Functional Programming Paradigm of Python for Scientific Computation Pipeline Integration

Chen Zhang * [1], Lecheng Jia[1], Wei Zhang[2], and Ning Wen[3]

[1] *Real-time Technology Laboratory, Shenzhen United Imaging Research Institute of Innovative Medical Equipment*

[2] *Bussiness Division of Radiotherapy, United Imaging Healthcare Inc.*

[3] *Institute for Medical Imaging Technology, Shanghai Jiao Tong University School of Medicine, United Imaging Healthcare*

Abstract

As the preferred programming language for artificial intelligence (AI), Python has gained immense popularity due to its versatile syntax, flexibility of multiple programming paradigms, as well as the robust community support. This popularity becomes even further in the realm of scientific computation, machine learning, or AI algorithms where there is a great demand of data processing. Nevertheless, modern data processing unveil increasingly tendency of interdisciplinarity, which frequently involves in importing different technical approaches. An unified data control, is therefore in urgent, to gain performance of system integration among varying libraries. This integration is of the profound significance of accelerating prototype verification, optimizing algorithm performance, and minimizing maintenance cost.

Most researchers still have to make trade-offs between the ease of prototype implementation, and completeness of data engineering. For quite a few labs, using Python as a glue language, utilizing its advantages of abundance of third-party libraries, are preferable to accelerate in implementing prototypical algorithm dealing with complicated tasks. However, the lack of standardized data management, controlling, possibly pins their achievements on experimental stage, instead of contributing for practical usage. To tackle this issue, we proposed a frame of functional programming tailored for data processing pipelines. Utilizing this framework, we hope to achieve the following objectives:

1) Facilitate data processing and analytical tasks by leveraging a plethora of meta toolkit of scientific computation. By providing a unified interface, the framework enables seamlessly integrating existing functions, or even any callable object in Python, in highly engineered form. These features accelerate the construction for standard functional components, used both in researches, and data engineering.

*E-Mail: chen.zhang_06sept@foxmail.com;

2) Establish the units of data flow controlling with concise syntax and unification application programming interface (API). The predefined operators of these units ensure consistency without conflict in component connection, therefore contribute to the creation of more systematic and complicated data processing pipelines, through existing functional units.

3) Increasing the readability and maintainability of logical units via massively utilizing the closure functions as decorators. This design is featured as decoupling the definition for operations on data, from the assignment in arguments that affects these operations. Once a system becomes enormous, both the data operations and the arguments are tough to be managed in the design of object-oriented programming (OOP), while the one in case of this frame results seldom in mess.

4) Proposing a scalable solution for data science and engineering, ensuring reliable performance across full ranges, from exploratory research to agile development and deployment. The extensibility of this framework guarantees its adaptation to the evolving requirements of data-driven projects, as well as to the increasing complexity of a data processing system.

Keywords Python, scientific computation, functional programming

Introduction

Python gains increasing popularity since its compact but efficient design in recent years. As an interpreted programming language, Python is of the characteristics of concise syntax, flexibility on multiple programming paradigms, cross-platforms, etc. [1–3]. Its software ecosystem gets enriched, which benefits from the contributions of soaring researchers and developers studied in various fields. As with the booming development of AI techniques generally, Python has become a de facto development standard for scientific computation and AI algorithms due to not only some excel high performance libraries such as numpy, scipy, and tensorly [4–6], but also high compatibility for integrating other programming languages.

Nonetheless, as the expense of exceeding flexibility and the richness of software ecosystem, it gets also challenging in integration for complicated project via Python, particularly in scientific computation used in interdisciplinary application [7–10]. Such as, most of data manipulations are generally in demands of importing third party libraries designed on basis of different specifications, which can frequently result in incompatibility problems raised by data types or such like. Or, even if there is no compatibility issue raised by the data, when using interfaces with the identical function sourced from different packages (e.g. Gaussian filters in Pillow and opencv-python), factors such as different argument design or predefinition can further lead to variation in the final export. Similar situations, are generally too cumbersome to troubleshoot at the function invocation level, but have a major impact on the systematic reliability of our computation and analysis.

Those difficulties become even tough in the condition of operations involved in introducing complex algorithms, heavy numerical computations, or the manipulations on the great scale of dataset [11, 12]. The conventional OOP approaches can become cumbersome and inefficient in such circumstances, as it may not offer the necessary

flexibility and scalability to handle the complexity of scientific computing, especially during the exploring stage when functions or scripts will need frequent modification for refactoring.

