Complex Python Features in the Wild

Yi Yang
yangy25@rpi.edu
Rensselaer Polytechnic Institute
USA

Ana Milanova
milanova@cs.rpi.edu
Rensselaer Polytechnic Institute
USA

Martin Hirzel
hirzel@us.ibm.com
IBM Research
USA

ABSTRACT

While Python is increasingly popular, program analysis tooling for Python is lagging. This is due, in part, to complex features of the Python language—features with difficult to understand and model semantics. Besides the "usual suspects", reflection and dynamic execution, complex Python features include context managers, decorators, and generators, among others. This paper explores how often and in what ways developers use certain complex features. We analyze over 3 million Python files mined from GitHub to address three research questions: (i) How often do developers use certain complex Python features? (ii) In what ways do developers use these features? (iii) Does use of complex features increase or decrease over time? Our findings show that usage of dynamic features that pose a threat to static analysis is infrequent. On the other hand, usage of context managers and decorators is surprisingly widespread. Our actionable result is a list of Python features that any "minimal syntax" ought to handle in order to capture developers' use of the Python language. We hope that understanding the usage of Python features will help tool-builders improve Python tools, which can in turn lead to more correct, secure, and performant Python code.

CCS CONCEPTS

• **Software and its Engineering → Language features**.

KEYWORDS

Python, AST

ACM Reference Format:
Yi Yang, Ana Milanova, and Martin Hirzel. 2022. Complex Python Features in the Wild. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524842.3528467

1 INTRODUCTION

Dynamic languages such as Python are increasingly popular. Python in particular is widely used in data science and machine learning[1]. Unfortunately, *static analysis* tooling for Python is not widely developed or used, while such tooling could undoubtedly benefit Python development. Most research prototypes (e.g., [10, 16, 22, 26, 28]) as well as GitHub projects (e.g., [1, 2, 4, 5, 11]) we are aware of that target static analysis for Python are ad-hoc explorations of AST constructs; they redefine classical analyses and provide *no correctness guarantees* as they are forced to ignore many Python features.[2]

The principal problem, we conjecture, is that Python is rich in what we dub *complex features*. These include classical *dynamic features* such as reflection and dynamic code execution with eval, as well as features such as *context managers* [3], *decorators*[3], and *generator expressions*[4]. The semantics of these features is either inexpressible statically, or it is poorly understood, or it requires careful consideration as it significantly complicates static analysis.

Typical *flow-sensitive* static analysis works on a 3-address-code control-flow graph (CFG) intermediate representation (IR) of the program source code. Most programming languages (e.g., C, C++ and Java) have an established translation from the higher-level AST representation of the program to a 3-address-code CFG IR, carried out by widely-used frameworks, most notably LLVM[5], WALA[6], Soot[7], and DOOP[8]. Following the translation, one can define a wide variety of analyses, such as classical dataflow analysis, pointer analysis, and call graph construction on this representation. As another point, typical *flow-insensitive* analysis ignores control-flow and translates the AST into a sequence of 3-address statements:

```
x = new A # object creation
x = y # assignment
x.f = y # update
x = y.f # field read
x = y(z) # y evaluates to a function value
```

Unfortunately, there are few tools that translate the Python AST constructs into either the flow-sensitive CFG IR or the flow-insensitive sequence of statement. WALA includes a Python front-end [8], and Scalpel aims to provide a suite of Python analysis facilities [14]. Aside from these, at this point, Python cannot take advantage of analysis and algorithms that have been developed throughout decades of research. Even something as basic as call graph construction analysis for Python remains an open problem — the first call graph analysis was published in 2021 [26] and although it handles a larger subset of features compared to previous attempts, it is still an ad-hoc analysis over the AST and leaves out soundness reasoning and the handling of flow of values. Surprisingly, pointer

[1]see https://www.aitrends.com/data-science/here-are-the-top-5-languages-for-machine-learning-data-science/

[2]Fromherz et al [10] define a subset of the Python syntax and carry out sound static analysis over this subset; our results show the syntax misses a significant portion of developer code.
[3]https://www.python.org/dev/peps/pep-0318/
[4]https://www.python.org/dev/peps/pep-0289/
[5]https://llvm.org/
[6]https://github.com/wala/WALA
[7]http://soot-oss.github.io/soot/
[8]https://bitbucket.org/yanniss/doop/src/master/

analysis, which has been studied for decades in Java and C and can benefit Python applications, is an open problem in Python.

To illustrate the daunting task a program analysis faces, consider one complex feature, the with statement [3]:

```
with expr [as var]: stmt_seq
```

The idea of the with statement is that expression expr evaluates to a *context manager* object and the context manager object is responsible for the handling of exceptions and resources related to the execution of stmt_seq. The typical example is the handling of files and streams:

```
with open(...) as file:
    for line in file ...
```

The open(...) expression opens the file but it also creates a context manager object, which wraps around the body of the with statement, handles exceptions, and releases resources (e.g., closes a file).

The one-line with construct is syntactic sugar for the following non-trivial sequence (see the Python documentation):

```
manager = (expr) # evaluation of expr returns context manager
enter = type(manager).__enter__
exit = type(manager).__exit__
value = enter(manager) # evaluates to an object, e.g., the file object
hit_except = False
try:
    var = value
    stmt_seq
except:
    hit_except = True
    if not exit(manager, *sys.exc_info()):
        raise
finally:
    if not hit_except:
        exit(manager, None, None, None)
```

Evaluation of expr returns a *context manager* object whose class implements the special __enter__ and __exit__ methods. Clearly, it is non-trivial to translate this code into a 3-address code CFG IR (we are not aware of an analysis that does); translation requires handling of try-except-finally semantics, as well as context manager semantics. A context manager can be built-in (e.g., open), defined in a standard library such as contextlib, defined in a third-party library, or user-defined. This complicates even the more straightforward flow-insensitive analysis.

The complexity of Python motivates the study in this paper. We explore Python code in the wild to find out whether, and in what ways, developers actually use complex features. This empirical exploration helps determine whether one could define a "Featherweight Python" syntax, a subset of AST constructs that covers a reasonably large portion of developer code, while at the same mapping into known CFG IR (or flow-insensitive IR) and allowing for reuse of existing sound static analysis technology.

Concretely, we mine feature usage over two datasets of public Python repositories collected from GitHub. One dataset reflects a snapshot from March 2019 and the other dataset reflects a snapshot from November 2021. Each dataset includes millions of files and hundreds of thousands of GitHub repositories.

We define the following categories of complex features and mine the two datasets for occurrence of these features.

(1) dynamic features (e.g., getattr, eval);
(2) functional features (e.g., list comprehensions, yield statements);
(3) decorators (e.g., @property);
(4) context managers (i.e., the with statement); and
(5) asynchronous execution (e.g., AsyncFunctionDef).

The dynamic features require minor analysis of the Call AST construct; the rest of the features correspond to syntactic AST constructs. For example, list comprehensions are parsed into the ListComp AST construct. Sect. 2 details the exact sets of features we mine and the mining methodology. We present aggregate usage numbers, histograms that break down usage within each category, results on how these categories overlap, as well as qualitative analysis of code. We address the following research questions.

