PyDeclarative, where plain expressions are executed immediately, whereas reactive properties are declared using functions, that define how to compute the current property values. Similarly to QML, non-property functions operate according to normal rules. If Python featured a context manager-style solution allowing to flip the behavior of expressions and/or to intercept their declarations, it would certainly allow to make PyDeclarative’s syntax terser. On the other hand, the functionality described above could be as easily achieved using the existing capability of

Python to generate Abstract Syntax Trees (ASTs) of Python code in order to write a preprocessor, that would translate the more concise notation into the current PyDeclarative syntax.

The question of robustness is a highly multifaceted topic and requires taking into consideration the interactions of reactivity with such Python mechanisms as: asynchronous operation, context managers, multi-threading, context variables, the interpreter lock, sub-interpreters, data location (e.g. in GPU-accelerated frameworks) and more. In this regard, full compatibility could be more easily achieved if the mechanism of reactivity was integrated into the Python interpreter. With that said, as long as reactivity stays synchronous, single-threaded and is used predominantly to describe the behavior of the UI rather than to define the entire application, clashes with the language facilities enumerated above can be easily circumvented or avoided entirely with just a bit of programmer's insight.

Native support for reactivity could bring speed improvements by implementing the performance critical parts (e.g. dependency tracking, DOM update delta computation or event propagation) in a more efficient language, for example in C in the case of the CPython interpreter. However, implementing critical paths in the main code base would have to be weighted against doing so in a compiled module, which could deliver comparable performance benefits. We believe that modifying the core implementation for performance reasons would only be justified if it was coupled with addressing the other issues mentioned above.

Lastly, the programming experience is a broad and not strictly defined category that could include elements such as language consistency, syntactic sugar, standardization, co-evolution with other language features, documentation, community and ecosystem support (e.g. linters, code completion, etc.). There is a strong chance that those aspects would be addressed more readily should reactivity/declarativeness constitute a part of the core language. There are precedents of core language features being popularized by third-party libraries. To name a few examples, the concept of promises in JavaScript was first popularized by libraries like jQuery[72] to later become part of the ECMAScript 2015 (ES6) standard. The Stream API introduced in Java 8[73] was influenced by functional programming concepts and libraries like Guava[74]. C#[75] features like async/await were first available through the Async CTP (Community Technology Preview) library[76] before being integrated into C# 5.0. Reactive/declarative programming concepts like those found in RxJava for Java, RxJS for Javascript or PyDeclarative for Python could potentially follow a similar path if they prove to be widely beneficial and if there is enough demand in the community.

Further Considerations

Formalizing PyDeclarative as a distributed reactive system, where the distributed parts would be the Python backend and the JavaScript/DOM frontend, could pave a way to mirroring the full capabilities of the browser-based code on the Python side by exposing the entire DOM and the JavaScript APIs server-side, using reactivity as an enabler of state mirroring. It could be as well a promising venue towards developing more elaborate distributed setups, e.g. applications consisting of multiple reactive Python backend nodes interacting with the frontend or with multiple frontends (e.g. multiple users) in an arbitrarily defined fashion. As suggested by Bainomugisha[33], to ensure consistency in a distributed dependency graph, one would need to take into account network latency, failures, etc. These additional challenges would need to be carefully weighted against the obtained benefits and the practicality of such a solution.

Integration with existing reactive programming frameworks for Python like RxPY could be a promising way to extend the reactivity beyond the UI (e.g. streams, schedulers) without introducing unnecessary redundancy.

Lastly, the support for multidirectionality could be an attractive and practical addition to PyDeclarative, with the most popular example being dialog boxes, where values of data model fields could be bidirectionally bound to the values used by the dialog's child controls.