In order to solve the above-mentioned challenges, based on a large number of practices, we have summarized a set of flexible and effective generic infrastructure of Python function based on functional programming (FP), to ensure both the invoking legality and extensibility for heavy modification as well as for further integration. With this framework, we achieved significant improvements with respect to code readability, maintainability, and performance. The FP paradigm, with its emphasis on immutable data and pure functions without side effects [13–16], provides a more natural approach to handling the intricacies of scientific computing. It is in advantages on creation for modular and highly decoupled code style, which is precisely the essential for achieving completeness of functions, and the further optimization on the software architecture as well. The reliability and consistency of functions, or components built on this frame can therefore be both guaranteed.

Background and Architecture

In the context of scientific computing, the most fundamental process is the transformation of data forms. In Python, this abstraction can be applied to a wide range of computing and analysis scenarios, including data pre- and post-processing, as well as machine learning model training (as illustrated in Figure 1). Despite differences in code or script organization, for identical functional units, they should at least include declarations of the types of data that can be accepted, the types of data that will be returned after manipulation, and so on. The Python programming language has already incorporated a similar feature since version 3.5, called typing hint [17]. This allows programmers to assign the suggested argument type when defining a function, although it acts more like an annotation than a static constraint.

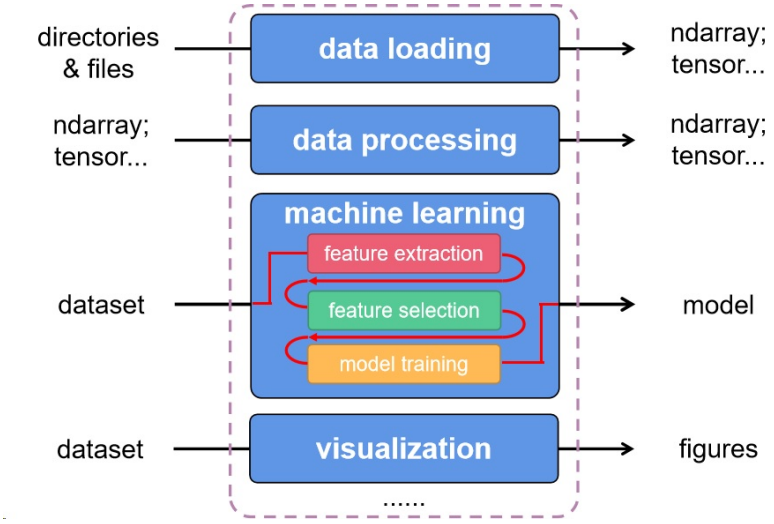

Figure 1: The essence of data manipulation: transforming data to represent it from one form to another.

The situation is further compounded in the domain of data exploration, where it

is virtually impossible for an architecture to simultaneously satisfy the two contrasting requirements: the flexibility required by researchers in data investigation and the reliability required by engineers in implementation and deployment. The inherent technical divergence between these two populations can be illustrated in Figure 2. Researchers are generally more concerned with approaches and concepts than with their implementation, whereas data engineers are conversely more concerned with the implementation of concepts. This situation inevitably increases the communication costs in teamwork, as well as those in programming implementation and maintenance, which represents a significant obstacle to the transformation of scientific and technological achievements.

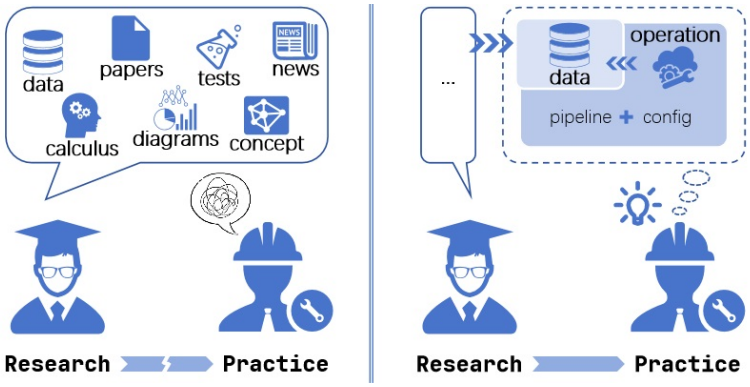

Figure 2: The technical divergence between data research and engineering

As for the majority of data analysts, Python is regarded as an essential tool to master due to its robust community support and plethora of exemplary third-party libraries. As Python is a dynamic language with a high degree of flexibility, it is important to implement a run-time check for passing in and out in order to avoid potential errors such as parameter type confusion or improper call. This can further reduce the potential risk for calling and integrating functions. Based on the aforementioned causes, we propose a specific functional programming (FP) frame for Python that introduces forced constraints on simple functions, thereby ensuring logical correctness in more complex constructions.