- RQ1. How often do developers use complex features?
- RQ2. In what ways do developers use complex features?
- RQ3. Does use of complex features increase or decrease over time?

Our findings can be viewed as both positive and negative. For example, usage of dynamic features that pose a threat to static analysis is infrequent (e.g., eval and setattr are rarely used, and getattr is most commonly used with a constant string argument). On the other hand, usage of context managers and decorators is surprisingly widespread. Our actionable result is a list of Python features that any "Featherweight Python" ought to handle in order to capture developers' use of the Python language. Subsequently, static analysis ought to define a translation semantics for them and handle them in an efficient and (desirably) sound way. Our final observation is that usage of complex feature has not changed significantly over time.

In addition to the implications for static analysis, we hope this paper will also be interesting for the broader Python community. Understanding feature usage can help with documentation and education. It can inspire new features and tooling beyond static analysis. And, even more broadly, understanding adoption of Python features could inform design decisions for other programming languages.

2 METHODOLOGY

Sect. 2.1 describes our datasets, Sect. 2.2 presents the mining analysis and details the features that we mine, and Sect. 2.3 briefly discusses manual code examination.

2.1 Datasets

Our goal is to study Python features as they occur in Python projects in the wild. This requires us to obtain as many Python projects as possible and the projects should not be restricted or filtered, eliminating bias towards projects of scale or of any particular purpose. The projects that we obtain are open-source repositories from GitHub. Some of them implement stand-alone programs, others reusable libraries. We used a query to mine GitHub public repositories[9]. We ran this query twice, obtaining two datasets of Python projects: one dataset reflects a capture from March 20, 2019, and the other dataset reflects a capture from November 11, 2021.

[9]https://github.com/wala/graph4code/blob/master/extraction_queries/bigquery.sql

Due to the tools we use, the Python AST library specifically[10], projects with syntax errors are not included in the final counts. As a reference point, out of 3,844,561 files in the 2021 dataset, 750,392 files are excluded due to the Python AST library issuing syntax errors. The 2019 dataset contains 70,826 GitHub repositories from 49,456 organizations; we analyzed 1,207,916 files. The 2021 dataset contains 51,493 GitHub repositories from 30,182 GitHub organizations; we analyzed 3,094,169 files.

2.2 Mining Complex Features

Upon acquiring the data, all files are put through a mining tool we developed[11]. The Python AST library parses the source code and creates the standard syntax tree structure. Syntactic constructs have their nodes in the syntax tree; for instance, a function definition is parsed into the *FunctionDef* AST construct/syntax tree node. We analyze the syntax tree and record occurrences of AST constructs. All records of occurrences include the name of the file, type of the construct, line number, and origin. For functional features, it is sufficient to record the type of the AST construct (e.g., a list comprehension corresponds to the *ListComp* AST construct); for dynamic features we analyze function calls and record when the function name matches the target function names (e.g., a call getattr(...) is parsed into the *Call* AST construct; with its *func* component being the *Name* 'gettattr').

We mine the following **dynamic features**: *getattr* (returns the value of the attribute of an object), *setattr* (sets an attribute of an object), *delattr* (deletes the attribution), *hasattr* (returns true if the object has the attribute), and *eval* and *exec* (dynamic code execution). A dynamic feature is recorded when the tool encounters a *Call* construct whose function name matches one of those names.

We mine the following **functional features**: *lambda* (the standard anonymous function definition), *set comprehension*, *dictionary comprehension*, and *list comprehension* (comprehensions take an element expression that defines how the set, dictionary, or list is filled in), and *yield*, *yieldFrom*, and *generator expression* (similarly to comprehensions generators take an element expression, however, the expression is evaluated when retrieved from the generator structure). They all correspond to AST constructs, i.e., nodes, and they are recorded accordingly. For example, in the code snippet:

```
a = [1,2,3]
b = [x**2 for x in a]
```

the right-hand-side of the second line is parsed as the *ListComp* AST construct and it is recorded in the analysis as the functional feature *list comprehension*.

Our motivation to classify and study comprehensions and generators as functional features stems from their importance in Haskell. In Haskell list comprehensions are syntactic sugar for building computations over the list monad[12].

Decorators includes *any decorators*. We record all decorators that are parsed into a *FunctionDef* AST construct, *AsyncFunctionDef* AST construct, or *ClassDef* AST construct. Each decorator's name is recorded. For example:

```
@classmethod
def from_json(cls, json):
    return _RANGE_ITERATORS[json["name"]].from_json(json)
```

The @classmethod decorator expression is parsed in the *FunctionDef.decorator_list* AST construct and recorded in our study as a decorator of name "classmethod".

Decorators are a complex feature because they alter control flow of the program. In the above example classmethod is a built-in function and the above syntax is roughly equivalent to from_json = classmethod(from_json) which transforms the from_json instance method into a class method; it is a burden on the programmer to ensure that the classmethod is called on class objects and it accesses only class fields and no instance fields. Furthermore, decorator code can be built-in, part of a standard library, third-party library or user code, which presents challenges for static analysis.

The **With** AST construct leads to the invocation of a *context manager*, which facilitates exception handling and bookkeeping, as discussed in the introduction. The canonical example is files:

```
with open("file.txt") as a:
  print(a.content())
```

There is an AST construct *with*. Our analysis records the occurrences of *with* constructs as well as the expressions that are enclosed in *with*. In the above example, we record a call to built-in function *open*.

Async is a newer feature of Python, introduced in Python 3.7, which was released in 2018. We record an async feature if there is an *AsyncFuncionDef*, *AsyncFor*, or *AsyncWith* nodes in a given Python program's syntax tree.

2.3 Manual Source Code Examination

To better understand the usage of certain features by developers we manually examine the source code. The source code for examination is randomly selected (30 to 50 files for a target feature).

The manual examination classifies and estimates the general purpose for a feature, which is then used to invalidate or support an assumption made about it. We use these rules when carrying out manual code examination:

(1) The classification or estimation of a feature should be determined from a single file's context, otherwise it is not included in the manual source code analysis.
(2) Non-Python text can be used as evidence to the high-level purpose of a segment of code or a program, including documentation, comments, and file names.
(3) Python's variable, class, and statement names in the file can be used for usage classification or estimation.

3 RESULTS

We address RQ1 (Sect. 3.1), RQ2 (Sect. 3.2), and RQ3 (Sect. 3.3).

3.1 RQ1: How Often Do Developers Use Complex Features?

Tab. 1 and Tab. 2 show the frequency of occurrence of complex features across our datasets. For all complex feature categories except for *Async*, occurrences are found in a large percentage of

[10] see https://docs.python.org/3/library/ast.html
[11] All scripts can be found here: https://github.com/2042Third/ast_analysis_python
[12] see https://wiki.haskell.org/List_comprehension and https://wiki.haskell.org/All_About_Monads#Exercise_3:_Using_the_List_monad

Feature	Frequency(files)	%
FNL (functional)	817,345	26.416
DYN (dynamic)	401,395	12.973
DEC (decorators)	386,134	12.479
WTH (with)	385,613	12.463
ASC (async)	1,831	0.059

Table 1: Per file data from 2021 dataset. E.g., functional features (FNL) are used in 26% of all Python files.