Summary

PyDeclarative is a milestone in the development of declarative and reactive UI frameworks for Python. As demonstrated in feature comparisons against its peers, it excels in many characteristics crucial for this category of software frameworks, which results in high expressiveness, scalability and ease of refactoring. Furthermore, as proven by extensive benchmarking, PyDeclarative delivers competitive performance to its state of the art competitor – Shiny for Python, while offering a more complete gamut of desirable features, crowned with its compositional syntactic uniformity. In summary, PyDeclarative is a highly practical and elegant reactive/declarative UI platform and establishes a capable new baseline for future developments in this research area.

References

1. declarative.dev. https://declarative.dev.
2. Torgersson, O. A note on declarative programming paradigms and the future of definitional programming. *Winteroete* **96**, 13.
3. Egri-Nagy, A. Declarativeness: the work done by something else. Preprint at http://arxiv.org/abs/1711.09197 (2017).
4. Ansel, J. *et al.* PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* 929–947 (ACM, La Jolla CA USA, 2024). doi:10.1145/3620665.3640366.
5. TensorFlow Developers. TensorFlow, Large-scale machine learning on heterogeneous systems. https://doi.org/10.5281/ZENODO.4724125 (2024).
6. Chollet, F. Keras. https://github.com/fchollet/keras (2015).
7. Schuster, C. & Flanagan, C. Reactive programming with reactive variables. in *Companion Proceedings of the 15th International Conference on Modularity* 29–33 (ACM, Málaga Spain, 2016). doi:10.1145/2892664.2892666.
8. Francis-Landau, M. Declarative Programming via Term Rewriting. (The Johns Hopkins University, Baltimore, Maryland, 2024).
9. GWT Project. https://www.gwtproject.org/.
10. React. https://react.dev/.
11. ReactiveX for Python. https://github.com/ReactiveX/RxPY.
12. Dyna - Logic Programming for Machine Learning. https://dyna.org/.
13. Hickey, R. A history of Clojure. *Proc. ACM Program. Lang.* **4**, 1–46 (2020).
14. Braßel, B., Hanus, M. & Müller, M. High-Level Database Programming in Curry. in *Practical Aspects of Declarative Languages* (eds. Hudak, P. & Warren, D. S.) vol. 4902 316–332 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2008).

15. Fischer, S. A functional logic database library. in *Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming* 54–59 (ACM, Tallinn Estonia, 2005). doi:10.1145/1085099.1085110.

16. Kordjamshidi, P., Roth, D. & Kersting, K. Declarative Learning-Based Programming as an Interface to AI Systems. *Front. Artif. Intell.* **5**, 755361 (2022).

17. Howard, M. Terraform -- Automating Infrastructure as a Service. Preprint at http://arxiv.org/abs/2205.10676 (2022).

18. Tomarchio, O., Calcaterra, D. & Modica, G. D. Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *J. Cloud Comput.* **9**, 49 (2020).

19. Mondal, S. K., Pan, R., Kabir, H. M. D., Tian, T. & Dai, H.-N. Kubernetes in IT administration and serverless computing: An empirical study and research challenges. *J. Supercomput.* **78**, 2937–2987 (2022).

20. Baresi, L., Quattrocchi, G., Tamburri, D. A. & Terracciano, L. A Declarative Modelling Framework for the Deployment and Management of Blockchain Applications. Preprint at http://arxiv.org/abs/2209.05092 (2022).

21. *Practical Aspects of Declarative Languages: 22nd International Symposium, PADL 2020, New Orleans, LA, USA, January 20–21, 2020, Proceedings*. vol. 12007 (Springer International Publishing, Cham, 2020).

22. *Practical Aspects of Declarative Languages: 26th International Symposium, PADL 2024, London, UK, January 15–16, 2024, Proceedings*. vol. 14512 (Springer Nature Switzerland, Cham, 2023).

23. *Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings*. vol. 12548 (Springer International Publishing, Cham, 2021).

24. *Practical Aspects of Declarative Languages: 24th International Symposium, PADL 2022, Philadelphia, PA, USA, January 17–18, 2022, Proceedings*. vol. 13165 (Springer International Publishing, Cham, 2022).