The illustration of this FP framework is as shown in Figure 3: the light salmon background represents the main scope of feature set of our infrastructure for Python functions, while blocks with different colours demonstrate the main function suites. In terms of type inspection (navy), the framework wraps the actual data manipulation inside, where data and corresponding arguments are validated before passing to the real execution, while the processed data is also checked again before returning. As a function or callable object wrapper (shown in gray), the frame first modifies the attributes of the decorated object, defines the data with the desired types in input and return, or even more strictly, configures the default values and their respective types of the function's associated arguments. The tester (shown in lime) is a development tool in function construction, used for executing case tests, logic validation, time-consuming evaluation, and so forth.

The design philosophy of this architecture is to logically isolate data manipulation into the data itself and the corresponding parameters for control, and then to technically separate the data and arguments from the body of the function definition. Its structure allows set operations on arguments of different decorated functions, which is

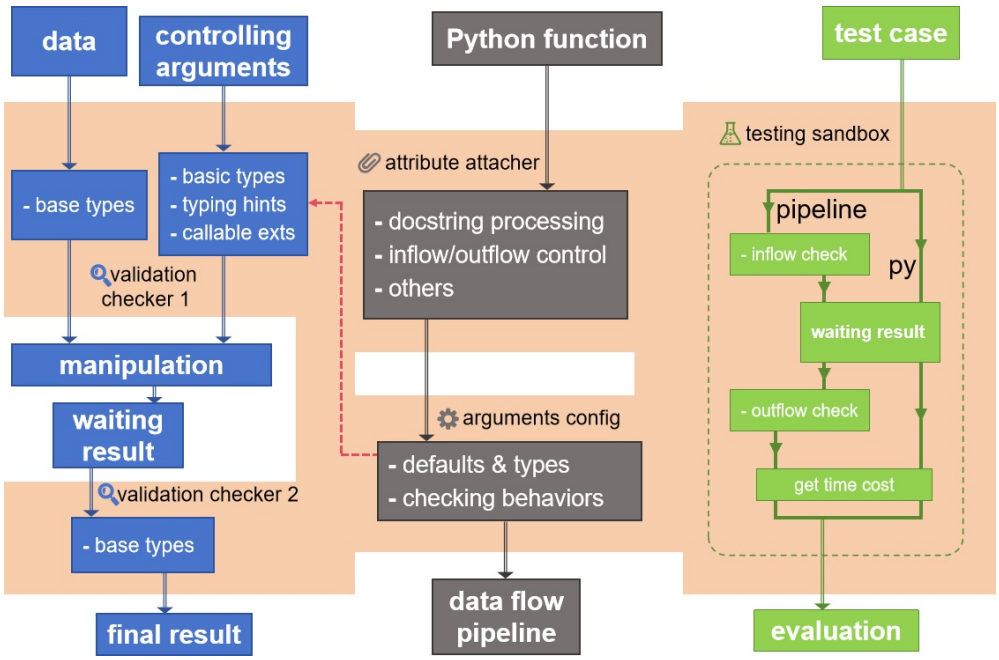

Figure 3: Illustration of the FP Python infrastructure used for the runtime checker (navy), function wrapper (gray), and tester (lime); the light salmon background is the framework's feature set.

crucial to support the construction of pipelines for more complicated structures. With regard to the manner of application, most of tools within this framework are closure functions employed as decorators in Python, therefore the utilisation of these features will not result in any additional learning costs for the majority of users. Furthermore, the necessity for extensive refactoring of the original project is also negated. The incorporation of these features facilitates the design and construction of data processing functions at all stages. The use of these wrappers enables function designers to conceal the inner details of the functions, while providing a set of standard argument configuration, akin to an API, in the function's pre-definition, which is then exposed to target users.

Function suite

It is first necessary to explain the basic concept of Python's argument passing mechanism. The example function in Listing 1 shows the four different types of parameters listed in order of inherent priority: positional argument(s), optional argument(s) with default value(s), var-positional, and var-keyword arguments. The prefix "var" means the uncertain length as a variable. In the context of Python syntax, it is not necessary to define all argument types simultaneously; however, once they have been defined, their relative order must be maintained in accordance with the aforementioned enumeration.

```
1 def function1(pos: int, opt: int = 1, *args, **kwargs):
2     print(pos, opt, len(args), len(kwargs))
3     return pos + opt + len(args) + len(kwargs)
4
5
6 function1(2)                  # 2, 1, 0, 0
```

```
function1(2, 3)                # 2, 3, 0, 0
function1(2, 3, 4, 5)          # 2, 3, 2, 0
function1(2, kw1=5, kw2=6)     # 2, 1, 0, 2
```

Listing 1: Python arguments passing mechanism.