Feature	Frequency(files)	%
FNL (functional)	340,158	28.161
DYN (dynamic)	196,472	16.265
DEC (decorators)	175,828	14.556
WTH (with)	112,481	9.312
ASC (async)	2,353	0.195

Table 2: Per file data from 2019 dataset.

files in both datasets. To shed additional light on prevalence, we took a snapshot of the usage of dynamic, functional, and decorator features per GitHub repository and per GitHub organization. The Venn diagram in Fig. 1 shows the occurrence of these features per repository and the one in Fig. 2 the occurrence per organization. About 50% of repositories use one of these features; interestingly, there is a large overlap among the features (in about 18% of repositories), which shows that developers who use one complex feature are more likely to use others as well. We elaborate on the overlap and the differences between the 2019 and 2021 results in Sect. 3.3. Fig. 2 shows that nearly all organizations use one of the features.

3.2 RQ2: In What Ways Do Developers Use Complex Features?

To address the second research question, we break down the usage within each broad category. We also examine code manually to better understand usage, particularly in cases when the results surprise us.

3.2.1 Dynamic Features. In the general case, *dynamic* features present an unsurmountable burden for static analysis. For example, *eval*, where the argument expression is constructed, parsed and interpreted at runtime, cannot be resolved statically. However, dynamic data is often available before program execution rendering the use of a dynamic feature unnecessary. Richards et al. study usage of *eval* in JavaScript and identify a significant percentage of unnecessary use [25].

Figure 3 shows the histogram of the dynamic features in the 2021 dataset. One observation is that *eval* and *exec* occur relatively infrequently. The built-in dynamic attribute getters and setters, namely *getattr*, *hasattr*, and *setattr*, dominate in our results and fortunately, they often can be handled soundly by a static analysis.

We examine 30 randomly selected files that contain dynamic features and 30 uses occurring in these files. (If there are multiple

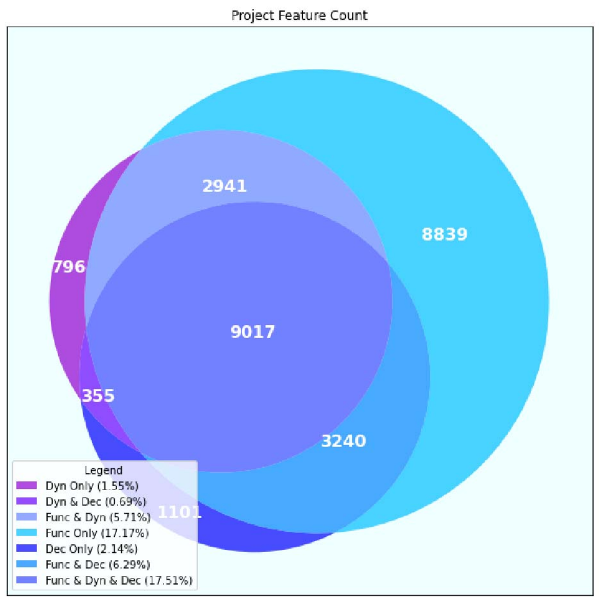


Figure 1: Per-repository features from the 2021 dataset.

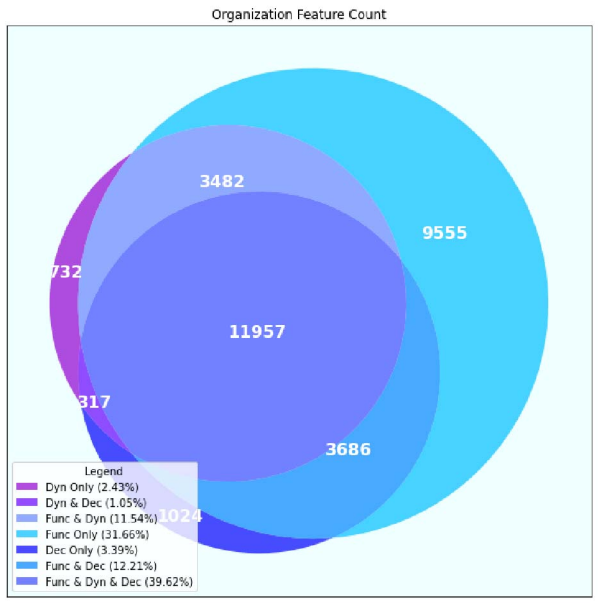


Figure 2: Per-organization features from the 2021 dataset.

uses of dynamic features, we randomly settle on one.) Based on our experience, we believe that in 5 cases dynamic features cannot be handled by static analysis, while in 24 cases they can be handled soundly in the analysis. In the 24 cases where dynamic features can be handled by static analysis, the majority are *getattr* accesses, where the attribute argument is a string constant, i.e., it is known

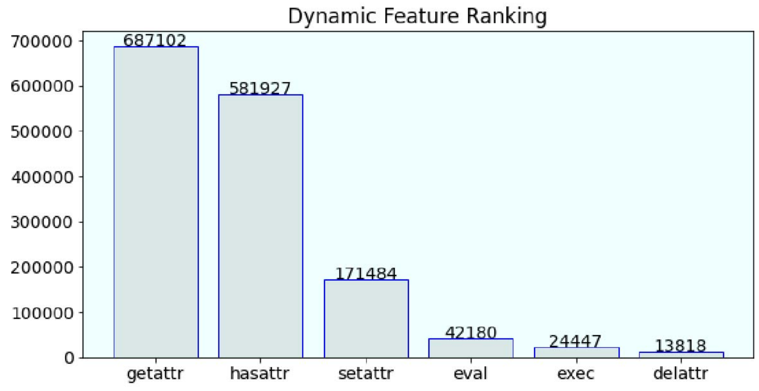


Figure 3: The frequency of overall dynamic features in Python, counting the number of occurrences of each construct across all files in the 2021 dataset.

before program execution what attribute of the target object the program may access at this point. For example,

```
# Cheap way of de−None−ifying things
hosts = hosts or getattr(dj_settings,'ES_HOSTS',DEFAULT)
timeout = (timeout if timeout is not None else
           getattr(dj_settings,'ES_TIMEOUT',DEFAULT))
```

In this example, we may replace the dynamic call with direct attribute access to dj_settings.ES_HOSTS. However, this misses the default value — the semantics of *getattr* is that if the object does not have the attribute argument, it returns DEFAULT. Thus, to completely mimic the behaviors of this example, a flow-sensitive static analysis may handle the case by reducing to a *try*/*except* block, catching the *AttributeError* and assigning the DEFAULT value accordingly. A flow-insensitive analysis may handle the case by reducing it to a sequence of assignment statements. For example, getattr(dj_settings,'ES_HOSTS',DEFAULT) would reduce to

```
tmp = dj_settings.ES_HOSTS
tmp = DEFAULT
```

There are 17 programs that use Python for *regex interpretation*, such as for reading raw HTTP packets and building relevant information inside a Python object and dynamically creating Python objects at runtime. Creating objects at runtime does not appear to be essential in our examination. Most such programs are doing so because the dynamically constructed structures are easily manipulated late in the data's lifetime.

