A Survey on Abstract Data Types

Anthony Hunt

February 8, 2025

1 Introduction

From the categorization of species to the periodic table of elements, the organization and manipulation of data is integral to all aspects of science and technology. Although the collection of data is not a new field of research by any means, the advent of computerized systems, with its ability to transfer and use massive amounts of data in mere seconds, necessitates understandable and sensible organization structures.

Over the past 60 years, as computational resource restrictions have eased dramatically, programming languages and software creation tools have evolved with increasingly better methods of dealing with data. By viewing pieces of data through specific lenses, programmers can then write algorithms that operate on any data matching the shape of those lenses, rather than the data itself. These lenses are more commonly referred to as data types, which assign a type to a piece of data. As we will see later in this paper, further abstractions over data types themselves enable extremely concise and readable code, both for the programmer and computer.

The rest of this article will explore data type concepts from ¡X¿ languages, structures to handle errors that might appear, and a small review of ergonomics and usability of those data types. A necessary discussion of various type systems will naturally evolve as a precursor to data types themselves.

2 Data Types in ¡X¿+ Languages

Abstract Data Types (ADTs) can look quite different in many languages, but often share the same principles and rules that make it understandable for programmers. In its essential form, an ADT is a collection of data together with operations that can be carried out on that data [21]. The key differences between ADTs from different languages are the conventions used to collect data and the power operations have over collected data.

In popular object-oriented programming (OOP) languages, ADTs are approached with a "class" mindset, where programmers specify a blueprint for objects that will eventually be used in a program. For better code reuse, classes can be built from pre-existing classes through the use of inheritance, a term used

to express that a new class inherits all the properties of the old class. Objects created by classes are referred to as instances of the class and often hold some amount of mutable data, with methods that have full access to the data. The methods can choose to make use of "side effects", referring to the modification of data not explicitly passed through regular function arguments or return values. Side effects in methods have the consequence that the output of a method may change independent of the values passed through its arguments. In other words, the object a method has access to may contain some state that influences the outcome of the method.

At the opposite end of the programming world lies purely functional languages. ADTs in these languages are more often referred to as data types rather than classes and are only used for collecting and operating on data. "Objects" created from these data types are immutable and methods are instead replaced with pure functions that explicitly take in the relevant data type as input. The operators attached to a functionally pure ADT are functionally pure themselves, meaning they cannot make use of any hidden state and therefore cannot have side effects.

2.1 Python

As a dynamic interpreted scripting language with thousands of high quality libraries and a short learning curve, Python has become one of the most dominant modern programming languages across several subdomains of computing. The language itself is characterized mainly by its simple approach to writing readable, practical, and user-friendly code [13], used by many as a "glue" to easily join high performance computing libraries. The role of Python as a mediator between optimized C libraries and human understanding, especially in the machine learning space, is adjacent to the role of documentation in expressing the intention and meaning behind code, rather than focusing on the small details of the code itself.

2.1.1 Python and Types

The overarching data model behind Python is that everything is either an object or a relation between objects. Almost all objects in Python are used by reference, and each object contains an identifier (memory address), an immutable type, and then the actual value itself. In contrast to other scripting languages (namely JavaScript), an immutable type means that Python will never implicitly recast the type of an object to fit the expected types of functions or methods (ie. it is strongly typed).

Although Python is strongly typed, its lack of a static typing system serves as a double edged sword for readability and reliability. With small programs, some developers may prefer to just "get things done", making use of whatever combination of builtin types are available. A mixture of sets, dictionaries, and lists with unrestricted element types are all too common in these types of scripts. However, assigning proper data types to work with various pieces of information

is essential to the longevity and maintenance of large scale projects. Since the introduction of optional static type hints via a library in Python 3.5 and increased support through Python 3.8, a 2022 survey [4] found a steady increase in the number of GitHub repositories making use of static type hints.

Aside from granting linters more information to statically analyze the program and warn developers of possible errors, types also provide programmers themselves with useful information regarding their program. As reinforced by the philosophy of TypeScript as a type-hinted version of JavaScript, data types serve as an important channel of communication between programmers. An additional finding of the 2022 survey [4] showed that typecheck errors in the linters did not prevent a large amount of code from being pushed, but were still found to be useful outside of the linter.

