Dynamic Pattern Matching with Python

Tobias Kohn
tobias.kohn@cl.cam.ac.uk
University of Cambridge
Cambridge, UK

Guido van Rossum
guido@python.org
Python Software Foundation
USA

Gary Brandt Bucher, II
brandtbucher@gmail.com
Research Affiliates, LLC
Newport Beach, CA, USA

Talin
viridia@gmail.com
USA

Ivan Levkivskyi
levkivskyi@gmail.com
Dropbox Ireland
Dublin, Ireland

Abstract

Pattern matching allows programs both to extract specific information from complex data types, as well as to branch on the structure of data and thus apply specialized actions to different forms of data. Originally designed for strongly typed functional languages with algebraic data types, pattern matching has since been adapted for object-oriented and even dynamic languages. This paper discusses how pattern matching can be included in the dynamically typed language *Python* in line with existing features that support extracting values from sequential data structures.

CCS Concepts: • **Software and its engineering → Patterns**.

Keywords: Pattern Matching, Python, Dynamic Language

ACM Reference Format:
Tobias Kohn, Guido van Rossum, Gary Brandt Bucher, II, Talin, and Ivan Levkivskyi. 2020. Dynamic Pattern Matching with Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '20), November 17, 2020, Virtual, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3426422.3426983

1 Introduction

With the increasing size and complexity of data, it becomes ever more important to discover structures in data and extract relevant information. Various computing tools have evolved to address this. Parsers, for instance, are highly efficient in discovering structure in textual data and turn linear streams of characters into trees of specialized symbols.

The programming language community has long risen to the challenges of increased data complexity through the design of abstract data structures such as algebraic data types or objects and classes to not only capture the structure of data but also make the underlying information accessible and available for manipulation. In addition to these data structures, however, a programming language must also provide the means to process the data. One such tool is *pattern matching*, supporting two objectives: decomposing complex objects into meaningful parts for further processing, and branching on the overall structure of a data object rather than on specific values.

Pattern matching has first emerged in functional languages with strict static typing. With the advent of object-oriented programming, pattern matching has increasingly also found its way into object-oriented languages. The major hurdle to overcome was the discrepancy between the concepts of encapsulation and abstraction in OOP as opposed to having full access to all data for discovering structure and extracting partial information in pattern matching.

In recent years, data processing and analysis has shifted towards the use of dynamic languages, combining the power of highly optimized libraries with the ease of a simple and accessible language. *Python*, in particular, has become a de-facto standard for many big data and machine learning tasks. Despite Python's range of data structure and processing facilities, however, it only provides limited means of pattern matching, namely for extracting elements from sequential data. This paper thus reports on a project to extend Python's pattern matching capabilities through the introduction of a

Program 1 Two examples where pattern matching is used to define functions for the factorial and the sum, respectively.

```
def fact(arg):
    match arg:
        case 0 | 1:
            f = 1
        case n:
            f = n*fact(n-1)
    return f
```

```
def sum(seq):
    match seq:
        case []:
            s = 0
        case [hd, *tl]:
            s = hd+sum(tl)
    return s
```


DLS '20, November 17, 2020, Virtual, USA

ACM ISBN 978-1-4503-8175-8/20/11.
https://doi.org/10.1145/3426422.3426983

Program 2 An example of Python's *match*-statement in action, as it branches on the type and structure of the subject `node` and binds local variables to nodes in the tree-structure.

```
def simplify(node):
    match node:
        case BinOp(Num(left), '+', Num(right)):
            return Num(left + right)
        case BinOp(left, '+' | '-', Num(0)):
            return simplify(left)
        case UnaryOp('-', UnaryOp('-', item)):
            return simplify(item)
        case _:
            return node
```

fully fledged structure for conditional branching and extraction of information, based on the structure of objects.

Python organizes data primarily in form of a directed labeled graph where all data values are represented by objects that act as vertices in the graph. Apart from an identity and attributes that refer to other objects, an object has no actual structure. Pattern matching in Python is thus primarily a task of matching a sub-graph. This differs from both the traditional approach within the setting of strongly and statically typed functional languages, as well as the concept of data abstraction and encapsulation of typical object oriented languages, and brings about some unique challenges. The class of an object, for instance, provides only limited information about the object's structure and its relationship to other objects. We address this issue by extending classic constructor patterns with the means to further specify the required structure.

The focus of this paper is on the syntax and semantics of pattern matching as an enhancement to the Python language. This differs from existing work on pattern matching in Python [13, 14] that approached the subject from an algorithmic and mathematical point of view. In contrast to pattern matching in other dynamic languages [3, 11], we implement it as a compiled feature of the language in the tradition of static languages [6, 22] rather than introducing patterns as first-class objects.

The primary contribution of this paper is a discussion of pattern matching in the context of the dynamically typed language Python. It summarizes theoretical considerations behind the design of the language enhancement. Ideally, patterns could be modeled as natural extensions of existing language features such as assignments or type annotations. However, we will argue that this is not entirely feasible in Python. A working proof of concept with support for pattern matching in Python is available and the core elements of our design have been proposed for integration into Python [4], although issues of the implementation are beyond the scope of this paper.

2 Overview

Python offers sophisticated features to deal with sequential data, including *iterable unpacking*, which extracts individual values from a sequence and binds them to variables. However, not all data is best described sequentially: sometimes the graph structure imposed by objects is a better fit. Moreover, due to Python's dynamic nature, the data provided to a specific part of the code might take on many different shapes and forms.

Two questions naturally arise out of this situation:

1. Can we extend a feature like iterable unpacking to work for more general object and data layouts?
2. How do we support selecting a specific data processing strategy based on the structure of the data provided?

Both questions can be answered by *pattern matching*, which we understand as providing a structural template (the *pattern*) and an associated action to be taken if the data fits the template. The template typically contains variables that act as formal parameters for the associated action.

While pattern matching has found its way into various object-oriented and even dynamic languages [3, 6, 7, 11, 22], our aim is to find a design that works particularly well with Python's existing syntax and semantics. This means, e.g., that the syntax of patterns should represent a natural extension to iterable unpacking or formal parameters.

We found that only parts of iterable unpacking syntax can be directly generalized to patterns. In particular, side effects should be avoided due to the conditional nature of patterns. Pattern matching can thus only operate on actual sequences rather than iterables, and the assignment targets are restricted to local variables, excluding attributes or item assignments. Although Python is a dynamic language, we strive for patterns to express static structures and use *guards* to express additional dynamic constraints where necessary.

Programs 1 and 2 provide a first impression of the overall syntax and structure of pattern matching in Python. The **`match`** keyword is followed by an expression that yields the *subject* to be matched against the individual patterns, together with a series of case clauses. Combining various patterns in sequences, alternatives, or through nesting allows them to express arbitrarily complex tree structures.

3 Pattern Matching

Pattern matching as supported by a programming language is a tool for extracting structural and structured information from given data (the *subject*). It is based on a hypothesis that the data in question follows a specific structural *pattern*, together with an action to be performed on the premise that the hypothesis holds. A hypothesis holds if there is a substitution $\sigma = \{x_i \mapsto v_i\}$ of variables to values for the pattern P, such that σP correctly 'matches' the subject, i.e. $\sigma P = \pi(s)$ for some well-defined projection π. By combining several potential structural patterns P_j and their associated

actions into a common selection statement, the system can chose appropriate action based on the structure of the input data, although a single hypothesis might be sufficient if the compiler can statically verify it.

Conditional pattern matching allows the system to take alternative action if a specific hypothesis does not hold, whereas in the case of unconditional pattern matching the correct execution of a program depends on the validity of a hypothesis. An assignment such as `(a, b) = e` is an instance of unconditional pattern matching that will raise an exception if e is not sequence of two elements. The **`case`** statements in Program 2, on the other hand, are an example of conditional matching (although the overall **`match`** structure forms a combined hypothesis that might still fail).