The call examples shown in the Listing 3 section can assist us in comprehending the assignment behavior of arguments in the Python programming language. Python's syntax permits the utilization of arguments in two distinct manners: direct pass-in by values and declaration for their keywords during assignment. However, it is not possible to interchange the relative positions of these two assigning approaches (i.e., no-keyword calling prior to keyword one). In the absence of a keyword declaration, the interpreter attempts to fill the positional type first, followed by the optional type by overwriting. If there are still unmatched arguments, they are packed into the var positional type as a tuple. In contrast, arguments declared with keywords are processed as a dictionary at once.

In theory, any type of argument can be replaced in the same manner as the var-keyword. For instance, all types of arguments that appear in the example function in Listing 3 can be encapsulated within a unified set of dictionary parameters, as illustrated in Listing 2.

```
def function2(**kw):
    return function1(
        kw.get('pos'),
        kw.get('opt', 1),
        *kw.get('args', ()),
        **kw.get('kwargs', {}),
    )
```

Listing 2: Equivalent wrapper using only var-keyword arguments.

As our demonstration encompasses all potential forms of argument passing, the wrapper utilising solely the var-keyword is theoretically universal in its capacity to support any callable object within the Python programming language. The concept of the var-keyword function is of paramount importance within our architectural framework, which is designated the *info function* (derived from the informatics project, as referenced in [18]). This designation will be reiterated below. The advantage of utilising the info function is that it enables the utilisation of features designed exclusively for var-keyword callable objects.

Attaching attributes

With regard to the information function, it is typical for the keyword "data" to be reserved as the object to be processed in the current step, with the processed result being returned. In order to ensure that the accepted data is of the desired types, it is possible to simply prepend a decorator to the function with the appropriate typing hint declarations. To illustrate, when scripting a function for text manipulation, if it is desired to support processing for a string or a list composed of a set of strings, the corresponding constraint can be implemented in a concise manner as demonstrated in the code snippet labelled Listing 3. From the function call examples, it is evident that type checking is also performed on inner elements if an iterable complex structure is detected.

```python
from info.me import FuncTools
from typing import Union, Any


@FuncTools.attach_attr(info_func=True,
                       entry_tp=Union[str, list[str]],
                       return_tp=Any)
def func1(**kw):
    _ = kw.get('data')
    ...
    return 0


func1(data='string0')                # valid, str
func1(data=['string1', 'string2'])   # valid, list[str]
func1(data=('string3', 'string4'))   # invalid, tuple[str, ...]
func1(data=50)                       # invalid, int
func1(data=['string5', 50])          # invalid, list[str | int]
```

Listing 3: Data inflow control via Python typing hint.

The innovative aspect of the frame design is the logical decoupling of the argument validation steps from the main body of the function, which is achieved at a minimal cost. Without the attachment of the inflow control attribute (lines 5 to 7 in Listing 3, mainly 6), it is unsafe to assume the actual type it accepts in function entrance (line 9), therefore the check logic as shown in Listing 4 must be embedded after where the data was acquired. The code snippet executed for similar purposes is technically referred to as defensive programming [19]. Despite its prevalence in modern software design, integrating this block into the function body sacrifices flexibility in dealing with varying requirements, while simultaneously reducing readability and maintainability of the code. To illustrate, in order to accommodate the case presented in line 16 of Listing 3, the conventional approach necessitates the incorporation of support for tuple containers synchronously, based on the original type check snippet. Conversely, for the info function, a straightforward replacement is sufficient. The value of `entry_tp`, which is defined as `Union[str, list[str]]`, should be replaced with the value `Union[str, list[str], tuple[str, ...]]`. The example presented here serves to illustrate the control of data input. In contrast, for the control of data output, the similar type syntax applies to the argument `return_tp`.

```python
...
if not isinstance(_, (str, list)):
    raise ValueError('...')
elif isinstance(_, list):
    if not all([isinstance(_v, str) for _v in _]):
        raise ValueError('...')
...
```

Listing 4: Defensive programming snippet for argument checking.

As a practice of functional programming, this architectural approach can not only serve software security well, but also contribute to the design of the software architecture itself. For many mature Python packages, the majority of their API function bodies consist of considerably long scripts, which include not only essential defensive programming snippets for all passed parameters, but also corresponding documenta-

tion and invocation examples. However, as for the info function, it is architecturally supported to introduce and then integrate those necessary attributes using outer objects. As illustrated by the pseudo-code in Listing 5, the documentation of `main_func` can be fulfilled by a carrier function, `doc_func`, defined elsewhere, rather than by direct insertion into the main body of `main_func`.

```
def doc_func(**kw):
    """
    documentation begin
    ... rst doc for main_func ...
    documentation end
    """
    ...

@FuncTools.attach_attr(docstring=doc_func)
def main_func(**kw):
    ...
```

Listing 5: Attaching documentation through Python function carrier.