Fundamentally, Python places little restriction on the programmer's actions. For example, while data privacy within Python objects is theoretically possible through name mangled fields (beginning with double-underscores), there is no real way to completely restrict access to an object's fields so long as that object exists in the code. Access to deeper implementation details, like a list of current variables, list of fields in any variable, and even interpreted bytecode is possible through the use of magic methods and variables (denoted by two underscores at the beginning and end of the identifier).

2.1.2 Example: A Linked List

Since Python is primarily an OOP language, the most common form of expressing data types is through the use of classes. A typical linked list data type in Python 3.11 would look something like:

```
from __future__ import annotations
from typing import Generic, TypeVar

T = TypeVar("T")


class LinkedList(Generic[T]):
    def __init__(self, value: T, next: LinkedList[T] | None = None) -> None:
        self.value = value
        self.next = next

    def append(self, new_value: T) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = LinkedList(new_value)

    def concat(self, other: LinkedList[T]) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = other
```

Note that Python 3.12 and later make some major changes involving scoping and the syntax of generic types [14].

The above `LinkedList` class expresses the recursive linking nature of the data type while demonstrating commonly used side effects of OOP methods. Note that the return type for both `append` and `concat` is `None`. Instead, the data contained within the `self` object is mutated in place as needed to achieve the desired functionality.

2.1.3 Pydantic: A data validation library

While native Python works well for simple data types like linked lists, type hints do not provide any guarantees that the type of value, for example, will be consistent across all elements in the list. So, while it does serve as a useful notation tool for documenting the structure of a data type to other programmers, there are no insurance that the runtime values for these objects are consistent with the type annotations. As with many topics in Python, a convenient library by the name of Pydantic will enable both type hints and data validation for those types *at runtime*. The Pydantic version of our linked list will now look something like:

```
import pydantic

from __future__ import annotations
from typing import Generic, TypeVar

T = TypeVar("T")


class LinkedList(pydantic.BaseModel, Generic[T]):
    value: T
    next: LinkedList[T] | None = None

    def append(self, new_value: T) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = LinkedList(value=new_value)

    def concat(self, other: LinkedList[T]) -> None:
        l = self
        while l.next is not None:
            l = l.next
        l.next = other
```

For this simple data type, only the init function changes. However, under the hood, Pydantic will now ensure that the type of `value` within this linked list remains consistent across all elements, throwing an exception otherwise. Pydantic's version of a class has the ability to transform an OOP construct into one more closely related to functional programming. If desired, the use of decorators above field names or methods (denoted by `@decorator`) can restrict mutability, modify constructors, and even change the data validation process to ensure objects are of the correct type.

2.1.4 Conclusion

Python's high-level nature, excellent documentation, library support, and exposure of underlying architectures when needed combine to create a powerful, flexible, and productive language. Although many aspects of the language are initially hidden or abstracted for the sake of convenience, Python allows programmers to dig deeper into the implementation when necessary and extend the language in a manner best suited for a specific task.

2.2 TypeScript

In JavaScript, types are very loosely bound, and many operations/functions will silently convert types at runtime. The silent recasting of JavaScript objects combined with unintuitive defaults can result in very confusing results for the uninitiated. For example, the expression `1 + "1"` returns a value of 11 and the expression `[5, 11, 2].sort()` returns `[11, 2, 5]`. In both expressions, the numerical values are converted into strings before the operation is performed.

TypeScript, as an extension of JavaScript with types, is similar to Python in the sense that both languages started with dynamic typing and patched on static types after the fact, and both aim to resolve issues caused by dynamic typing. TypeScript (and Python) use types not only for more assurance in program correctness, but also for the sake of convenience. When making full use of the available typing systems, text editors and IDEs are able to assist in new code development through intellisense and autocompletion of methods and variables. Types of objects and documentation for those types additionally appear on mouse-hover, providing fast in-context knowledge.

The below list outlines key properties of types in TypeScript.