Arguably the most common application of pattern matching occurs in function invocations, where the list of formal parameters specifies the structural hypothesis that any actual arguments passed to the function must fulfill. In this scenario, types and the number of values are used as structural information and build the pattern P. The actual arguments passed to the function form the substitution σ that maps parameter names to argument values. A first step towards more general pattern matching is then naturally to introduce function overloading, whereby the function body to be executed is chosen based on the types and count of actual arguments provided at the call site.

In the context of textual data, regular expressions and context-free grammars have become a widely accepted standard for formulating structural hypotheses. Instead of built-in support in programming languages, regular expressions and grammars are usually written separately and compiled to efficient parsers or finite state machines, respectively, that can then be included into a software application. Nonetheless, there is often some partial syntactic support for regular expressions in various programming languages.

Of particular interest for the present work are syntactic selection structures, offering an ordered set of hypotheses and associated actions to be applied to a specific data subject, as presented in Figure 1, Programs 1 to 3 and 5. Syntactically, these structures might resemble switch tables, where an action is chosen based on the (ordinal) value of a specific subject rather than its structure. In contrast to such linear switch-tables, pattern matching is typically performed based on a decision tree, selecting the first hypothesis that holds for the subject. In general, it is not practical or realistic to expect the patterns to form a disjoint partition of the possible data space, hence the first-to-succeed rule.

3.1 Objectives

In general, pattern matching pursues two objectives:

1. **Validate the structure/hypothesis:** ensure that the structure of the data subject conforms to a given pattern template;

Program 3 Pattern matching is used here to define a function that sums the elements of a list in *Scala* (on the left) and *F#* (on the right), respectively (cf. Program 1). The parameter *list* is the subject, `x :: rest` and _ are patterns, while `x + sum(rest)` is the handler for the first pattern.

```
def sum(list: List[_]) =
  list match {
    case x :: rest =>
      x + sum(rest)
    case _ =>
      0
  }
```

```
let sum list =
  match list with
  | x :: rest ->
      x + sum rest
  | _ ->
      0
```

2. **Bind variables:** extract specific information from the data subject and make it available by storing it in (local) variables.

More formally, a pattern P can be thought of as a tree structure consisting of variables and constructors (cf. Section 4). Given a data subject s and a pattern P together with a projection π, the objective of pattern matching is then to find a substitution σ that assigns a value to each variable in P, such that $\sigma P = \pi(s)$, or report that no such substitution σ exists. Functional languages with algebraic data types are based on exact equality $\sigma P = s$, but the abstraction employed in object-oriented languages requires the pattern P to be extended with an associated projection π. For instance, if a pattern P matches instances of a class C, we expect the pattern to also match instances of subclasses of C. Moreover, an instance matched by P might have additional data in private fields, say, that are neither expressed by, nor relevant to the pattern P. The projection π thus expresses that object-oriented pattern matching does not match objects exactly, but rather well-defined representations thereof.

In statically and strongly typed languages, the structural hypothesis may be checked or verified by the compiler during compile time, particularly if the program code offers only a single pattern to which the data has to conform (i.e. when pattern matching is used in assignments or function application, say). In this case, the compiler can then generate code to directly build the substitution σ.

After a successful match, subsequent processing of the data frequently depends on the pattern that matched the subject s, particularly if the sets of variables used in each pattern differ. A pattern P_k may therefore be equipped with a specific handler h_k that is executed as $h_k(\sigma_k)$ iff a matching substitution σ_k exists. This permits the construction of a *multi pattern matching* structure, which is an ordered sequence $\langle P_k, h_k \rangle$ of patterns and handlers. Given a subject s, the system then finds the smallest index k such that $\sigma_k P_k = \pi_k(s)$ and executes the handler h_k (cf., e.g., [20]):

$$\frac{\exists \sigma_k : \sigma_k P_k = \pi_k(s) \qquad \forall i < k . \forall \sigma_i' : \sigma_i' P_i \neq \pi_i(s)}{h_k(\sigma_k)}$$

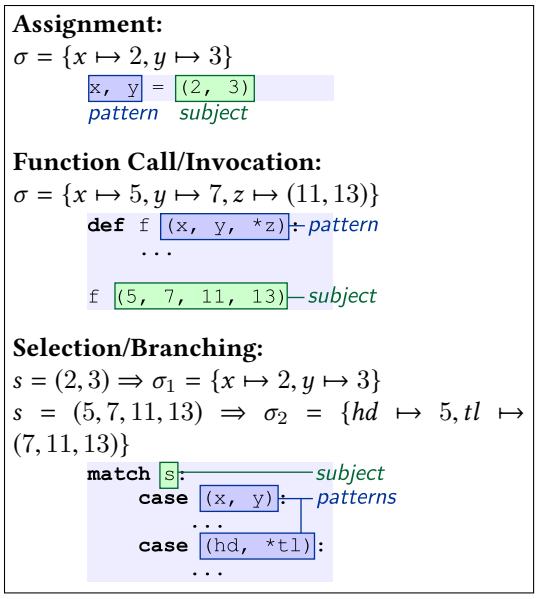

Figure 1. Three different forms of pattern matching. In each case the succeeding substitution σ is given. Python already supports limited pattern matching of sequential data in assignments and function calls. Our work adds a selection statement for pattern matching. '. . .' is an action to be taken if the respective pattern successfully matches the subject.

Note that pattern matching clearly depends on the definition and implementation of data comparison and equivalence.

3.2 Forms of Pattern Matching

We identified three different main forms of how pattern matching generally occurs in a programming language: in assignments, function invocations, and for branching in selection structures (Figure 1).

Assignment. Assignments can support pattern matching as a means to decompose a complex data structure and bind values encoded in the data structure to individual local variables, rather than extracting the desired piece of data from the structure manually. It constitutes an error if the underlying hypothesis about the structure of the subject does not hold. To some degree, pattern matching in assignments is a common feature shared by many programming languages and often written along the lines of `val (a, b) = t`.

Python supports the extraction of elements from sequential data. The subject to be matched needs to be iterable and yield a valid number of elements to fulfill the pattern. Nested patterns are possible and a star marks a binding target as accepting a sequence of any length rather than a single element. The following example will assign 2 to a, the list $[3, 5, 7]$ to b, and the letters `'P'` and `'R'` to x and y, respectively.

```
(a, *b, (x, y)) = [2, 3, 5, 7, "PR"]
```

If the subject on the right hand side is not iterable or contains too many or too few elements to fit the pattern, an exception is raised.

Function dispatch. The formal parameters of a function can be seen as a sequence pattern that must be matched in order to execute the function body. In this case the function body acts as the handler associated with the pattern as specified by the parameters. Statically typed languages usually associate a type constraint with each parameter. If the language supports function overloading, the function effectively becomes a multi pattern matching structure, offering a set $\{\langle P_k, h_k \rangle\}$ of patterns and associated handlers. Due to the declarative nature of functions in many languages, this set is not ordered, which leads to the constraint that each possible subject s must match exactly one pattern P_k.

As a dynamic language, Python does not have type constraints on function parameters, nor does it support function overloading (although it is possible to emulate function overloading, usually through the definition of a class [15, 24], cf. Program 4). Similar to assignments, it is possible to have parameters that capture an arbitrary number of argument values as a sequence. In combination with pattern matching, this provides a means to emulate function overloading as demonstrated in Program 5.