25. *Practical Aspects of Declarative Languages: 25th International Symposium, PADL 2023, Boston, MA, USA, January 16–17, 2023, Proceedings*. vol. 13880 (Springer Nature Switzerland, Cham, 2023).

26. W3C. XForms 1.1. World Wide Web Consortium. https://www.w3.org/TR/xforms11/ (2009).

27. Claessen, K., Vullinghs, T. & Meijer, E. Structuring graphical paradigms in TkGofer. *ACM SIGPLAN Not.* **32**, 251–262 (1997).

28. Willman, J. M. Working with Qt Quick. in *Beginning PyQt: A Hands-on Approach to GUI Programming with PyQt6* (Apress, Berkeley, CA, 2022). doi:10.1007/978-1-4842-7999-1.

29. Streamlit Authors. Streamlit: The fastest way to build custom ML tools. https://www.streamlit.io/.

30. Dash Documentation and User Guide. https://dash.plotly.com/.

31. Kasprzak, P., Mitchell, L., Kravchuk, O. & Timmins, A. Six Years of Shiny in Research - Collaborative Development of Web Tools in R. *R J.* **12**, 155 (2020).

32. Shiny for Python. https://shiny.posit.co/py/.

33. Bainomugisha, E., Carreton, A. L., Cutsem, T. V., Mostinckx, S. & Meuter, W. D. A survey on reactive programming. *ACM Comput. Surv.* **45**, 1–34 (2013).

34. Salvaneschi, G., Margara, A. & Tamburrelli, G. Reactive Programming: A Walkthrough. in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering* 953–954 (IEEE, Florence, Italy, 2015). doi:10.1109/ICSE.2015.303.

35. Van Rossum, G. & Drake Jr, F. L. *Python Tutorial*. (Centrum voor Wiskunde en Informatica, Amsterdam, 1995).

36. Ross, P., Corne, D. & Fang, H.-L. Improving evolutionary timetabling with delta evaluation and directed mutation. in *Parallel Problem Solving from Nature — PPSN III* (eds. Davidor, Y., Schwefel, H.-P. & Männer, R.) vol. 866 556–565 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1994).

37. Sun, Y., Peng, X. & Xiong, Y. Synthesizing Efficient Memoization Algorithms. *Proc. ACM Program. Lang.* **7**, 89–115 (2023).

38. Garcia, R., Lumsdaine, A. & Sabry, A. Lazy evaluation and delimited control. *ACM SIGPLAN Not.* **44**, 153–164 (2009).

39. McSherry, F., Murray, D. G., Isaacs, R. & Isard., M. Differential dataflow. in *Proceedings of the 6th Conference on Innovative Data Systems Research (CIDR)* (2013).

40. Maher, M. J. Propagation Completeness of Reactive Constraints. in *Logic Programming* (ed. Stuckey, P. J.) vol. 2401 148–163 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2002).

41. Dataflow Programming. https://devopedia.org/dataflow-programming (2020).

42. Davis, A. L. *Reactive Streams in Java: Concurrency with RxJava, Reactor, and Akka Streams*. (Apress, Berkeley, CA, 2019). doi:10.1007/978-1-4842-4176-9.

43. Wan, Z. & Hudak, P. Functional reactive programming from first principles. in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation* 242–252 (ACM, Vancouver British Columbia Canada, 2000). doi:10.1145/349299.349331.

44. Maglie, A. ReactiveX and RxJava. in *Reactive Java Programming* 1–9 (Apress, Berkeley, CA, 2016). doi:10.1007/978-1-4842-1428-2_1.

45. Oeyen, B., De Koster, J. & De Meuter, W. Reactive Programming without Functions. *Art Sci. Eng. Program.* **8**, 11 (2024).