- Types are treated as sets of values, where types can be part of multiple sets, a union of sets, or rarely even an intersection of sets.
- Types are structural. Any object containing the required fields needed to be a member of a certain type will be considered an object of that type, even if a different type name has been assigned to it. Further, if an object has additional fields not specified by the type, the extra fields will be ignored in the typechecking process. Essentially, as long as an object meets the minimum fields required by the specified type, the object is assumed to be a member of that type.
- TypeScript allows any literal string to act as a subset of its respective type, with the only allowed value being the literal itself. For example, a field with the type ``` ``name'' ``` will only accept the value of "name". However, a field with the type `String` will accept any string object.

In comparison to algebraic data types often seen in functional languages, TypeScript replaces inductive data types with discriminated union sets. For example, our linked list type may look something like:

```
type LinkedList<T> =
        | { kind: "nil" }
        | { kind: "cons"; value: T; next: LinkedList<T> };

function append<T>(self: LinkedList<T>, new_value: T): LinkedList<T> {
        if (self.kind === "nil") {
                return { kind: "cons", value: new_value, next: { kind: "nil" } };
        }

        let l = self;
        while (l.kind === "cons" && l.next.kind !== "nil") {
                l = l.next;
        }
        l.next = { kind: "cons", value: new_value, next: { kind: "nil" } };
        return self;
}

function concat<T>(self: LinkedList<T>, other: LinkedList<T>): LinkedList<T> {
        if (self.kind === "nil") {
                return other;
        }

        let l = self;
        while (l.kind === "cons" && l.next.kind !== "nil") {
                l = l.next;
        }
        l.next = other;
        return self;
}
```

Because TypeScript allows literals to act as types, the only valid values in the "kind" field of an object represented by the `LinkedList` type are our constructor names: "nil" or "cons". Although TypeScript treats every object as a reference, similar to Python, we have not used side effects in the `append` or `concat` functions. Instead, we explicitly accept the linked list we want to transform as input, perform the required action, and return the head of the linked list in the output.

For a non-inductive approach to the LinkedList data type, we can use TypeScript's interfaces to create something that looks much more like the Pydantic version:

```
interface LinkedList2<T> {
        value: T;
        next: LinkedList2<T> | null;
}

function append2<T>(self: LinkedList2<T>, new_value: T): LinkedList2<T> {
        let l = self;
        while (l.next !== null) {
                l = l.next;
        }
        // Structural typing - No need to refer to LinkedList2 name explicitly
        l.next = { value: new_value, next: null };
        return self;
}
```

```
function concat2<T>(
        self: LinkedList2<T>,
        other: LinkedList2<T>,
): LinkedList2<T> {
        let l = self;
        while (l.next !== null) {
                l = l.next;
        }
        l.next = other;
        return self;
}
```

Finally, a pure OOP version of our LinkedList type, using classes, can be formed. Note that the append and concat functions are now methods with side effects:

```
class LinkedList3<T> {
        value: T;
        next: LinkedList3<T> | null = null;

        append3(new_value: T) {
                let l: LinkedList3<T> = this;
                while (l.next !== null) {
                        l = l.next;
                }
                l.next = new LinkedList3();
                l.next.value = new_value;
        }

        concat3(other: LinkedList3<T>) {
                let l: LinkedList3<T> = this;
                while (l.next !== null) {
                        l = l.next;
                }
                l.next = other;
        }
}
```

2.3 C++

2.4 Rust

2.5 Dafny

2.6 Haskell

2.7 Agda

2.8 APL

2.9 UML

2.10 Event-B

2.11 SETL

2.12 Bend

2.13 Exploration of Conceptual Data Types

3 Error Handling and Undefinedness

4 A Word on Usability

5 Conclusion

References

[1] Jean-Raymond Abrial. *Modeling in Event-B : system and software engineering*. eng. Cambridge: Cambridge University Press, 2010. ISBN: 9781139195881.

[2] *C++ reference*. 2024. URL: https://en.cppreference.com/w/ (visited on 02/07/2025).

[3] Wikipedia contributors. *Type class — Wikipedia*. 2025. URL: https://en.wikipedia.org/wiki/Type_class (visited on 02/07/2025).