Furthermore, additional attributes can be attached to the target function, such as scripting author(s), maintainer(s) or tester(s), during practical development as needed. This can be achieved with relative ease (see examples). It can be observed that the frame is characterised by the flexible use of Python functions, which fully increases system reliability without sacrificing its scalability. Consequently, this leads to further rigor in software construction.

Argument configuration

The parameters of the info function are not solely comprised of the data to be processed; they also encompass the controlling arguments that can influence the processing steps. In general, the cases of controlling arguments are more intricate than that of data itself. Listing 6 reveals three typical modes of argument configuration using our functional frame. Firstly, built-in types, such as the `ndarray` type for `data`, can be used as constraints to restrict arguments. Secondly, typing hints, such as a tuple composed of float and integer for `arg1`, can be employed. Thirdly, callable objects, such as a coefficient between 0 and 1 for `arg2`, can be used as constraints to restrict arguments.

```
from info.me import FuncTools, T, Null
from numpy import ndarray


@FuncTools.params_setting(data=T[Null: ndarray],
                          arg1=T[(.3, 60): tuple[float, int]],
                          arg2=T[0.7: (lambda x: 0 < x < 1)])
def func2(**kw):
    ...
```

Listing 6: Three typical approaches for argument configuration.

The last type represents a revolutionary advancement in the field of type checking workflows, particularly in the context of scientific computation. The mathematical implementation in data engineering is more rigorous than that of software engineering. To illustrate, consider matrix computation. On occasion, users may wish to impose

certain constraints (such as symmetric, positive, or semi-positive definite) on the matrix to be acquired. Fortunately, these constraints can be easily defined by means of lambda functions. Once the constraint functions have been prepared, the target function can accept matrices with specific properties, using the frame with the form shown in Listing 7. Furthermore, the lambda functions are dumped as values (`_sym`, `_pos`, and `_semi_pos`) in script, allowing them to be reused in any context where necessary, in the form of common Python functions.

```
import numpy as np


_sym = (lambda x: True if (x.ndim == 2 and
                           x.shape[0] == x.shape[1] and
                           np.allclose(x, x.T)) else False)
_pos = (lambda x: np.all(np.linalg.eigvals(x) > 0))
_semi_pos = (lambda x: np.all(np.linalg.eigvals(x) >= 0))


@FuncTools.params_setting(mat1=T[Null: _sym],
                          mat2=T[Null: _pos],
                          mat3=T[Null: _semi_pos])
def func3(**kw):
    ...
```

Listing 7: Applying matrix constraints on decorated function.

This design feature has the potential to enhance the reusability of constraint functions while also improving the readability of decorated functions. Moreover, it provides an alternative solution for Python functions that can accommodate future derived types. The formal name for supporting those derivatives is terminologically called "duck typing" [20]. This concept is increasingly supported by features of modern programming languages, including `concept` in `C++20` and `GenericAlias` in Python [17, 21, 22]. To illustrate, consider the concept of a matrix. While its mathematical definition is relatively straightforward, the concrete implementations in Python remain challenging to enumerate. However, in our case, we can conveniently fulfill this requirement with an anonymous function that checks whether the dimension is 2 after type conversion into a numpy array. This approach allows for the legal passing of matrices and 2-dimensional arrays of numpy, sparse arrays or matrices of scipy, and any other nested iterable containers that are consistent with the mathematical concept of a matrix.

Testing frame

In addition to type control for input data and corresponding arguments, our infrastructure also provides a wealth of accessory tools, such as `default_param` for dynamic argument assignment, or `exception_logger` for capturing exceptional cases during execution, which support the serial requirements of agile development. In this context, the sandbox for execution case testing can drastically simplify script editing and organisation, since its feature of testing while scripting via function decorator. The testing framework can be applied not only to the info functions, but also to common Python functions in any form of argument declaration. For further details, please refer to the application examples provided by the API documentation.

This feature is beneficial for software architects working with Python-based tool chains. Typically, a given Python module can be decomposed into a number of callable components, with developers appending the test cases to the end of their script. As the number of necessary components increases, the length of the script under editing also increases, making it inefficient and potentially risky to repeatedly roll up and down on the script for each necessary test. In addition to the features of in-place testing on callable components, this frame also records the time consumed during the entire call cycle. Consequently, this infrastructure can not only provide convenience of validation for each component in the module, but also simultaneously evaluate basic performance on each component. This can be the reliable evidence for hot-path analysis or other possible optimisations in the future.

The testing frame can be employed at various stages of the architectural design and module construction processes. Additionally, it can be utilized in unit testing, if necessary. To illustrate, the test directory of the informatics project contains numerous test examples for the info functions, which are executed via a pipeline of unit tests with the test frame embedded.

Analysis and discussion

In this section, we will further elucidate the reasons why our FP infrastructure and its application can contribute to pipeline integration. We will examine the features of the functional programming paradigm, the performance loss after function decoration, and the convenience brought by the pipelining code style.