This is an example of use of *eval* in a network setup program from our dataset:

```
def fortios_ips(data, fos):
    ...
    methodlist = ['system_zone']
    for method in methodlist:
        if data[method]:
            resp = eval(method)(data, fos)
            break
    ...
```

Clearly, the method is available before program execution and it is unclear why the programmer used *eval*.

In 5 cases the *dynamic* features we observed cannot be easily handled by static analysis. Consider the example below, which loads a module dynamically, then queries the module object based on a substring that is not known until runtime:

```
...
try:
    ...
    module = __import__(path)
...
for part in path.split('.')[1:]:
        try:
            module = getattr(module, part)
        except AttributeError: return None
    return module
```

We observe that *hasattr*, *getattr*, *setattr*, and *delattr* are sometimes used for extensions and updates. They would be used for inspecting, deleting, creating or modifying objects at runtime. On the other hand, statically typed languages can only read or modify objects' attributes at runtime. Developers may account for future extensions by adding potentially redundant structures/classes in static environments. This may lead to overhead in development and in run-time memory, compared to dynamic methods.

Another observation is that dynamic features such as *hasattr* are used in programs that do object "introspection" and observation; in the 5 programs, 2 are programs that check the class of an object. For example:

```
# list_max_show_all
if hasattr(cls, 'list_max_show_all') ...:
    raise ImproperlyConfigured(...)
```

These programs inspect class objects and return information about the structure of the class of the object. For example, one part of a program is able to directly reverse-engineer the structure of input objects using combinations of dynamic features such as getattr(object, "__name__", None) to reconstruct the source code.

These introspection-type features are difficult to handle by static analysis. In a statically-typed language such as Java, an objects' structure is determined at compile time and it does not change during the objects' lifetime.

3.2.2 Functional Features. Functional features do not inherently impede static analysis. We have selected them for mining because they do complicate the analysis — traditional analysis for imperative object-oriented languages (the analysis that most closely fits Python) does not handle functions as first class values, closures, or infinite lists; it requires extensions to adapt to those features.

Functional features occur frequently, as shown in Tab. 1 and Fig. 4. As discussed earlier, the functional features are computed by counting AST constructs, such as *ListComp*. Not surprisingly, list comprehensions and lambdas have the largest number of occurrences in code by far. On the other hand, we found the high number of occurrences of *Yield* somewhat surprising.

Yield is used to define *generator functions* that allow for Haskell-style infinite lists and lazy generation of elements of the list. For example, in Haskell [1,_] represents the infinite list of positive integers; the programmer can retrieve the elements of this list one by one for as long as they are needed, but if they try to evaluate the entire list (for example, to display on screen) the program descends into infinite recursion. Yield-constructed generators have similar functionality:

```
def ints():
  i = 1
  while True:
    yield i
    i = i +1

infinite = ints() # infinite is a generator object
print(infinite.__next__()) # prints 1
print(infinite.__next__()) # prints 2
... # can print as many integers as needed
```

The above ints function returns a generator object. The first call infinite.__next__() evaluates yield, and passes the value of 1 to the main function. Control returns to the main function and it prints 1. The next call to infinite.__next__() returns control to ints, where the first yield left off; ints increments i, yields a second time and returns control to main, and so on. The client can generate as many integers as needed.

Python provides the *GeneratorExp* construct as well:

```
finite = (i for i in range(1,3)) # finite is a generator object

print(finite.__next__())
print(finite.__next__())
```

In the above example, finite is a generator object; its task is to generate two integers. The above use is similar to list comprehensions, however, a list comprehension is fully evaluated while the elements of the generator are produced one-by-one (e.g., as a result of calls to __next__()). Note that if we try to get a third integer the program terminates with an exception as we have reached the limit.

We were surprised by the high number of occurrences of yield because the semantics of yield are non-trivial. A deeper look into *yield* from the **source code** shines light on how developers use *yield* as they use it almost as frequently as *lambda*. We examine 30 randomly selected files, and for all but 4 inconclusive cases, *yield* appears to be used for Haskell-style *lazy generation* of elements. In Haskell, lazy lists can improve the algorithmic complexity of incremental data structures, such as Okasaki's functional queues and deques [18].

In most cases the callee passes a value to the caller using a *return* statement and the value is fully evaluated. However, there are cases where the callee passes a generator expression and the elements are not fully evaluated. They are evaluated when they are needed. *Lazy evaluation* or more precisely lazy generation of list elements can have two applications. In one case, it may be beneficial to have better run-time performance at generation time but a slowdown at data access time. In another case, the bound on the number of elements that are needed may not be known at generation time.

Consider this example from our 2021 dataset:

```
1  def ibytes2icompressed(source):
2      yield (
3        b'\037\213\010\000' +
4        # Gzip file, deflate, no filename
5        struct.pack('<L', long(time.time())) +
6        # compression start time
7        b'\002\377'
8        # maximum compression,
9      )
10     ...
11     for d in source:
12       ...
13       chunk = compressor.compress(d)
14       if chunk:
15           yield chunk
16     yield compressor.flush()
17     ...
```

Function ibytes2icompressed(source) returns a generator object. The first element the client retrieves with __next__() is the timestamp (generated by the yield in Line 2), the subsequent elements are the compressed chunks (the yield at Line 15), and the last element is the result of flush. Since the argument source can itself be a generator, defining compression as a generator helps retain the incremental nature of the data generation and transformation. A benefit of this style of generator chaining is that intermediate results need not be materialized in their entirety; for instance, the result of compression could be streamed straight to disk or a socket.

Consider another example of usage from our dataset:

```
def gen_invalid_vectors():
    '''Generate invalid test vectors'''
    yield "",
    yield "x",
    while True:
        ...
        for template in templates:
            val = gen_invalid_vector(...)
            ...
            if not is_valid(val):
                yield val,
```

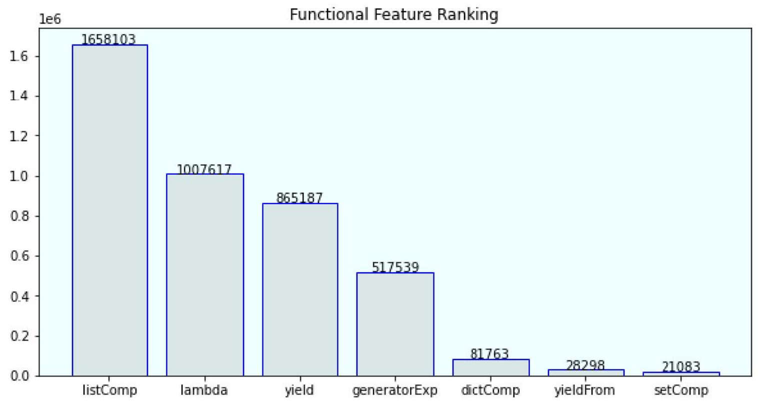


Figure 4: The frequency of overall functional features in Python, counting the number of occurrences of each construct across all files in the 2021 dataset.