Program 4 The visitor pattern as implemented in Python's standard module *ast*. The *visit*-method creates a specialized method name based on the node's class and then looks up that method using `getattr`. The same principle allows programs to emulate function overloading.

```
class NodeVisitor(object):
    def visit(self, node):
        method = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method,
                          self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        ...
```

Selection structures. Branching based on pattern matching allows programs to choose an action based on the structure and value of the match subject. It is similar to function overloading, but allows the individual handlers of the patterns to share a common (local) scope, and imposes an order on the patterns involved. The patterns need therefore not necessarily be a disjoint partition of the subject's value space. Examples of selection structures in *Scala* and *F#* are given in Program 3. The addition or extension of such selection structures is what can usually be found in the literature on *pattern matching* in a narrow sense [3, 6, 9, 12, 16, 22].

At the moment (that is, as of version 3.9), Python does not have a pattern matching selection structure. Conditional

Program 5 Pattern matching is used here to emulate function overloading (slightly abbreviated): `arg` is a sequence containing all positional arguments whereas the dictionary `kwargs` contains keyword arguments. Note the guard in the first case clause to add a dynamic (non-structural) test.

```
def create_rectangle(*args, **kwargs):
   match (args, kwargs):
      case ([x1,y1,x2,y2], {}) if x2-x1 == y2-y1:
         return Square(x1, y1, x2-x1)
      case ([x1,y1,x2,y2], {}):
         return Rect(x1, y1, x2, y2)
      case ([(x1, y1), (x2, y2)], {}):
         return Rect(x1, y1, x2, y2)
      case ([x, y], {'width': wd, 'height': ht}):
         return Rect(x, y, x+wd, y+ht)
```

selection of an action is either implemented through **`if-elif-else`** chains or possibly using dictionaries/maps. The concept of branching based on the class/constructor of a subject is usually implemented using the visitor pattern as indicated in Program 4. Our selection structure is thus an effectively novel addition to and enhancement of Python.

Other variants. The concept of a *partial function* as in, e.g., *Scala* [18] or *Grace* [11] extends a function so that it can be queried as to whether it accepts a given list of arguments. That is, a partial function's domain is reified and made explicit. While Scala uses non-exhaustive pattern selection to define partial functions, Grace uses the reverse approach and builds pattern selection out of partial functions.

Some languages support pattern matching without explicit syntactic support. *Newspeak* [7], e.g., implements pattern matching through an object-oriented message passing interface and the authors note that patterns must be first-class objects in such a setting. Although first-class patterns are possible in Python as evidenced by regular expressions and *PyMatch* [14], we follow a more syntax-oriented approach where pattern matching is provided by the compiler and runtime environment. However, the concept of active patterns (Section 5) could be used to emulate such an approach.

3.3 Guards

Not all types of constraints can be adequately covered by static (context-free) patterns, where a *static pattern* P can be seen as a (state-less) function that maps a subject s either to a substitution σ or to a special value $\emptyset$ to indicate that s does not match the pattern.

For instance, static patterns cannot express that two values in a sequence, say, must be equivalent (Figure 2). If we consider such a pattern, which takes the general form of (x, P_x), we find that the pattern P_x needs access to the variable x and its binding. However, in a purely static setting, there is no information flow from x to P_x. In the tree structure of

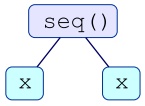

Figure 2. The pattern (x, x) naturally forms a tree without any flow of information between the isolated variables x.

static patterns [10], information flows only vertically, but not horizontally (cf. Section 4). By extending the static patterns so that patterns accept not only the match subject but also an additional match context as an argument passed in, such an information flow can be established explicitly [21]. Another approach is to establish a quasi-global scope for the entire pattern where variable bindings are not returned as a substitution, but rather stored in the common scope [3].

Alternatively, dynamic constraints can be placed on the substitution σ *after* the static pattern succeeded. Such constraints are known as *guards* (cf. Program 5). The above pattern of two equal values could then be written as, e.g., '(x, y) if $x = y$'. The guard can be any valid expression such that $\Gamma, \sigma \vdash e$ evaluates to either true or false (Γ is the context in which the pattern matching is executed). The extended pattern '$\tilde{P} = P$ if e' succeeds iff P succeeds with the substitution σ and $\Gamma, \sigma \vdash e$ evaluates to true.

The evaluation operator \$ in Thorn [3] implements a concept similar to guards by allowing arbitrary expressions to be evaluated in a context Γ, σ', where σ' denotes the substitutions established up to this point.

The clear separation of a pattern into a static pattern and an associated guard allows, in principle, for the static pattern to be heavily optimized by reordering, caching or parallelizing the evaluation of subpatterns. The separation of static patterns and dynamic guards is also reflected in parsing. Context-free grammars cannot capture the equality of two symbols. The compiler thus needs to check explicitly whether a name occurs more than once in a list of parameters.

3.4 Scope and Binding

Variables assigned in a case clause are usually only visible in the associated handler. With strong static typing, this is critical because the patterns of different case clauses might bind values of differing types to variables that share a common name. In dynamically typed languages, this is less of a concern since the variables themselves are untyped. However, different patterns are likely to bind different sets of variables. Depending on whether pattern P_1 or P_2 matched successfully, a variable x might be bound to a value, or remain undefined and thus inaccessible (trying to read from an undefined variable throws an exception). Clearly, in order for variables to be well defined, each case clause with its handler must be a scope of its own.

The issue of undetermined variable bindings also occurs with partly successful matches. This can be illustrated with

the simple example: `case x if x > 0`. The pattern itself is irrefutable and simply binds the subject to the variable *x*. This happens necessarily before the comparison in the guard is evaluated. If *x*, however, is not a positive value, the overall pattern fails and *x* should remain unbound.

In Python, such scoping rules are difficult to enforce or implement (cf. Section 6.1). Any actual scope would have to be implemented internally as a function (as is the case with list comprehensions) with some unpleasant consequences. On the one hand, a pattern matching statement could no longer directly influence control flow by means of, e.g., `return` or `break` as neither of these statements would refer to the context the pattern matching statement is located, but rather to the functional scope of the successful case clause. On the other hand, any assignment to local variables inside a case clause would remain limited to that case clause without special markers (e.g. `nonlocal`). Hence, both control and data flow from a case clause to its surrounding context would be severely hindered and limited. Since Python is statement- and not expression-oriented, this would effectively reduce the application of pattern matching statements to being the single statement inside a function's body.

Based on the observation that the body of a loop does not form a proper scope, either, but that the loop variable of a *for*-loop remains accessible in the surrounding context, and in line with the semantics of assignment expressions [1], we decided not to define special scopes for case clauses. Any variables bound during a successful pattern match will thus also remain accessible after the match statement—even if the pattern eventually failed due to a guard evaluating to false.

3.5 Syntax

The overall syntax of the pattern matching statement as shown in Program 6 includes a top-level *match* statement, whose body is a non-empty sequence of *case clauses*. The design follows Python's general syntax in that each compound statement starts with a unique keyword that identifies the statement (cf. Section 6). `match` and `case` are 'soft' keywords that can be used as regular names in different contexts, so as to maintain backward compatibility with existing code.

Program 6 The grammar of the added *match* statement with *case* clauses, slightly simplified.

⟨*match_stmt*⟩ ::= '`match`' ⟨*expr*⟩ '`:`' NEWLINE INDENT ⟨*case_block*⟩+ DEDENT

⟨*case_block*⟩ ::= '`case`' ⟨*pattern*⟩ [⟨*guard*⟩] '`:`' ⟨*block*⟩

⟨*guard*⟩ ::= '`if`' ⟨*expr*⟩

3.6 Related Work

Pattern matching originally evolved as a tool to define recursive equations in the context of functional programming languages with algebraic data types [2, 5, 23]. Wadler [27] pointed out early on the apparent conflict between pattern matching and (data) abstraction and proposed *views* as a means to combine pattern matching with data abstraction.