Flexibility of FP

The concept of a programming paradigm is not immediately intuitive. In essence, OOP represents a conceptual abstraction of entities with a noun attribute. This approach is analogous to the scientific classification of organisms, where the context domain, kingdom, phylum, class, order, family, and genus are used to describe creatures. OOP is a rigorous and comprehensive approach, but it lacks flexibility in dealing with open sets, such as the description of a completely new species that cannot be categorized by this system.

It is unfortunate that openness is precisely what matters in data exploration. The process of reading data from files is referred to as *data loading*, the manipulation of data is defined as *preprocessing*, the extraction of information from data is termed *feature engineering*, and the training of data to identify underlying patterns is referred to as *model training*. All of these operations have the verb attribute rather than that of noun. These behaviours exist objectively, but cannot be abstracted by the OOP paradigm without the input of concrete data for processing.

The appeal of FP lies in its capacity to define hitherto unknown behaviours, which can result in functions with high expressiveness and flexibility. For further illustration, consider the example of the generic data loader as presented in the informatics project tutorials. This is an integration pipe. The high-order functions `search_from_root` and `generic_filter` can be employed as an implementation of the generic data loader, despite the fact that the data to be loaded is unknown.

Performance loss

Given that the majority of the features in our programming framework applied to Python function are achieved through the use of decorators, it is reasonable to conduct a comprehensive evaluation of the performance loss when applying decorators. To this end, we have designed a benchmark test using a simple Python function with the behavior of hanging up for 0.1 seconds to mimic actual data processing, followed by a return. Two types of decorators were employed in this study: inflow/outflow control and argument configuration. These were applied separately or simultaneously to the original function, resulting in the generation of new callable objects. Each object was then subjected to 30 trials, after which their average time consumption was compared. The corresponding results were summarized in Figure 4.

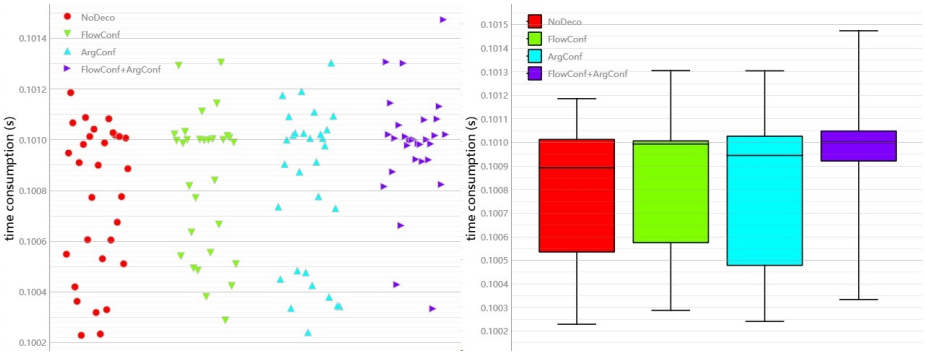

Figure 4: Beeswarm and box plots of benchmark tests on invocation time consumption, for an identical simple functions. **NoDeco**: without decorator; **FlowConf**: decorated by inflow/outflow control; **ArgConf**: decorated by argument configuration; and **FlowConf+ArgConf**: with both decorators.

The majority of the call instances are situated in the vicinity of the 0.1010 seconds position, as indicated by the scatter points in the beeswarm plot. Despite the varying degrees of numerical increase in function call time subsequent to decoration, their values are relatively close, with only minor discrepancies. Therefore, it is justifiable to accept a slight reduction in performance in order to guarantee the security and dependability of the input data and the parameters passed.

In particular, the core idea of the framework design is to prioritize data mapping over legal invocation for a specific computational step (although it can be utilized in this manner). It is also the distinction in design concept from the element-wise based high-order Python functions like map-reduce, as well as the essence of the FP paradigm. As illustrated in Figure 1, the majority of data manipulation within a computer system can be described as a process of mapping data from one existential form to another. This concept forms the basis of our proposed infrastructure, which can be viewed as an invocation protocol for data and arguments in data mapping. In its application, the user level may be more concerned with the content within the protocol itself, rather than the concrete logic of the function body. However, both aspects should be fully considered at the developer level.

Code style of pipeline

Thus far, we have expounded at length on the advantages of our infrastructure for secure invocation and good code organization. However, there is a deeper significance that extends beyond these benefits. By employing a unified protocol designed for both data and argument passing, we can ensure that the entries and exits of functions will have clear definitions. Consequently, the decorated functions will be equipped with the ability to communicate with each other in the absence of any practical data. This feature is particularly advantageous for the design of computer systems and the integration of scientific computation pipelines, particularly in the absence of data.