In the case of the gen_invalid_vectors function, the generator appears to abstract an infinite list of values — the number of vectors that can be generated is unbounded at runtime (notice **while True** in Line 5). Such usage allows for better flexibility and likely better performance.

There are 10 cases where *yield* abstracts away application logic and makes it available as a generator. In these cases, a function codes a set of application-specific logic rules and yields a sequence of corresponding elements. For example:

```
def tokens(self, event, next):
    kind, data, pos = event
    if kind == START:
        tag, attribs = data
        name = tag.localname
        namespace = tag.namespace
        ...
        if ...:
            for token in self.emptyTag(...):
                yield token
        else:
            yield self.startTag(...)

    elif kind == END:
        ...
        if name not in voidElements:
            yield self.endTag(namespace, name)

    elif kind == COMMENT:
        yield self.comment(data)
    ...
```

This function is part of a XML parser, which uses *yield* statements to walk through a XML-tree structure. In this category of *yield* usage, there is logic for generation of HTML files, application settings, HTTP packets, and so on.

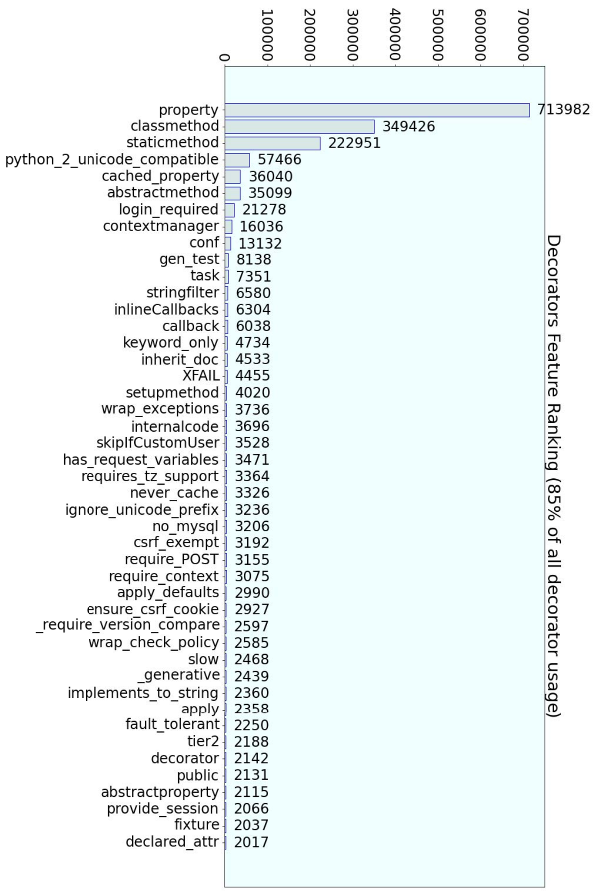
One observation is that in nearly all examples developers used multiple *yield* statements in generator functions. This places the burden on the client to ensure consistency when they retrieve expressions. In the above example, the expressions can be tags or comments and the client may need to make a distinction. It was unclear whether and how consistency of yield-generated elements is handled.

There are 11 cases that use *yield* as in the following Python code:

```
def createFields(self):
    # Access flags (16 bits)
    yield NullBits(self, "reserved[]", 4)
    yield Bit(self, "strict")
    yield Bit(self, "abstract")
    yield NullBits(self, "reserved[]", 1)
    yield Bit(self, "native")
    yield NullBits(self, "reserved[]", 2)
    yield Bit(self, "synchronized")
    yield Bit(self, "final")
    yield Bit(self, "static")
    yield Bit(self, "protected")
    yield Bit(self, "private")
    yield Bit(self, "public")
    ...
```

This function simply constructs a sequence of expressions without any logical filtering via if statements. It appears that these cases could be translated into invocations at the client side. They might be used as base cases for recursively-applied generators. We encountered this method of using *yield* multiple times in the programs we examined.

On a final note, while use of *yield* is ubiquitous, use of *yieldFrom* is surprisingly rare (see Fig. 4). We conjecture two reasons for this: (1) *yieldFrom* was introduced in Python 3.3 in 2012, while *yield* was introduced in Python 2.2 ten years earlier. (2) The semantics of *yieldFrom* is perhaps too complex and developers forgo usage. Overall, we conclude that *yield* statements are often used and static analysis ought to support them, while they can omit the more complex *yieldFrom* and still achieve good coverage of Python code. Fortunately, Fromherz et al. [10] have defined a static semantics for *yield*.



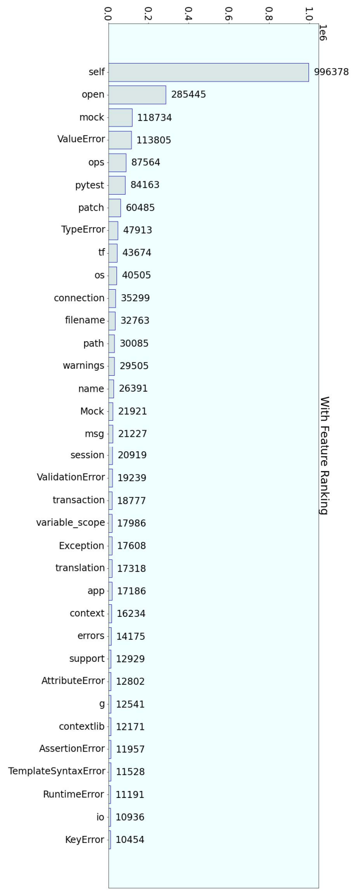


Figure 5: Occurrences of the most frequent decorator strings across all files in the 2021 dataset.

Figure 6: Occurrences of names in with-statement items.

3.2.3 Decorators. Another complex feature is the *decorator*. Tab. 1 and Fig. 1 show that decorators are widely used. About 12% of the files and 10% of repositories make use of decorators. Fig. 6 shows a histogram of the number of occurrences of different decorator strings in our 2021 dataset. As one can see, there is a wide variety of decorators — built-in decorators (e.g., @property, @classmethod), standard-library-defined (e.g., @abstractmethod), third-party-library-defined and user-defined ones.

Importantly, our results show that the built-in decorators @property, @classmethod, and @staticmethod are by far the most widely used. The semantics of these decorators is well-known and can be modeled in static analysis. (But they should not be ignored.)

3.2.4 With Statement. Fig. 5 shows a histogram of all AST Names that occur during the recursive traversal of items in with-statements. For example, in with contextlib.nested(mock.patch(...), mock.patch(...)): ... we count contextlib once and mock twice. High frequency of self, mock, pytest, and ...Error point to usage with testing frameworks. In most developers' experience with Python, *with* is used for opening a file in a block. As expected, our histogram shows high frequency of usage of open as well.