[4] Luca Di Grazia and Michael Pradel. "The evolution of type annotations in python: an empirical study". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, pp. 209–220. ISBN: 9781450394130. DOI: 10.1145/3540250.3549114. URL: https://doi.org/10.1145/3540250.3549114.

[5] *Event-B and Rodin Documentation Wiki*. 2024. URL: https://wiki.event-b.org/index.php/Main_Page (visited on 02/07/2025).

[6] A. Hsu. *Is APL Dead?* 2020. URL: https://www.sacrideo.us/is-apl-dead/ (visited on 02/07/2025).

[7] Kenneth E. Iverson. "Notation as a tool of thought". In: *Commun. ACM* 23.8 (Aug. 1980), pp. 444–465. ISSN: 0001-0782. DOI: 10.1145/358896.358899. URL: https://doi.org/10.1145/358896.358899.

[8] S. Klabnik and C. Nichols. *The Rust Programming Language, 2nd Edition*. No Starch Press, 2023. ISBN: 9781718503106. URL: https://books.google.ca/books?id=SE2GEAAAQBAJ.

[9] *LearningAPL*. 2024. URL: https://xpqz.github.io/learnapl/intro.html (visited on 02/07/2025).

[10] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press Series. No Starch Press, 2011. ISBN: 9781593272838. URL: https://books.google.ca/books?id=R2RbBAAAQBAJ.

[11] J.J. Martin. *Data Types and Data Structures*. Prentice-Hall International series in personal computing. Prentice-Hall International, 1986. ISBN: 9780131959835. URL: https://books.google.ca/books?id=554ZAQAAIAAJ.

[12] *OMG Unified Modeling Language*. Version 2.5.1. Object Management Group. 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF (visited on 02/07/2025).

[13] *PEP 20 - The Zen of Python*. Python Software Foundation. 2004. URL: https://peps.python.org/pep-0020/ (visited on 02/07/2025).

[14] *PEP 695 - Type Parameter Syntax*. Python Software Foundation. 2022. URL: https://peps.python.org/pep-0695/ (visited on 02/07/2025).

[15] *Pydantic*. Python Software Foundation. 2025. URL: https://docs.pydantic.dev/latest/ (visited on 02/07/2025).

[16] A. Rauschmayer. *JavaScript metaprogramming with the 2022-03 decorators API*. 2024. URL: https://2ality.com/2022/10/javascript-decorators.html (visited on 02/07/2025).

[17] D. Rosenwasser. *Announcing Typescript 5.0*. 2024. URL: https://devblogs.microsoft.com/typescript/announcing-typescript-5-0/#decorators (visited on 02/07/2025).

[18] *The Dafny Programming and Verification Language*. 2024. URL: https://dafny.org/ (visited on 02/07/2025).

[19] *The Python Language Reference (3.13.2)*. Python Software Foundation. 2025. URL: https://docs.python.org/3/reference/ (visited on 02/07/2025).

[20] *The Rust Reference*. 2024. URL: https://doc.rust-lang.org/reference/ (visited on 02/07/2025).

[21] P.G. Thomas, H. Robinson, and J. Emms. *Abstract Data Types: Their Specification, Representation, and Use*. Oxford applied mathematics and computing science series. Clarendon Press, 1988. ISBN: 9780198596639. URL: https://books.google.ca/books?id=u61QAAAAMAAJ.

[22] *Type — HaskellWiki*. HaskellWiki. 2024. URL: https://wiki.haskell.org/index.php?title=Type (visited on 02/07/2025).

[23] *Typeclassopedia — HaskellWiki*. HaskellWiki. 2022. URL: https://wiki.haskell.org/index.php?title=Typeclassopedia (visited on 02/07/2025).

[24] *Typescript Documentation*. 2024. URL: https://www.typescriptlang.org/docs/ (visited on 02/07/2025).

[25] *Welcome to Agda's documentation!* Version 2.7.0.1. 2025. URL: https://agda.readthedocs.io/en/v2.7.0.1/# (visited on 02/07/2025).