The framework allows for the highly reusable integration of either the info function or the pipeline derived from info functions. This can be illustrated by considering a concrete computer vision (CV) task. In natural image processing, a pipeline including cropping, denoising and resampling is required. As these atomic operations are already implemented in the informatics project, they can be easily connected using the info functions wrapped by the inline code `Unit`. The code structure would be represented by the Listing 8 (lines 4 to 5), where the ellipses within the list indicate the positions of the corresponding info functions. Similarly, if one were to compare different edge detection methods following this preprocessing operation, one could reuse this `processing` pipe in the definition of the experimental pipeline (lines 6 to 7 in Listing 8).

```
1 from info.me import Unit
2
3
4 crop, denoise, resample = [Unit(mappings=[_]) for _ in [...]]
5 processing = crop >> denoise >> resample
6 prewitt, canny, log = [Unit(mappings=[_]) for _ in [...]]
7 compare_experiment = processing >> (prewitt | canny | log)
```

Listing 8: Integrating scientific computation flows.

To demonstrate the functionality of the pre-processing pipeline, an 8-bit greyscale image was used as a test case. The image was pre-processed using a set of parameter configurations, and the intermediate image and edge-enhanced results after convolving the image with different kernels were generated. These results are shown in Figure 5 (b) to (e), respectively. The example image was derived from a demonstration instance in the scipy package. The wrapped units and integrated pipeline can be perceived as a series of atomic operations of mappings on data.

Pipeline integration based on atomic operations can be conceptualised as the design of a data assembly line, with basic components analogous to the material flow in this data assembly line. In light of the growing importance of data science and artificial intelligence, researchers are increasingly reliant on interdisciplinary analysis of data in different modes. The establishment of a unifying criterion for programming paradigms is of profound importance now and in the future, as it ensures that data can flow seamlessly throughout prototypical tools designed in different disciplines, analysis frameworks and corresponding libraries.

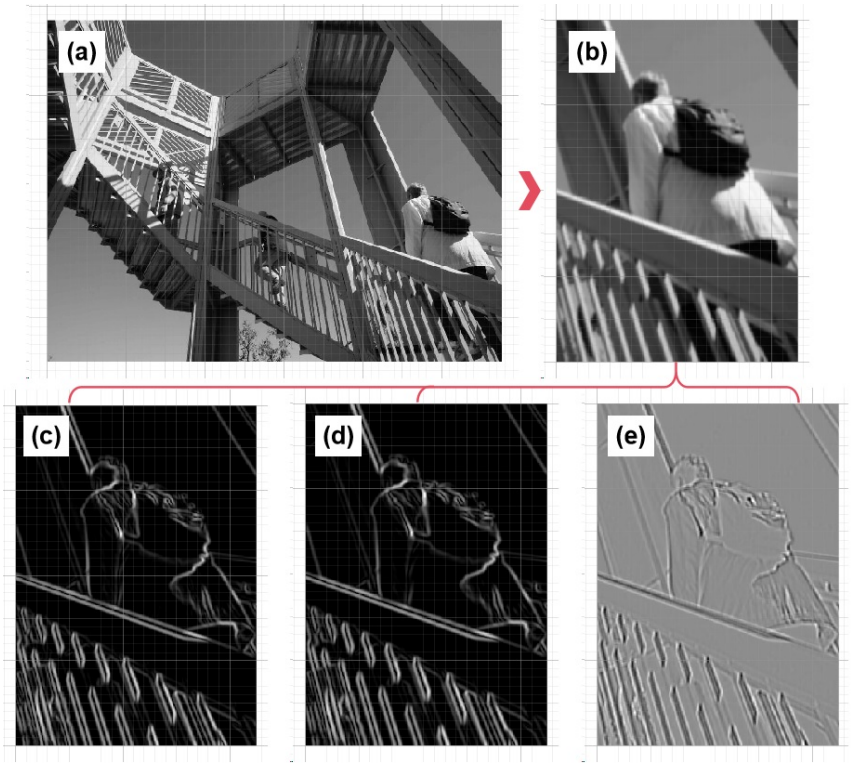

Figure 5: Demonstration of experimental pipe applied to natural image processing. (a) original 8-bit grayscale image; (b) preprocessed image after cropping, denoising, and resampling; and final images filtered by (c) Prewitt; (d) Canny; and (e) Laplacian of Gaussian.

Conclusion

This paper presents a novel FP paradigm based on the Python architecture, designed for the integration of pipelines of different data mapping operations. In particular, it is intended for the integration of scientific computation flows, which are commonly involved in the applications of computational tools and the validation of prototypical ideas.