Following the advent of object-oriented programming built on the notion of abstraction, research started to addressed the issue of combining an object-oriented design with pattern matching. While various proposals to extend *Java* [9, 12, 16] did not find their way into the language, *Scala* [6] and *F#* [22], for instance, both introduced pattern matching in a statically typed object-oriented context.

A different approach [7] introduced pattern matching to the message-based object-oriented language *Newspeak* in the tradition of *Self* and *Smalltalk*. In this context, patterns become necessarily fist-class objects (rather than syntactic structures) that react to messages querying whether a given subject *s* matches. Structural patterns are expressed through message keywords rather than constructors and types.

Pattern matching in *Grace* [11] built on this idea of first-class objects, together with ideas taken from Scala: partial functions support patterns to annotate parameters and can be queried as to whether a given list of argument falls into the domain of the partial function. The match-statement is then a sequence of partial functions, each defining both its domain and bindings in the form of parameters and patterns, as well as the associated handler.

In contrast, pattern matching in *Thorn* [3] follows the principle that patterns 'should be allowed wherever variables are bound'. Structural matching is then supported through extensive and comprehensive syntax, which is in strong contrast to Grace's design that sought to implement pattern matching with a minimum of dedicated syntax.

The work we present in this article draws primarily on the designs from *Scala*, *F#* and *Thorn*: Patterns are syntactic constructs rather than first-class objects with emphasis on variable binding. However, instead of static types as in Scala, say, Python needs to express constructor patterns through dynamically resolved class objects.

4 Expressing Patterns in Python

In its basic form, a pattern is either a variable or a constructor applied to a possibly empty sequence of subpatterns [21]. Pattern thus naturally form trees [10]. A variable matches any subject and binds the subject to a given name (with the exception of the wildcard variable, which forgoes the binding). Constructors express (structural) constraints which a subjects needs to fulfill in order to match the pattern. Nullary constructors such as literals and constants may form a constraint on the subject's identity or value. In strict implementations, a constant would only match itself, e.g., the constant 1.0 would only match the floating point value 1.0, whereas in equivalence-based matching a constant c matches any value v with $c = v$ according the rules of the programming language.

Conditional pattern matching often also allows a pattern P to be a sequence of alternative patterns P_i, where P matches a subject if at least one subpattern P_i matches the subject. Formally, we could define $P_1|P_2 \Rightarrow h$ with patterns P_1 and P_2 and the handler h to be shorthand for $P_1 \Rightarrow h, P_2 \Rightarrow h$. However, it is often convenient to fully include alternatives in the definition of a pattern so that patterns may be factored out and we can write, e.g., `case Node('+'| '-', lt, rt)` rather than `case Node('+', lt, rt) | Node('-', lt, rt)`.

Summary. The following table summarizes the patterns supported by our implementation. A detailed discussion of the semantics and rationale for each of the patterns follows.

Variable	v (any name)
Wildcard	_ (underscore)
Literal	1, 2.7182818, `"abc"`
Named constant	$v.w$ (dotted name)
Sequence	$(P_1, P_2, \ldots, P_n)$ or $[P_1, P_2, \ldots, P_n]$
Mapping	$\{k_1 : P_1, k_2 : P_2, \ldots, k_n : P_n\}$
Alternatives	$P_1\|P_2\|\ldots\|P_n$
Constructor	$C(P_1, P_2, \ldots, a_1 = P_k, a_2 = P_{k+1}, \ldots)$

4.1 Variables

A variable v matches any subject and binds the variable v to the subject. Using a variable v inside a pattern counts as assignment, making v local to the current scope (usually a function body). The value thus assigned to v during a succeeding pattern becomes visible to the entire scope and is not limited to the respective case clause.

The variable binding in a pattern must be unambiguous, i.e. a variable cannot be bound twice by the same pattern. In fact, if a pattern succeeds with a substitution σ, it must assign a single value to each variable v in $Var(P)$. For composite patterns $\tilde{P} = P_1 \oplus P_2$ this means that $Var(P_1) \cap Var(P_2) = \emptyset$ if both P_1 and P_2 must be satisfied to satisfy $\tilde{P}$, and $Var(P_1) = Var(P_2)$ if either P_1 or P_2 can be satisfied to satisfy $\tilde{P}$. Patterns must thus be *effectively* (or *dynamically*) *linear*.

This rule differs from existing iterable unpacking, where a variable can be (re-)bound arbitrarily many times, i.e. the assignment `(v, v) = (e1, e2)` is legal and will, in accordance with Python's strict left-to-right evaluation order, eventually assign e_2 to v. But in the context of 'full' pattern matching, (x, x) could easily be understood as denoting a tuple of two equal values, and the nature of nested structures makes a strict left-to-right semantics less obvious (cf. Section 3.3).

The underscore _ is a legal name in Python without special meaning. In the context of pattern matching, however, _ does not bind any value but rather acts as a wildcard (there is general agreement in the pattern matching literature that the underscore is treated as a wildcard that matches everything but does not bind anything). Hence, $Var(\_) = \emptyset$ and thus $\_ \notin Var(P)$ for any pattern P. This allows programs to reuse _ any number of times in a pattern without violating the requirement of unambiguous name binding.

Note that a subject can also be bound to a variable v using the 'walrus' operator `:=` [1]. The syntax `v := P` matches a subject s iff the pattern P matches s and in case the match succeeds, the variable v is bound to the subject s.

Although iterable unpacking in Python can assign values directly to targets such as attributes and elements in a container, this would be problematic in the context of pattern matching due to possibility of a failed match. Assignment to an attribute `a.b` or an element `a[n]` often causes side-effects and might therefore lead to inconsistent program states. With respect to variables, patterns behave thus more like formal parameters than like iterable unpacking.

Equality. It is tempting to allow a variable v to occur more than once to specify that values occurring at different positions in the subject must be equal. This would support the aforementioned pattern (x, x) to effectively express that the subject is expected to be a tuple of two equal elements.

There are, however two issues to be considered. The patterns would need a context as discussed in Section 3.3 to allow for information exchange between (otherwise unrelated) patterns. The compiler could also create a fresh variable pattern v_i for each occurrence of a variable v and then add guards to ensure $\forall i, j : v_i = v_j$ (which would have to be reflected in the evaluation semantics). In either way, patterns would no longer be *static* and form a context-free tree.

Without any restriction on the types of v, it is generally not decidable whether two objects are equal or how equality should be properly defined (cf., e.g., [8, 10]). For instance, the subject could be a tuple (f, g) of two functions, in which case $f = g$ is clearly undecidable in general. In case of a tuple containing two floating point numbers, minute differences (which usually remain invisible) could equally lead to unexpected behavior. Of course, Python has a clear definition of equality and any two objects are comparable with respect to equality. A function object, for instance, is only ever equal to strictly itself (referential equality). Moreover, guards still allow pattens such as '(x, y) if $x = y$'. However, the comparison for equality is made explicit, clearly visible and not delegated to the 'magic' of pattern matching.

4.2 Literals

A literal ℓ matches a subject s if $\ell = s$ according to the usual equivalence semantics of Python. A literal is either an integer, floating point or complex number, a string, or one of **`True`**, **`False`**, or **`None`**, respectively. Because of Python's equality semantics the floating point number `1.0` and the integer `1` are interchangeable and match the same set of values. A pattern needs to explicitly specify both type and value in order to only match one of these values.

Comparison of floating point numbers is known to be problematic due to rounding errors. *Standard ML* does therefore not support the equivalence operator = on floating point numbers by default, and *Rust* excludes floating point literals

from pattern matching. However, in Python the aforementioned equivalence semantics are already well established.