46. RxJava: Reactive Extensions for the JVM. https://github.com/ReactiveX/RxJava.

47. RxJS: Reactive Extensions Library for JavaScript. https://rxjs.dev/.

48. Bacon.js - Functional Reactive Programming library for JavaScript. https://baconjs.github.io/.

49. Understanding reactivity. *MobX: Simple, scalable state management* https://mobx.js.org/understanding-reactivity.html.

50. ReactiveUI - an advanced, composable, functional reactive model-view-viewmodel framework for all .NET platforms. https://www.reactiveui.net/.

51. Elm - delightful language for reliable web applications. https://elm-lang.org/.

52. Reactivity Declarations. *SVELTE: cybernetically enhanced web apps* https://learn.svelte.dev/tutorial/reactive-declarations.

53. Reactivity Fundamentals. *The Progressive JavaScript Framework* https://vuejs.org/guide/essentials/reactivity-fundamentals.html.

54. Bootstrap · The most popular HTML, CSS, and JS library in the world. https://getbootstrap.com/.

55. pandas - Python Data Analysis Library. pandas - Python Data Analysis Library.
56. DataTables | Javascript table library. https://datatables.net/.
57. Plotly javascript graphing library in JavaScript. https://plotly.com/javascript/.
58. The WebSocket Protocol. https://datatracker.ietf.org/doc/html/rfc6455.
59. Websockets 12.0 documentation. https://websockets.readthedocs.io/.
60. Gradio. https://www.gradio.app/.
61. Anvil | Build web apps with nothing but Python. https://anvil.works/.
62. Overview - Panel v1.4.2. https://panel.holoviz.org/.
63. SwiftUI Overview - Xcode - Apple Developer. https://developer.apple.com/xcode/swiftui/.
64. UI Benchmark. https://localvoid.github.io/uibench/.
65. Imba: the friendly full-stack language. https://github.com/imba/imba.
66. kivi. https://github.com/localvoid/kivi.
67. snabbdom - a virtual DOM library with a focus on simplicity, modularity, powerful features and performance. https://github.com/snabbdom/snabbdom.
68. Secure, Fast, & Private Web Browser with Adblocker | Brave. https://brave.com/.
69. Chromium. https://www.chromium.org/Home/.
70. Chen, B. & Miyao, Y. Syntactic and Semantic Uniformity for Semantic Parsing and Task-Oriented Dialogue Systems. in *Findings of the Association for Computational Linguistics: EMNLP 2022* 855–867 (Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 2022). doi:10.18653/v1/2022.findings-emnlp.60.
71. Lewis, M. Syntactic Consistency/Uniformity. https://drmarkclewis.medium.com/syntactic-consistency-e9d900df83b.
72. jQuery: The Write Less, Do More, JavaScript Library. https://jquery.com/.
73. Java Software | Oracle. https://www.oracle.com/java/.
74. Guava: Google Core Libraries for Java. https://github.com/google/guava.
75. C# language documentation. https://learn.microsoft.com/en-us/dotnet/csharp/.
76. Visual Studio Async CTP (Version 3). https://www.microsoft.com/en-us/download/details.aspx?id=9983.

Tables and Figures

Table 1: Comparison of Shiny for Python, Streamlit, Dash, Gradio, Anvil, Panel, QML along the six reactive programming dimensions defined by Bainomugisha; N/A not applicable.

Framework	Basic abstractions	Evaluation Model	Lifting Operations	Multidirectionality	Glitch Avoidance	Distribution Support
PyDeclarative	Signals, slots, properties, bindings	Push-based	Implicit	No	No	No
Shiny for Python	(Reactive) Calc(ulation), Effect, Value	Pull-based[a]	Manual	No	Yes[b]	No
Streamlit[c]	N/A	N/A	N/A	N/A	N/A	N/A
Dash	Inputs, outputs, callbacks	Limited push-based[d]	N/A	No	N/A	No
Gradio[e]	N/A	N/A	N/A	N/A	N/A	N/A
Anvil[f]	N/A	N/A	N/A	N/A	N/A	N/A