To understand usage of the *with* statement we examine 40 randomly selected Python programs. While a common use of *with* is for resource management, only 8 *with*-enclosed expressions were calls to *open*; the remaining uses of *with* came from testing frameworks. For example:

```
def test_onetoone_reverse_no_match(self):
    ...
    with self.assertNumQueries(0):
        with self.assertRaises(...):
            ...
```

As suggested by the histogram in Fig. 5 we observed self a lot during manual inspection. Methods in custom testing frameworks, as in the example above, are wrapped in the contextmanager decorator which turns the code into a context manager. This finding is consistent with Fig. 6 where the contextmanager decorator features as the 8th most frequently used one.

Or another example:

```
class IBMPluginV2TestCase(...):
    def setUp(self):
        with contextlib.nested(
            mock.patch(...),
            mock.patch(...)):
            super(IBMPluginV2TestCase,self)
                .setUp(plugin=_plugin_name)
```

The names of functions, variables, classes, or files led us to conclude that the uses are for testing purposes: they all contain keywords such as "test" or "testing".

We observed one user-defined method, which we found was a wrapper around the built-in *open* and it was passed into a mock object. These results lead to a conjecture that *with* statements are used for handling files and streams or in conjunction with testing frameworks. However, more mining and analysis is needed in order to confirm or refute this conjecture.

To expand the scope of the manual analysis, we analyzed an additional 30 random programs this time excluding *open* and testing frameworks. There are 24 user-defined methods that appear to function similarly to *mock* objects, and 6 enclosed statements that appear to just set the environment of the code block enclosed in the *with* statement. This leads us to a conjecture that developers may not grasp the role of context managers and may mistakenly believe that *with* statements introduce their own nested scope.

Although use of *with* statements is widespread, it appears in limited contexts: (1) testing frameworks, and (2) file management. Therefore, we conjecture that static analysis tools can define special handling in these contexts and still achieve good coverage and precision.

3.2.5 Async features. There are only a few occurrences of **async**. This might be because the feature is still new or because its semantics are not clear to developers. Somewhat surprisingly, **async** is the only feature that has lower usage frequency stand-alone than in combination with other features, as we will see in Table 3.

Feature Combination					2019 Frequency Count	2019 Frequency %	2021 Frequency Count	2021 Frequency %
(none)					620,300	51.353	1,822,194	58.891
FNL					177,236	14.673	365,869	11.824
	WTH				96,596	7.997	146,389	4.731
		DEC			78,579	6.505	126,778	4.097
FNL			DYN		28,096	2.326	113,120	3.656
			DYN		40,096	3.319	112,020	3.620
FNL	WTH				54,201	4.487	98,827	3.194
FNL		DEC			41,063	3.400	77,458	2.503
FNL		DEC	DYN		14,436	1.195	63,908	2.065
FNL	WTH	DEC			13,132	1.087	33,101	1.070
FNL	WTH	DEC	DYN		3,781	0.313	32,739	1.058
FNL	WTH		DYN		7,295	0.604	31,393	1.015
		DEC	DYN		9,819	0.813	25,692	0.830
	WTH	DEC			12,217	1.011	20,542	0.664
	WTH		DYN		6,957	0.576	17,373	0.561
	WTH	DEC	DYN		1,755	0.145	4,808	0.155
		DEC		ASC	441	0.037	355	0.011
				ASC	607	0.050	317	0.010
FNL		DEC		ASC	262	0.022	281	0.009
FNL				ASC	308	0.025	172	0.006
FNL		DEC	DYN	ASC	68	0.006	157	0.005
FNL	WTH	DEC		ASC	90	0.007	123	0.004
	WTH	DEC		ASC	112	0.009	91	0.003
FNL	WTH			ASC	124	0.010	82	0.003
	WTH			ASC	163	0.013	68	0.002
FNL	WTH	DEC	DYN	ASC	14	0.001	51	0.002
FNL			DYN	ASC	36	0.003	50	0.002
		DEC	DYN	ASC	49	0.004	42	0.001
			DYN	ASC	44	0.004	16	0.001
FNL	WTH		DYN	ASC	16	0.001	14	0.000
	WTH	DEC	DYN	ASC	10	0.001	8	0.000
	WTH		DYN	ASC	9	0.001	4	0.000

Table 3: Per-file feature combinations. Each file that uses exactly a given combination of the features is counted as 1.

3.3 RQ3: Does Use of Complex Features Increase or Decrease Over Time?

To address the third question we compare feature usage across the 2019 dataset and the 2021 dataset. Tab. 3 shows a detailed breakdown of the results presented earlier in Tab. 1 and Tab. 2. As Python is becoming feature-rich, we were curious whether Python usage is diverging. Specifically, it is conceivable that it either diverges by paradigm (e.g., functional vs. imperative) or by sophistication (beginner vs. advanced). Therefore, this table shows intersections of features; for example, FNL counts files that have only functional features but no other features, and FNL WTH DEC counts files that have functional features, **with** statements, and decorators, but no dynamic features and no **async**.

Overall, the number of files that have no complex features increased from 51% to 59% (row (none) in Tab. 3). The main takeaway is that even though there are some changes — the percentage of files with dynamic features increased from 9% to 13% and the percentage of *with* files decreased from 16% to 12% (Tables 2 and 1) — occurrence of complex features remains mostly stable from 2019 to 2021. Looking at the intersections, it seems like when a file uses some complex features, it often also uses others, possibly because it was written by a programmer who is more familiar with advanced language features across the board.

Fig. 7 and Fig. 8 present our final datapoint. The Venn diagrams show overlapping features per file. One can make several observations: (1) Dynamic feature usage has increased from 2019 to 2021 while functional features and decorators are about the same. Based on our analysis in RQ2, we conjecture that the increase is driven by use of *getattr* with constant string arguments. (2) The overlap of features per file, though significant, is a much smaller subset compared to the per-repository and per-organization Venn diagrams that we presented earlier (Fig. 1 and Fig. 2). This is expected as repositories typically include multiple Python files.

3.4 Not-so-Featherweight Python

To summarize, nearly 50% of all projects and over 40% of all files in the 2021 dataset use complex features. While it is reassuring that

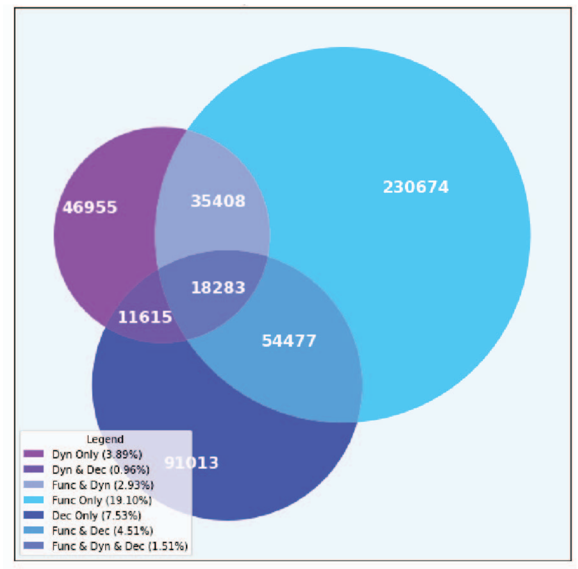


Figure 7: Per-file features from the 2019 dataset.