4.3 Named Constants

Particularly in larger projects, it is customary to define named constants or enumerations/sets for recurring (parametric) values. It would clearly be desirable to allow named constants in patterns as a replacement and extension of literals. However, Python has no concept of a constant, i.e. all variables are mutable (even where the values themselves are immutable). While variables intended as constants are often written in uppercase letters, there is no strict rule enforcing this and exceptions prevail. The compiler has thus no way of identifying which name in a pattern is intended to be a variable, and which a constant value. In other words: the compiler cannot distinguish between names with *store* and those with *load and compare* semantics.

Scala [6, 18] follows the rule that any name starting with an uppercase letter is a constant and thus creates a constraint on the subject, whereas variables as binding targets must start with a lowercase letter. Moreover, all qualified names (i.e. dotted names) are also treated as constant values (in accordance with the local scoping of variables as assignment targets).

Thorn [3] introduces an evaluation operator $ so that `$e` evaluates the expression *e* first and then compares the subject to the resulting value. This allows a program to succinctly express a pattern `[x, $x]` that only matches sequences of two equal elements (cf. Section 3.3). *Elixir*[1] uses the operator `^` (pin operator) to mark names as having load-and-compare semantics, i.e. `^pi = x` will only match if x has the same value as `pi`.

Our design does not use an operator to mark names as load-and-compare constants. However, given that attributes cannot be pattern variables (binding targets), we adopted the rule that a qualified (dotted) name $v.w$ matches any subject that is equal to $v.w$ according to the usual comparison semantics. This allows the convenient use of constants from specific modules or enumerations.

4.4 Sequences

A sequence pattern P consists of a sequence of patterns $P_0, P_1, \ldots, P_{n-1}$. It matches a subject s if the subject supports a sequence function f_s such that for all k with $0 \leq k < n$ the pattern P_k matches $f_s(k)$, and $f_s(j)$ is undefined for $j \geq n$. In addition, a sequence might contain at most one marked variable $^*v$ (including the wildcard *_) that matches a subsequence and binds it to the variable v (Figure 1).

The starred variable allows for patterns like `(x, *_, y)` that match any sequence with at least two elements, where the first and last elements are bound the the variables x and y, respectively. However, patterns like `(x, *_, y, *_)` are not supported, as they quickly require inefficient backtracking.

Python already supports sequence patterns as *unpacking* in assignments, and the starred variable is also supported in formal parameters to define variadic functions, where the starred parameter will bind any actual arguments not otherwise bound, as a tuple. Iterable unpacking in Python can be written either as tuple or list (with no semantic distinction), and can be nested so that, e.g., `(a, [b, c], *d) = e` is legal. Iterable unpacking matches any object *e* that is iterable. In the context of conditional pattern matching, this is problematic, as reading from an iterator changes the iterator's internal state.

While iterators can be used to conceptually implement lazy lists, they provide less structure: the next element cannot be inspected without removing it from the sequence. Nor does an iterator reveal the number of available elements (i.e. the length of the underlying sequence). It is therefore impossible to support iterators as non-atomic subjects (i.e. having sequential structure) in conditional pattern matching. Accordingly, the patterns in our enhanced pattern matching are not a strict superset of existing sequence unpacking.

With respect to the interface protocol, sequences in Python are a special case of mapping structures (this is similar to JavaScript). Any object that supports the mapping protocol implements methods `__len__()` and `__getitem__(key)` to return the number of elements and to return a specific element as identified by its key, respectively. While actual maps permit the key to be any hashable Python object, sequences restrict keys to be integer values. Unfortunately, this distinction is not reflected in the interface. In order to correctly match sequences, it is therefore necessary to check the type of a potential match subject, which restricts sequence patterns to only accept subjects that are known to the system to be sequences (in particular lists and tuples).

Note that strings (either containing textual or binary data) play a special role in that they fully implement the sequence protocol while also being atomic values. Notably, an element of a string is not a char, say, but rather a string of length one, i.e. strings are recursive types with respect to item access. In the context of pattern matching, we decided to focus on strings as atomic values and ignore their sequence aspect. The pattern `(a, b)` will therefore not match a string of length two. This is a somewhat arbitrary, although informed, decision in contradiction to unpacking, which does regard strings as sequences.

Syntactically, sequences can be either written as tuples `(a,b)` (delimited by round parentheses) or as lists `[a,b]` (delimited using square brackets), respectively, with no syntactic difference. This reflects the already existing syntax in iterable unpacking, where tuples and lists are also treated as equivalent syntactic constructs.

[1] https://elixir-lang.org/

4.5 Mappings

Mappings identify each individual element through a unique key, i.e. a hashable object such as a string, number or a tuple. Our design requires that the keys k_i in a mapping pattern be either literals or named constants. The pattern $\{k_1 : P_1, k_2 : P_2, \ldots, k_n : P_n\}$ succeeds in matching a subject s if the subject is a mapping structure such as a dictionary that has an object v_i associated with each key k_i in the pattern, and if each such object v_i matches the respective pattern P_i.

Mapping patterns induce no constraint on the number of elements contained in a mapping, or their ordering. While they work on the same interface protocol as sequences, their semantics is similar to that of constructors and attributes.

4.6 Alternatives

A pattern P can comprise several alternative subpatterns $P_i, 0 \leq i < n$, where P matches a subject s if there is (at least) one subpattern P_k that matches s. The pattern P is written as $P_0|P_1|\ldots|P_{n-1}$. The patterns P_i are tried from left to right until one succeeds in matching s. That is, P matches s if there is a k such that P_k matches s and no pattern P_j with $j < k$ matches s. Patterns P_j with $j > k$ are not tried or taken into consideration after a matching pattern P_k has been found.

Each alternative P_j in a pattern P must cover the same set of variables: $\forall i, j : Var(P_i) = Var(P_j)$. This rule is less restrictive than in, e.g., *Scala* [18], where the alternatives P_j cannot bind any variables, whereas *Thorn* [3] defines $Var(P_i|P_j) = Var(P_i) \cap Var(P_j)$. In contrast to Scala, variables are untyped in Python and a variable that is bound by both P_i and P_j does not need to be reduced to a common unifying type. Binding a variable v in a subpattern P_j but not in P as in Thorn, on the other hand, does not make sense because the subpattern P_j cannot access v and hence the binding to v would be lost (cf. Section 3.3).

4.7 Constructors

When pattern matching was originally introduced in *Hope* [5], constructors were introduced as 'uninterpreted' functions that 'just construct'. In the context of algebraic data types, we can think of constructors both as markers that specify the exact shape and structure of the object, as well as functions that create new objects containing exactly the objects passed to the constructor. This gave raise to what was later called *punning* [3]: the reuse of the same syntax to both mean the actual creation of new objects, as well as the de-construction of an object in pattern matching.

The introduction of fully object-oriented languages has complicated this data model. Constructors no longer reflect the internal structure of an object and are allowed to perform any kind of computation. This is particularly evidenced by overloaded constructors as found in, e.g., *Java*, *Scala*, and many other object-oriented languages. Moreover, constructors in such languages are often prefixed by the keyword `new`, further breaking the symmetry between the construction of new objects and a possible deconstruction in the context of pattern matching. Principles of data abstraction and encapsulation also rule out direct access to the internal structure of an object.

In order to address these issues in the context of pattern matching, new mechanisms were introduced, such as, e.g., views, extractors, and active patterns [6, 9, 12, 22, 27]. Constructors as functions to create objects are thereby decoupled from de-constructors in pattern matching that specify a structural constraint and extract data from an object. In general, a *de-constructor* is a function that accepts a single object and either fails or returns a structured representation of (part of) the data encapsulated in the object.

The strict separation of de-constructors from objects and their constructors allows programs to seamlessly work with different representations of data. For instance, Syme at al. [22] present an example where complex values can either be de-constructed in rectangular or polar form: $c = a+bi = re^{i\varphi}$ (Program 8). Nonetheless, in reality, constructors and de-constructors are often strongly related and reflect each other.