a See shiny/reactive/_reactives.py: Calc_.update_value() is called as late as Calc_.get_value()

b Implicit in the pull-based evaluation model

c Generally reruns the entire script on each change to the UI (memoization and compartmentalization possible)

d Dependencies have to be explicitly specified by declaring callbacks which enumerate their input dependencies and the outputs; each output can only be the target of one callback

e https://github.com/gradio-app/gradio/issues/4689

Framework	Basic abstractions	Evaluation Model	Lifting Operations	Multidirectionality	Glitch Avoidance	Distribution Support
Panel	Reactive parameters and expressions	Push-based / Pull-based[a]	Manual	No	No / N/A[b]	No
QML	Signals, slots, properties, bindings	Push-based[c]	Implicit	Yes[d]	No[c]	No
SwiftUI	State vars, computed vars, bindings, events	Pull-based[e]	Implicit	Yes[f]	N/A[e]	No

f Not a declarative framework

a Depends on whether a reactive expression is eagerly watched; pull-based by default

b No glitch avoidance in eager mode but creating glitches requires a very explicit redefinition of a parameter with dependencies; not needed in pull-based mode

c https://doc.qt.io/qt-6/qtqml-syntax-propertybinding.html

d Simple use cases can be handled by using property aliases; for more advanced ones, see: https://www.kdab.com/qml-component-design/

e @State computed vars are eagerly evaluated only if they appear and in the order that they appear in in a View, non-@State computed vars are lazily evaluated

f https://developer.apple.com/documentation/swiftui/binding

Table 2: Comparison of PyDeclarative, Shiny for Python, Streamlit, Dash, Gradio, Anvil, Panel, QML, SwiftUI on the following traits: Programming Language, In-place Logic, Declarative Values, Compositional Syntactic Uniformity, Syntax Resemblance to UI, Scoping Rules, Dynamic Callbacks

Framework	**Programming Language**	**In-place logic**	**Declarative values**	**Compositional Syntactic Uniformity**	**Syntax Resemblance to UI**	**Scoping Rules**	**Dynamic Callbacks**
PyDeclarative	Python	Yes	Yes	Any part of UI can be made into a component without code modification	Strong	Dynamic scope; Can access any element by non-unique ID in a predefined search order	Fully dynamic signals/slots
Shiny for Python	Python	No	Yes	No	Strong	Same as Python	Indirectly[a]
Streamlit	Python	N/A	N/A	No	Weak	Same as Python	N/A
Dash	Python	No	No	No	Strong	Same as Python	Pattern-matching callbacks
Gradio	Python	No	No	No	Weak	Same as Python	Yes
Anvil	Python	No	No	No	Weak	Same as Python	Yes
Panel	Python	No	Yes	No	Middle	Same as Python	Yes
QML	QML, JavaScript	Yes	Yes	Yes	Strong	Dynamic scope; Direct or qualified access	Yes

a Via storing the current callback as a reactive value

Framework	Programming Language	In-place logic	Declarative values	Compositional Syntactic Uniformity	Syntax Resemblance to UI	Scoping Rules	Dynamic Callbacks
						within component instance hierarchy[b]	
SwiftUI	Swift	Yes	Yes	Yes	Strong	Same as Swift; Direct access to parent properties, access to child properties using @Binding	Indirectly[c]

b https://doc.qt.io/qt-6/qtqml-documents-scope.html

c By providing own dispatch logic in the single handler or by using the Combine framework

*Figure 1: Benchmark results: this could include the regular uibench results on a second X axis. Error bars displayed as Mean – 1.96 * Std, Mean + 1.96 * Std. The lower end is clipped to zero. Individual measurements marked as points. Dashed vertical line at x=7ms illustrates the refresh rate of the monitor which influences the behavior of the window.requestAnimationFrame() function in JavaScript. Std, standard deviation.*