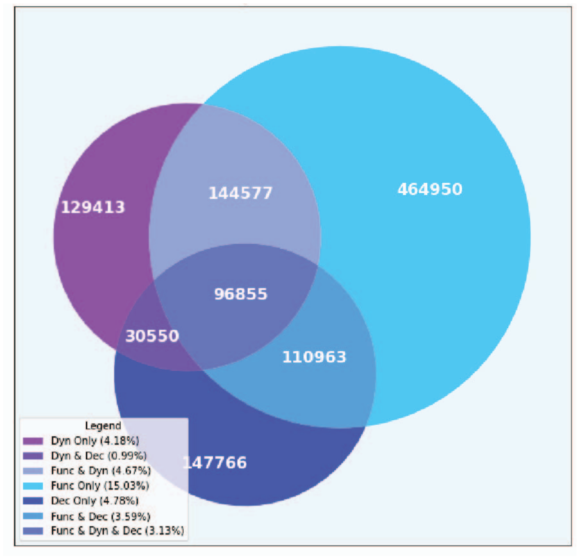


Figure 8: Per-file features from the 2021 dataset.

dynamic execution (i.e., *eval*) and *async* occur infrequently, uses of complex features such as *with* and decorators are widespread. A minimal "Featherweight Python" with a known imperative semantics is unfortunately a mirage. Our actionable result is a list of features that program analysis ought to handle in order to cover developer usage of Python features and move forward towards sound and effective analysis of Python:

(1) *getattr* with constant string argument occurs frequently. As discussed earlier, we believe this feature can be modeled effectively in both flow-sensitive and flow-insensitive analysis.
(2) *hasattr* with constant string argument occurs frequently as well. While in many cases the code takes an object (an instance of given class) as argument, in a number of cases it takes a class (a meta-object) as argument. We believe that the former can be modeled easily, but the latter requires a significant extension of standard static program analysis techniques with reasoning about classes and runtime modification of class structure.
(3) List comprehensions and generator expressions (via explicit constructors or *yield* statements) occur frequently. List comprehensions can be modeled effectively by rewriting them into a for-loop. Fortunately, Fromherz et al. [10] have defined a semantics for generator expressions.
(4) *lambdas* and higher-order functions occur frequently. This again requires extensions to flow analysis techniques for imperative languages.
(5) *With* statements and context managers. They present perhaps the biggest challenge to soundness and effectiveness of static analysis.
(6) Built-in decorators, particularly @property, @staticmethod, and @classmethod. A static analysis needs to be able to model the semantics of the corresponding built-in functions.

4 THREATS TO VALIDITY

Our study shares some threats to validity common among studies of language feature usage "in the wild". First, the set of complex features we select may be incomplete. Based on our experience with Python and static analysis, we believe we are covering an important set of features. However, Python has a rich set of features that constantly evolve.

Second, the scope of the analysis may be incomplete. We settled on analysis within the boundaries of the user code of a particular repository. We did not analyze standard libraries or third-party libraries imported with the project. This may underestimate usage since libraries may use complex features even if the user code does not. Sound static analysis ought to handle libraries as well. For this study we set a boundary — we settled on purely syntactic analysis, fixed the feature categories and analysis scope and focused on obtaining initial results. On the positive side, keeping our analysis simple and general makes the results interesting not just to tool builders but also to language users and educators.

One threat to validity, particularly the findings on RQ3, is the composition of the two 2019 and 2021 datasets. The two datasets may overlap, in which case we may overstate similarity of feature usage. We were unable to collect overlap and finer grained statistics on the datasets due to severe time constraints; we plan to do so as we continue work in this exciting direction.

Finally, our mining is static. It may be the case that a single syntactic occurrence never executes at runtime or that it executes thousands of times. Since our interest is static analysis and applicability of static analysis we think our numbers are still relevant. Dynamic analysis can bring additional information and insights.

5 RELATED WORK

Studies of programming languages “in the wild”. Not all programming language features get adopted widely, and sometimes they are used in surprising ways. There have been several studies that empirically explore how features are used in practice, typically based on mining a corpus such as from a software repository, as we do. In 2009, Holkner and Harland instrumented 24 Python application and studied their dynamic execution traces to understand how they use dynamic Python features [12]. They found many uses of getattr and setattr, no uses of delattr, and some uses of reflection over local or global symbol tables and of eval and exec. In 2014, Åkerblom et al. did a similar study on 19 Python application, with similar results [23]. Our paper expands on these results by studying a larger corpus and a broader (and newer) set of Python features including context managers and decorators. Pimentel et al. analyzed code in Jupyter notebooks, 93% of which use Python, and found that notebooks tend to use fewer complex features, often not even defining classes or functions [20]. In 2020, Rak-amnouykit et al. analyzed Python 3 type annotations in around 173,000 Python files mined from GitHub [21]. They found shockingly many type mistakes, and on top of that, found that popular type checking tools frequently disagree with each other. Our paper studies different Python features, looking not at types but at dynamic features, functional features, etc. In 2021, Peng et al. analyzed the ASTs of 35 Python repositories for a variety of Python features such as different flavors of parameter passing and class inheritance [19]. Our paper also analyzes ASTs, but considers orders of magnitude more Python repositories, and studies complex features such as comprehensions, generators, and context manager that they omit.

Going beyond Python, Richards et al. studied dynamic traces of browser-side JavaScript code for 100 websites [25]. They measured things like function size, prototype depth, kinds of objects, and polymorphism, as well as dynamic features: adding fields, adding methods, and eval. Their study showed that JavaScript tools cannot ignore the dynamic features, including eval. In a follow-on study, they expanded their corpus to 10,000 websites while narrowing their focus to eval [24]. They find that while many strings passed to eval contain JSON data, there are also many other categories of eval uses in JavaScript. Morandat et al. analyzed 3.9 million lines of code in the R language, with a focus on features affecting computational performance [17]. They found that R’s lazy evaluation incurs a high performance penalty and is used a lot in standard libraries but rarely in user code. In contrast, our study of Python focuses less on performance and more on features that inhibit bug finding and developer productivity tools. Dyer et al. analyzed 9 million Java files for their usage of features newly introduced by Java versions 2, 3, and 4 [9]. They found that the features most commonly adopted were Java annotations (similar to Python decorators) and generic types, while Java 4’s try-with-resources (similar to Python context managers) had not yet been adopted much at the time of their study. Our study includes some similar features but focuses on Python, whose tool support lags far behind that of Java.

We wrap up our discussion of “in the wild” studies with two pieces of work that are further afield but still related in spirit. Meyerovich and Rabkin explored the adoption of not individual language features but entire programming languages [15]. They find that “when considering intrinsic aspects of languages, developers prioritize expressivity over correctness”. We study Python features whose primary motivation is expressivity and that complicate the construction of tools for correctness. Tsay et al. mined 7,998 machine-learning models, mostly in Python and mostly from GitHub [27]. They focus on metadata that is needed to understand and reuse these models, and found that most models have poor data reproducibility but somewhat better method reproducibility. Our study is similar in that it focuses on GitHub Python code, but while they explore applications, we explore language features.