Classes and attributes. Objects in Python have no structure other than mapping attribute names to other objects: objects thus form the vertices in a directed graph with attributes (fields) as edges (cf. Section 6.2). The attribute `__class__` refers to an object's *class* (also known as the object's *type*). Attributes can be modified, added, or deleted. The class of an object does therefore not necessarily guarantee the presence or absence of certain attributes, nor does it impose an order among the attributes.

Despite the absence of guarantees, the class of an object is usually still a strong indicator concerning the 'external' structure, i.e. the set of attributes. The function `isinstance()`, which is used to check if an object is an (indirect) instance of a given class (cf. Section 6.2), is one of the most often used functions. Moreover, in the case of built-in types like numbers or strings, the class does indeed specify the actual structure of the object.

In the tradition of actual constructor patterns, our implementation supports patterns of the form $C()$ that match a subject s if s is an instance of C. As every object is an instance of the base object, `object()` will match everything. Note that the parentheses are mandatory to mark C as a class 'constructor' and distinguish it from variables.

The main objective of pattern matching, however, is to impose structural constraints on the subject and extract relevant information. This implies that a selection of the match subject's attributes need to match corresponding subpatterns. The constructor pattern $C()$ can therefore include a sequence of attributes and patterns: $C(a_1 = P_1, a_2 = P_2, \ldots)$. A subject matches this constructor pattern if the subject matches $C()$

Program 7 The constructor pattern $C()$ allows programs to impose additional patterns on selected attributes, including the possibility to bind an attribute's value to a variable as in `left=lt`, which binds the value of the attribute `left` to the local variable `lt`. Note that the subject `node` might have additional attributes, which are ignored.

```
match node:
  case BinOp(left=Num(lt), op='+', right=Num(rt)):
     return Num(lt + rt)
  case BinOp(left=lt, op=('+'|'-'), right=Num(0)):
     return lt
  case BinOp(left=Num(0), op=('*' | '/')):
     return Num(0)
  case UnaryOp(op='-', operand=Num(x)):
     return Num(-x)
```

and each of its attributes a_k matches the associated pattern P_k (*Newspeak* [7] follows the same approach to express the structure of objects without referring to classes or types). Program 7 presents an example of the constructor pattern with additional patterns imposed on the attributes.

There is no means to express the absence of an attribute or to give an exhaustive list so that the pattern would reject objects with other attributes. Such a scheme would likely prohibit that an instance of a subclass of the intended class would match the pattern [3]. Moreover, Python objects tend to have a large number of attributes, including methods and 'private' fields, which would make such an approach impractical.

Positional arguments. Program 7 already indicates that explicitly naming the arguments can incur an unnecessary overhead with objects that have a relatively clear and simple structure. For instance, we wrote `Num(0)` rather than the more verbose `Num(n=0)`. This requires that there be a mapping from positional arguments in the constructor pattern to the subject's attributes.

The canonical approach to solve the problem would extract the respective attribute names from the class' initializer `__init__` (which corresponds to the constructor in other languages and declares the parameters to be passed to the constructor). Unfortunately, such an approach is not practical in Python. Instead of strictly separated overloaded constructors as in other languages, Python's initializers achieve the same result through a versatile and flexible signature, including various parameters to govern how any data passed in should be read or consumed.

Hence, rather than fall back on the initializer's parameter list, our design uses a special attribute `__match_args__` to provide a sequence (tuple) of field names. It specifies for each position in the constructor pattern the respective attribute. Python's AST nodes already implement such a special attribute as `_fields`. For instance, the class `BinOp` as used in Program 7 specifies:

```
_fields = ('left', 'op', 'right')
```

This mapping actually allows the system to match subpatterns given by position to their respective attributes and we can simply write:

```
BinOp(Num(lt), '+', Num(rt))
```

as shown in Program 2, where the attribute names are inferred.

Syntax. The choice of syntax for constructor patterns aims to mirror the syntax used when actually invoking a constructor (also known as *punning* [3]). The positional arguments as well as the patterns imposed on named attributes resemble the positional and named keyword arguments when invoking a function. In case of the variable pattern, this leads to, e.g., `op=x`, which effectively assigns the value of the attribute `op` (on the left) to the local variable `x` (on the right). The 'inverse flow' of control and information inherent in these patterns thus becomes clearly visible and must be minded by the programmer (the idea that pattern matching is a program running 'backwards' has been noted several times and informed the design of some pattern matching implementations, cf., e.g., [9, 12]).

Alternative approaches with the same functionality but different syntax are possible. In *JavaScript*, for instance, there is a close relationship between dictionaries and objects, leading to a syntax for object de-construction based on dictionary syntax, i.e.[2]:

```
case { 'type': BinOp, 'left': Num(lt),
       'op': '+', 'right': Num(rt) }:
    return Num(lt + rt)
```

In Python, however, there is a clear distinction between objects and dictionaries and this syntax would feel alien to describe an object (although it is used for mappings).

5 Active Patterns

A constructor is essentially a function that returns an object of a specific type, ideally representing the data passed in through the arguments. When used in pattern matching, a constructor stands for the reverse operation, accepting an instance of a specific class and returning a number of objects representing individual aspects or fields of the instance passed in. As constructors are free to perform any computation and either incorporate or dismiss data passed in, they are generally not reversible and it is not possible to reconstruct the original arguments passed in. Case in point: the `str()` constructor takes any Python object and returns a string representation thereof. Since information on the type, class, or structure of the original object is lost during this projection, the operation is irreversible.

The constructor patterns as introduced in our design address this issue by replacing arguments to the actual constructor by the notion of fields and attributes accessible on

[2] cf. https://github.com/tc39/proposal-pattern-matching

the instance. This also follows the intention of the original inception of constructor patterns as a means to avoid explicit field access [5]. An alternative is to introduce explicit de-constructors as functions (or methods) to provide a manual inverse of the constructor [6, 9, 12, 16, 22].

Taking this one step further, we can completely decouple the de-constructor from the class itself, arriving at *views* (Miranda) [27], *active patterns* (F#) [22], or *extractors* (Scala) [6, 18]. This allows programs to offer different ways of extracting data from a given object, or to (re-)structure data where appropriate. Program 8 provides an example where complex numbers are either represented in rectangular or polar form, depending on the operation to perform.

An active pattern or extractor is a function that accepts a single subject as an argument and returns a sequence or mapping of values for further pattern matching, or a special value indicating that the provided subject does not match the structure required for the specific extraction of data. The solutions implemented in both *F#* and *Scala* return an optional tuple, which can be readily adapted to Python, i.e. a (possibly empty) tuple signals a successful match whereas the value `None` signals a failed match (Program 8).

Our design supports extractors as special methods of a class. If a class C provides a method `__match__` then the constructor pattern $C(P_1, P_2, \ldots)$ for a subject s is compiled to `t = C.__match__(s)`. If the returned value t is a tuple, the match succeeds and t is matched against $(P_1, P_2, \ldots)$ as the new subject.

Returning a tuple from extractors is slightly at odds with the idea of specifying attributes as constructor arguments in pattern matching. This could be addressed by creating and returning a full 'proxy' object to replace the original subject. However, it turned out that all our examples and use cases worked with positional arguments only and did thus not require the extra structure of a full proxy object. A combination where positional Patterns match the subjects returned by `__match__` and keyword Patterns match attributes on the original subject, for instance, could still be implemented.

Requiring that the extractor function is a class method `__match__` rather than an independent function has the advantage that the extractor can share a name with the constructor and it allows an API to fully customize the deconstruction of instances, possibly hiding or adding attributes to a specific object where necessary.