As a fundamental concept within the framework, we proposed a specific form of function derived from the generic Python function, which has been demonstrated to be of theoretical equivalence to that of the general one. With a suite designed for that specific Python function, we logically separate a Python function into factors such as the main execution body, the target data, the corresponding argument configuration, the attributes, and so forth. The framework employs the features of the functional programming paradigm and high-order functions to provide a multitude of meta-implementations with considerable flexibility, thereby affording analysts and researchers a wide range of options for addressing problems. For engineers and maintainers, the framework also offers advantageous characteristics, including forced type checking, inflow/outflow control, and the prevention of illegal invocations within functions. This facilitates the tracking and subsequent resolution of issues.

Furthermore, it supports a distinctive connection syntax when the atomic data operations are wrapped as data processing units. With our infrastructure for Python functions and the informatics project, we hope researchers can make full use of these sophisticated tools to achieve their creative and flexible goals, which will then lead to

the valuable transformation of technological achievements.

Acknowledgements

Our sincerest gratitude to the United Imaging Healthcare Group, the Shanghai Jiao Tong University School of Medicine, the Human Resources and Social Security Administration of Shenzhen Municipality, and the Human Resources Bureau of Shenzhen Longhua District. Their unwavering support has been instrumental in facilitating a multitude of innovations, including this research, the informatics project, and the subsequent industrial application of AI techniques.

Gratitude to the Python community and its esteemed contributors. Your unwavering dedication to technical precision and expertise has established a robust foundation for developing cutting-edge tools for scientific computing. Your unparalleled commitment to open source collaboration and relentless innovation has fostered a thriving ecosystem that addresses unique challenges across diverse fields. Without your contributions, our progress in exploring programming practices in Python would be significantly hindered, both in terms of inspiration and the implementation of pipeline integration.

References

[1] Kashif Munawar and Muhammad Shumail Naveed. The impact of language syntax on the complexity of programs: A case study of java and python. *Int. J. Innov. Sci. Technol*, 4:683–695, 2022.

[2] David M Beazley. *Python essential reference*. Addison-Wesley Professional, 2009.

[3] Mark Summerfield. *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2009.

[4] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with numpy. *Nature*, 585(7825):357–362, 2020.

[5] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[6] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. Tensorly: Tensor learning in python. *Journal of Machine Learning Research*, 20(26):1–6, 2019.

[7] Travis E Oliphant. Python for scientific computing. *Computing in science & engineering*, 9(3):10–20, 2007.

[8] Fernando Perez, Brian E Granger, and John D Hunter. Python: an ecosystem for scientific computing. *Computing in Science & Engineering*, 13(2):13–21, 2010.

[9] Peter R Turner, Thomas Arildsen, and Kathleen Kavanagh. *Applied scientific computing: with Python*. Springer, 2018.

[10] Christian Schou Oxvig, Thomas Arildsen, and Torben Larsen. Storing reproducible results from computational experiments using scientific python packages. In *Scientific Computing with Python 2016*, pages 45–50, 2016.

[11] Davy Cielen and Arno Meysman. *Introducing data science: big data, machine learning, and more, using Python tools*. Simon and Schuster, 2016.

[12] Lisbeth Rodríguez-Mazahua, Cristian-Aarón Rodríguez-Enríquez, José Luis Sánchez-Cervantes, Jair Cervantes, Jorge Luis García-Alcaraz, and Giner Alor-Hernández. A general perspective of big data: applications, tools, challenges and trends. *The Journal of Supercomputing*, 72:3073–3113, 2016.

[13] Maurizio Gabbrielli and Simone Martini. Functional programming paradigm. In *Programming Languages: Principles and Paradigms*, pages 335–368. Springer, 2023.

[14] Vaskaran Sarcar. Functional programming overview. In *Introducing Functional Programming Using C# Leveraging a New Perspective for OOP Developers*, pages 3–32. Springer, 2023.

[15] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[16] Dino Alic, Samir Omanovic, and Vaidas Giedrimas. Comparative analysis of functional and object-oriented programming. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 667–672. IEEE, 2016.

[17] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. Type hints. PEP 484, 2014.

[18] Chen Zhang. *informatics: Framework of fast implementation data processing and operating pipelines*, 2024. Python version 3.9 or later.

[19] Jean-Louis Boulanger. 6 - technique to manage software safety. In Jean-Louis Boulanger, editor, *Certifiable Software Applications 1*, pages 125–156. Elsevier, 2016.

[20] M Anton Ertl. Methods in objects2: Duck typing and performance. In *28th EuroForth Conference*, page 96. EuroForth, 2012.

[21] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for C++. In *Software Language Engineering: 4th International Conference, SLE 2011, Braga, Portugal, July 3-4, 2011, Revised Selected Papers 4*, pages 97–118. Springer, 2012.

[22] Łukasz Langa. Type hinting generics in standard collections. PEP 585, 2019.