Minimal language subsets. We have motivated our paper with static analyses and tools that benefit from focusing on a language subset. By exploring how language features are used in the wild, we can help prioritize which features to model and to what extent. A famous language subset is Featherweight Java, a core Java subset with formal semantics just big enough to include classes and methods [13]. Featherweight Java focuses on type systems, and enables the authors to prove type safety, including for an extension with generics. Another well-known language subset from 2007 is RPython, a static Python subset [6]. Its main motivation was not theoretical proofs but rather computational performance: RPython is suitable as a target language for dynamic compilation and as a source language for static compilation. Our work differs in that we empirically study modern Python, focusing on features that complicate static analysis rather than compilation. Featherweight TypeScript is a subset of TypeScript plus associated type rules [7]. It helps delineate exactly how far the correctness guarantees of types go and where TypeScript becomes unsound.

Mainstream static analysis tools. PyLint [5] and PyType [4] are mainstream program analysis tools that provide style checking and linting (PyLint) and type checking and inference (PyType). PyLint in particular has been adopted widely in Python development [11]. We believe that our study, which aims to support deeper semantic analysis, can lead to improvement in mainstream productivity tools.

6 CONCLUSION

This paper explores how often and in what ways developers use certain complex features. Our goal is far-reaching — we wanted to understand use of complex features in order to build better static analysis tools. We analyzed millions of Python files mined from GitHub to address the following research questions: (i) How often do developers use certain complex Python features? (ii) In what ways do developers use these features? (iii) Does use of complex features increase or decrease over time? We present an actionable result — a list of Python features that any “minimal syntax” ought to handle in order to capture developers’ use of the Python language.

REFERENCES

[1] [n. d.]. Depends: Comprehensive Code Dependency Analysis Tool. https://github.com/multilang-depends/depends (retrieved March 2022).
[2] [n. d.]. Pyan: Static Analysis of Python Code. https://github.com/davidfraser/pyan (retrieved March 2022).
[3] [n. d.]. The Python Language Reference: 8. Compound Statements. https://docs.python.org/3/reference/compound_stmts.html (retrieved March 2022).
[4] [n. d.]. pytype: A static type analyzer for Python code. https://google.github.io/pytype/ (retrieved March 2022).
[5] 2001. PyLint: Code analysis for Python. https://pylint.org (retrieved March 2022).

[6] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. 2007. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Dynamic Languages Symposium (DLS)*. https://doi.org/10.1145/1297081.1297091

[7] Gavin Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *European Conference for Object-Oriented Programming (ECOOP)*. 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

[8] Julian Dolby, Avraham Shinnar, Allison Allain, and Jenna Reinen. 2018. Ariadne: Analysis for Machine Learning Programs. In *Workshop on Machine Learning and Programming Languages (MAPL)*. 1–10. http://doi.acm.org/10.1145/3211346.3211349

[9] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. 2014. Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features. In *International Conference on Software Engineering (ICSE)*. 779–790. https://doi.org/10.1145/2568225.2568295

[10] Aymeric Fromherz, Abdelraouf Ouadjaout, and Antoine Miné. 2018. Static Value Analysis of Python Programs by Abstract Interpretation. In *NASA Formal Methods Symposium (NFM)*. 185–202. https://doi.org/10.1007/978-3-319-77935-5_14

[11] Hristina Gulabovska and Zoltán Porkoláb. 2019. Survey on Static Analysis Tools of Python Programs. In *Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA)*. http://ceur-ws.org/Vol-2508/paper-gul.pdf

[12] A. Holkner and J. Harland. 2009. Evaluating the dynamic behaviour of Python applications. In *Australasian Computer Science Conference (ACSC)*. 17–25. https://crpit.scem.westernsydney.edu.au/abstracts/CRPITV91Holkner.html

[13] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)* 23, 3 (May 2001), 396–450. https://doi.org/10.1145/503502.503505

[14] Li Li, Jiawei Wang, and Haowei Quan. 2022. Scalpel: The Python Static Analysis Framework. https://arxiv.org/abs/2202.11840

[15] Leo A. Meyerovich and Ariel S. Rabkin. 2013. Empirical Analysis of Programming Language Adoption. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 1–18. https://doi.org/10.1145/2544173.2509515

[16] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Static Type Analysis by Abstract Interpretation of Python Programs. In *European Conference on Object-Oriented Programming (ECOOP)*. 17:1–17:29.

[17] Floréal Morandat, Brandon Hill, Leo Osvald, and Jan Vitek. 2012. Evaluating the Design of the R Language. In *European Conference for Object-Oriented Programming (ECOOP)*. 104–131. https://doi.org/10.1007/978-3-642-31057-7_6

[18] Chris Okasaki. 1995. Simple and efficient purely functional queues and deques. *Journal of Functional Programming (JFP)* 5, 4 (1995), 583–592. https://doi.org/10.1017/S0956796800001489

[19] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *Conference on Software Analysis, Evolution and Reengineering (SANER)*. 24–35. https://doi.org/10.1109/SANER50967.2021.00012

[20] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A Large-Scale Study About Quality and Reproducibility of Jupyter Notebooks. In *Conference on Mining Software Repositories (MSR)*. 507–517. https://doi.org/10.1109/MSR.2019.00077

[21] Ingkarat Rak-amnouykit, Daniel McCrevan, Ana Milanova, Martin Hirzel, and Julian Dolby. 2020. Python 3 Types in the Wild: A Tale of Two Type Systems. In *Dynamic Languages Symposium (DLS)*. 57–70. https://doi.org/10.1145/3426422.3426981

[22] Ingkarat Rak-amnouykit, Ana Milanova, Guillaume Baudart, Martin Hirzel, and Julian Dolby. 2021. Extracting Hyperparameter Constraints from Code. In *ICLR Workshop on Security and Safety in Machine Learning Systems (SecML@ICLR)*. https://aisecure-workshop.github.io/aml-iclr2021/papers/18.pdf

[23] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing Dynamic Features in Python Programs. In *Conference on Mining Software Repositories (MSR)*. 292–295. https://doi.org/10.1145/2597073.2597103

[24] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do: A Large-Scale Study of the Use of Eval in JavaScript Applications. In *European Conference for Object-Oriented Programming (ECOOP)*. 52–78. https://doi.org/10.1007/978-3-642-22655-7_4

[25] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *Conference on Programming Language Design and Implementation (PLDI)*. 1–12. https://doi.org/10.1145/1806596.1806598

[26] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *International Conference on Software Engineering (ICSE)*. 1646–1657.

[27] Jason Tsay, Alan Braz, Martin Hirzel, Avraham Shinnar, and Todd Mummert. 2020. AIMMX: Artificial Intelligence Model Metadata Extractor. In *Conference on Mining Software Repositories (MSR)*. 81–92. https://doi.org/10.1145/3379597.3387448

[28] Weijie Zhou, Yue Zhao, Guoqiang Zhang, and Xipeng Shen. 2020. HARP: holistic analysis for refactoring Python-based analytics programs. In *International Conference on Software Engineering (ICSE)*. 506–517.