Parametrization. Constructors frequently take additional parameters that do not provide data to be stored in the new object, but rather specify how the data passed in should be interpreted. For instance, a string constructor taking in a sequence of bytes usually also takes a parameter specifying the coding, such as ASCII or UTF-8. Such information is not stored in the string and the reverse operation of encoding a string as byte data requires the coding to be specified as a parameter again.

The notion of parametrization applies equally to de-constructors in the context of pattern matching. In contrast to *Scala* [6], we understand parametrization here not as applied to types, but rather as a means to pass additional information to the extractor in the form of regular Python objects, in line with parametrized active patterns in *F#* [22].

Program 10 presents a example, where pattern matching is used to process the results of textual matches by regular

Program 8 An example adopted from Syme et al. [22]: a complex number c can either be deconstructed into rectangular or polar form using 'active patterns'.

```
class Rect:
   @classmethod
   def apply(cls, x, y):
      return complex(x, y)

   @staticmethod
   def __match__(subject):
      if isinstance(subject, complex):
         return (subject.real, subject.imag)

class Polar:
   @classmethod
   def apply(cls, r, phi):
      return complex(r*cos(phi), r*sin(phi))

   @staticmethod
   def __match__(subject):
      if isinstance(subject, complex):
         return (abs(subject), polar(subject))

def calc(op, a, b):
   match (op, a, b):
      case ('+', Rect(x1, y1), Rect(x2, y2)):
         return Rect.apply(x1 + x2, y1 + y2)
      case ('*', Polar(r1, phi1), Polar(r2, phi2)):
         return Polar.apply(r1 * r2, phi1 + phi2)
```

Program 9 In this example, the active pattern `TextDecode` is parametrized on the binary encoding of the text.

```
class TextDecode:
   @staticmethod
   def __match__(subject, enc):
      try:
         return (subject.decode(enc),)
      except UnicodeDecodeError:
         return None

match binary_data:
   case TextDecode["utf-8"](text):
      print(text)
   case TextDecode["latin-1"](text):
      print(text)
```

Program 10 Using regular expressions requires actively 'polling' the match for information such as group name. Parametrization allows programs to express this by passing additional query-information on to the extracting function.

```
class RegExGroup:
   @staticmethod
   def __match__(subject, group_name):
      if isinstance(subject, re.Match) and \
                           re.group(group_name):
         return (subject.group(group_name),
                 subject.start(group_name),
                 subject.end(group_name))

TOKEN_RE = re.compile(
  r"(?P<digits>\d+)|(?P<letters>[a-zA-Z]+)"
)
match TOKEN_RE.match(input):
  case RegExGroup["letters"](value, start, end):
     print(f"found a sequence of letters: {value}")
  case RegExGroup["digits"](value, start, end):
     print(f"found a sequence of digits: {value}")
```

expression. When a match is found by the regular expression engine, information such as the name of the group must be actively queried. Parametrization allows programs to pass this 'active' name to the `__match__` extractor as additional parameter.

Parameters to a constructor clearly have different semantics than patterns. A pattern is essentially a function that maps a subject to an optional substitution. A parameter, on the other hand, is a passive value with no 'knowledge' of the match subject. In particular names have different semantics: when used as a pattern a name is a pattern variable that matches and binds any subject. As a parameter a name refers to an existing variable and loads its current value.

It is therefore essential to syntactically separate parameters from patterns. While we choose square brackets, as shown in Program 10, other variants are equally possible: for instance, F# writes parametrized patterns as mappings, e.g. `RegExGroup "letters"-> (value, start, end)`.

6 Background on Python

Various design decisions are due to the nature of Python, its syntax and semantics. This section provides a brief overview of elements that were relevant for our design decisions.

6.1 Scope

The scope of variables is governed by *module* and *frame* objects. Module objects typically persist through the entire runtime of a program and act as the global namespace, i.e. as a container for global variables (there are usually several coexisting modules in a program). Frame objects, on the other hand, hold temporary runtime data for code objects, such as local variables, the instruction pointer, the operand stack, as well as a pointer to the caller frame. Frame objects thus form the call stack as a simply linked list. Each invocation of a function creates a new frame object that gets destroyed when the function returns.

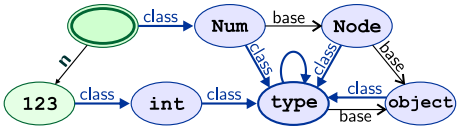

Figure 3. The graph shows how the object `Num(n=123)` (on the upper left) is represented internally (simplified). Classes are fully reified objects with a class of their own. The `Num`-object is an instance of the classes `Num`, `Node`, and `object`.

The granularity of the scope of local variables is at the level of functions and frames. If a name is assigned to anywhere in the body of a function (i.e. the name appears with *store* semantics), it is marked as a local variable by the compiler, together with all formal parameters. Access to variables of a surrounding scope is achieved by creating a closure.

The only way to restrict the scope of a variable to part of a function's body (such as a case clause) would be to actively delete the variable when leaving the block. This would, however, not restore any previous value of a local variable in the function's scope. The introduction of specific scopes for case clauses would therefore incur an overhead with no evident gain.

Name resolution. All name resolution happens during runtime, although the compiler can optimize it in case of local variables. Global names are actually (mutable) attributes of the module object, so that program code external to a module can change a module's global variables. A compiler can therefore not reliably decide for a name occurring in a pattern whether it refers to a globally defined constant or should be interpreted as a local variable.

The interpreter needs to resolve the named constants or class names occurring in constructor patterns for each execution of the match statement. While this presents an obstacle for optimizing compilation of pattern matching, it also naturally supports type parametrization in that the class name can be a parameter passed into the surrounding function.

6.2 Python's Data Model

Python organizes data as a directed labeled graph: the *object graph*. Each vertex is represented by an object with a unique identity. Objects have no internal structure or size that would be exposed to a Python program, although they emulate such structure. Vertices are mostly expressed as attributes or mappings on a given object. Every object refers to another object as its *class* or *type*, written `type(o)` or `o.__class__` in Python (Figure 3).

A sequence $(a_0, a_1, \ldots, a_{\ell-1})$ of length ℓ is represented as an object s equipped with a function g_s (`__getitem__`) that maps integer representatives to the respective elements, i.e. $g_s(n) = a_n$ for each $0 \leq n < \ell$. However, there is no internal structure exposed and an actual implementation of the Python language is free to internally organize sequences as arrays, linked lists, trees, or any other data structure.

Immutable objects do not permit manipulation of any of their outgoing edges: once the object has been created, all outgoing edges remain fixed. However, an immutable object might refer to a mutable object, so that the data value represented by an immutable object might change with time. For instance, while tuples are immutable, the actual 'value' of a tuple that includes a list like $(0, [1, 2])$ might change when the list $[1, 2]$ itself changes.

Classes. Class objects distinguish themselves by additional mandatory attributes such as a tuple of base classes (Python supports multiple inheritance). A class C is a *subclass* of C' if there is a path following 'base class' edges from C to C'. An object o is an (indirect) *instance* of C if $type(o)$ is a subclass of C. However, a class can override the mechanism to check whether an object o is an instance of a class C [26].

6.3 Annotations

Annotations or type hints [25] are primarily a syntactic feature to annotate, e.g., a parameter or function with a specific Python object. An annotated function like:

```
def sqrt(x: float)->float:
```

looks as if fully typed. However, neither the Python compiler nor the executing interpreter check or enforce any types from annotation. Annotations are a tool for documentation and external third-party checkers.

It seems tempting to reuse annotations in patterns as a means to express type constraints. A constraint such as `x := int()` using the constructor patterns could be rewritten as `x: int` instead. Besides an (obviously minor) syntactic simplification, this would not add to the expressive power of patterns, but come at the expense of violating the principle that annotations do not express type constraints. Moreover, more complex type expressions such as `list[int]` could not be implemented through a simple instance or type check, but would have to involve checking each element of the list for its respective type. The potential complexity, often involving de-construction of the subject, is therefore better expressed through actual constructor patterns.

7 Discussion

7.1 Implementation

Match statements can be compiled to a series of nested if/else statements, checking each pattern one by one using standard functions such as `isinstance()`, `getattr()` and `len()`. However, even relative small examples frequently include room for optimization. In Program 2, for instance, the first two case clauses both check whether the subject in question is an instance of `BinOp`. If the check fails in the first case, the second case does not need to be considered at all. Such optimization effectively leads to a *decision tree* that selects the correct case clause [2, 6, 22].

While optimized compilation is common in statically typed languages, it is much more difficult to achieve similar results in dynamic languages, which favor an approach where patterns are first-class objects [3, 11], which naturally precludes optimization. In contrast to *Grace* and *Thorn*, however, we pursue a more compiler-oriented approach where patterns are not reified as first-class objects within the language.

The primary difficulty concerning Python is the possibility to customize the behavior of built-in functions like `isinstance()` above for specific classes, which is even more pronounced with active patterns. In principle, such functions could cause side effects and thus observe how often they were invoked during a pattern matching process. Similarly, attribute or item access might yield values generated dynamically.

In order to leave room for compilers to optimize match statement and generate code for a decision tree rather than force sequential testing of each case clause, we specifically state that the interpreter is allowed to cache and reuse any values obtained during the matching process, or even to reorder evaluation of patterns. However, guards must be fully evaluated without caching and in the correct order. In other words: any compiler in Python can act under the assumption that the patterns in a match statement are static. A more precise specification as in [19] is problematic due to, e.g., aliasing and parametrization.

A full discussion of compilation and evaluation of pattern matching in Python with optimization strategies is beyond the scope of this paper and will be tackled by future work.

7.2 Python Enhancement Proposal

The authors of this paper submitted a proposal to add structural pattern matching to Python [4], evoking a wealth of comments and discussions in the wider Python community. As a direct outcome of our engagement with the community, we decided to focus the initial proposal on core elements with the possibility to extend it to the full design as presented here later on. In particular, active patterns were dropped from our Python Enhancement Proposal [4].

One of the main issues turned out to be the obvious similarity with 'switch' statements as found in, e.g., *C*. It was often felt that the use of a simple name as a pattern should be a *named constant* pattern rather than a *variable* pattern. Others noted that the underscore _ is a valid name in Python and should therefore be treated no differently than other names, hence as a *variable* pattern with binding.

Despite structural pattern matching finding its way into ever more 'mainstream' programming languages, there is

still considerable reservation concerning the perceived complexity and usefulness of pattern matching. This is in line with reports from Odersky when introducing pattern matching to Scala [17].

8 Conclusion

Pattern matching originated in statically typed functional languages, but has since been adopted to object-oriented and even dynamic languages. While object-oriented languages tend to rely on *views* or *active patterns* to map objects to matchable representations, dynamic languages favor an approach where patterns are first-class objects.

The approach presented in this paper introduces pattern matching as a statically compiled feature to the dynamic language Python. It builds on already existing language features like iterable unpacking. Differences arise mostly due to the conditional nature of patterns. The structure of objects can be expressed through both classes or explicit attribute enumeration. Active patterns and parametrization offer extensive customizability akin to first-class patterns, while staying within the framework of statically compiled patterns.

The authors of this paper presented the core features (Section 4) to the Python community for inclusion in Python. The corresponding *Python Enhancement Proposal* (PEP) [4] has sparked widespread interest from the community: while pattern matching is generally welcomed, various details of concrete syntax are still being discussed.

A proof-of-concept implementation is available as part of the PEP. However, the emphasis on static patterns in our design should allow future work to support the compilation of match statements to efficient decision trees.

Acknowledgments

This work was partially supported by the Engineering and Physical Sciences Research Council (EPSRC), through grant eferences EP/K026399/1 and EP/P020011/1.

References

[1] C. Angelico, T. Peters, and G. van Rossum. PEP 572 – Assignment expressions. https://www.python.org/dev/peps/pep-0572/, 2018.

[2] L. Augustsson. Compiling pattern matching. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.

[3] B. Bloom and M. J. Hirzel. Robust scripting via patterns. *SIGPLAN Not.*, 48(2):29–40, Oct. 2012.

[4] B. Bucher, D. F. Moisset, T. Kohn, I. Levkivskyi, G. van Rossum, and Talin. PEP 622 – Structural pattern matching. https://www.python.org/dev/peps/pep-0622/, 2020.

[5] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*, LFP âĂŹ80, pages 136–143, New York, NY, USA, 1980. Association for Computing Machinery.

[6] B. Emir, M. Odersky, and J. Williams. Matching objects with patterns. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, pages 273–298, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[7] F. Geller, R. Hirschfeld, and G. Bracha. *Pattern Matching for an object-oriented and dynamically typed programming language*. Number 36 in Technical Report. Universitätsverlag Potsdam, 2010.

[8] 'Haskell-cafe' mailing list. Conflicting variable definitions in pattern. https://www.mail-archive.com/haskell-cafe@haskell.org/msg59617.html, 2009.

[9] M. Hirzel, N. Nystrom, B. Bloom, and J. Vitek. Matchete: Paths through the pattern matching jungle. In P. Hudak and D. S. Warren, editors, *Practical Aspects of Declarative Languages*, pages 150–166, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[10] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, Jan. 1982.

[11] M. Homer, J. Noble, K. B. Bruce, A. P. Black, and D. J. Pearce. Patterns as objects in Grace. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, page 17âĂŞ28, New York, NY, USA, 2012. Association for Computing Machinery.

[12] C. Isradisaikul and A. C. Myers. Reconciling exhaustive pattern matching with objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 343–354, New York, NY, USA, 2013. Association for Computing Machinery.

[13] M. Krebber and H. Barthels. MatchPy: Pattern Matching in Python. *Journal of Open Source Software*, 3(26):2, June 2018.

[14] M. Krebber, H. Barthels, and P. Bientinesi. Efficient pattern matching in Python. In *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*, PyHPC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[15] Ł. Langa. PEP 443 – Single-dispatch generic functions. https://www.python.org/dev/peps/pep-0443/, 2013.

[16] J. Liu and A. C. Myers. JMatch: Iterable abstract pattern matching for Java. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages*, pages 110–127, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[17] M. Odersky. In defense of pattern matching. https://www.artima.com/weblogs/viewpost.jsp?thread=166742, 2006.

[18] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: Updated for Scala 2.12*. Artima Incorporation, 2016.

[19] C. Okasaki. Views for standard ml. In *SIGPLAN Workshop on ML*, pages 14–23. Citeseer, 1998.

[20] K. Scott and N. Ramsey. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA*, 2000.

[21] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, pages 446–464, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[22] D. Syme, G. Neverov, and J. Margetson. Extensible pattern matching via a lightweight language extension. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 29–40, New York, NY, USA, 2007. Association for Computing Machinery.

[23] D. A. Turner. SALS language manual, 1976.

[24] G. van Rossum. All things Pythonic: Five-minute multimethods in Python. https://www.artima.com/weblogs/viewpost.jsp?thread=101605, 2005.

[25] G. van Rossum, J. Lehtosalo, and Ł. Langa. PEP 484 – Type hints. https://www.python.org/dev/peps/pep-0484/, 2014.

[26] G. van Rossum and Talin. PEP 3119 – Introducing abstract base classes. https://www.python.org/dev/peps/pep-3119/, 2007.

[27] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. Association for Computing Machinery.