An Empirical Study for Common Language Features Used in Python Projects

Yun Peng, Yu Zhang*, Mingzhe Hu
Lab for Intelligent Networking and Knowledge Engineering (LINKE)
University of Science and Technology of China, Hefei, China
Email: py2016@mail.ustc.edu.cn, yuzhang@ustc.edu.cn, hmz18@mail.ustc.edu.cn

***Abstract*—As a dynamic programming language, Python is widely used in many fields. For developers, various language features affect programming experience. For researchers, they affect the difficulty of developing tasks such as bug finding and compilation optimization. Former research has shown that programs with Python dynamic features are more change-prone. However, we know little about the use and impact of Python language features in real-world Python projects. To resolve these issues, we systematically analyze Python language features and propose a tool named PYSCAN to automatically identify the use of 22 kinds of common Python language features in 6 categories in Python source code. We conduct an empirical study on 35 popular Python projects from eight application domains, covering over 4.3 million lines of code, to investigate the the usage of these language features in the project. We find that single inheritance, decorator, keyword argument, for loops and nested classes are top 5 used language features. Meanwhile different domains of projects may prefer some certain language features. For example, projects in DevOps use exception handling frequently. We also conduct in-depth manual analysis to dig extensive using patterns of frequently but differently used language features: exceptions, decorators and nested classes/functions. We find that developers care most about ImportError when handling exceptions. With the empirical results and in-depth analysis, we conclude with some suggestions and a discussion of implications for three groups of persons in Python community: Python designers, Python compiler designers and Python developers.**

I. INTRODUCTION

Python has become one of the most popular programming languages, and is widely used in many fields such as artificial intelligence and data science. According to GitHub Octoverse 2019 [20], Python outranked Java as the second most popular language on GitHub among repository contributors. Python is constantly evolving, including extending language constructs or features to enhance the language expressiveness, or improving the performance or functionality of core libraries, *etc.* For example, Python 3.8 introduces *positional-only* parameters to allow pure Python functions to fully emulate the behavior of existing C coded functions, or allow the parameter name to be changed in the future without affecting client code [25].

Various language features and libraries of Python bring much convenience to developers, especially its flexibility, expressiveness and succinctness as a dynamic language. However, due to the evolution and dynamic features of the language, Python typically pays in weaker performance and safety [44], and brings problems for developers to build and maintain Python applications. For example, objects in Python may change their types in execution, which can cause type errors and make it hard to infer and check the type of Python objects.

Researchers have proposed some solutions to address the above challenges. *PySonar2* [49], a Python type inference tool proposed to avoid type errors cooperated with type checkers by automatically infer types of variables in source code, can only have 49.47% accuracy in real-world programs [53]. To improve the accuracy of type inference, Xu *et al.* combined probability and machine learning methods to infer types in Python [53]. Furthermore, recently researchers try to capture the natural language features of code by introducing deep learning models in this task and has accomplished better accuracy [3, 9, 33]. Another example is Numba [4], a Python Compiler, which tries to compile restricted Python source code to LLVM IR in order to accelerate the execution of Python programs by reusing the LLVM backend. These solutions are effective and valuable, but often encounter new problems or challenges for certain Python language features. In view of the large scale of real-world Python projects, often exceeding 10 or even 100 thousands of lines of code (KLOC) as shown in Table II, we think it will be very valuable to conduct an empirical study to find the distribution of common language features used in Python projects and help Python users better use these features by digging language feature using patterns from popular Python projects.

In this paper, we systematically summarize 6 categories of 22 language features and develop an automatic language feature scanner named PYSCAN. PYSCAN combines Abstract Syntax Tree (AST) traversal, type inference and standard library scanning to comprehensively identify the usage of language features. It accepts Python source files as input and reports the language feature usage. We also conduct in-depth analysis for some commonly and differently used language features such as exception handling by manually checking their using scenarios. Finally, we conclude some suggestions and implications from the distribution and analysis of these language features for three user groups consisting of the whole Python community: Python designer, Python compiler designer and Python developers.

To our knowledge, this is the first study to systematically analyze language features and automatically identify their use in Python projects via static analysis. Although Malloy *et al.* [31] have investigated the transition from Python 2 to

Python 3 by using PyComply, they only focus on a limited number of features at the grammatical level; while some other studies focus on analyzing certain features such as dynamic features [34], polymorphism [35], or code changes [29].

The main contributions of this paper are as follows:

- We summarize 22 kinds of common language features, which are divided into 6 categories including *function, type system, object-oriented programming, data structure, metaprogramming* and *evaluation strategy*.
- We develop an automatic language feature recognizer named PYSCAN to identify and collect the usage characteristics of language features in different Python projects.
- We analyze the distribution of language feature usage for 35 popular Python projects from 8 popular domains and find that except for general used language features different domains focus on some different features, among which exception handling statements, decorators and nested classes/functions are used most differently.
- We conduct in-depth analysis on exception handling statements, decorators and nested classes/functions, and then summarize their using scenarios and advantages.
- We conclude some suggestions and implications for developers and researchers targeting Python from the empirical results and in-depth analysis of language features.

II. METHODOLOGY

In this section, we first highlight the specific research questions (RQ) we wish to answer in Part II-A. Then we introduce common language features we concern in Part II-B. In Part II-C we present PYSCAN to automatically analyze language features in Python projects, including its overall architecture, and key information used in the scanning process. Finally we explain the techniques we used to analyze empirical results collected by PYSCAN in Part II-D and the dataset we used to conduct our empirical study in Part II-E.

A. Research Questions

RQ1: What is the general distribution of language features in real-world Python projects?

The continuous evolution of Python makes the language features constantly changing. Some language features bring challenges to the analysis of Python programs on correctness, safety and improving performance. Compared with the program analysis of these goals, the existence of language features in Python source code is much easier to judge. If certain language features are not used in practice, then there is no need to brainstorm to explore their safety issues, optimization and so on. In this research question, we hope to get a general view of what language features are used in real-world Python projects and how their usage is distributed, and find out commonly used and rarely used language features.

RQ2: What are the differences of language feature usage distribution among different domains of Python projects?

Projects in different domains may be developed under different requirements to accomplish various tasks. Do they show similar trends in the use of language features? If the distributions of language features used by projects across different domains are similar, we can rank these features from the most to the least common, and summarize the general rules for using these features. Otherwise, if a project in a certain domain is very different from other projects in the use of language features, we should pay more attention to summarizing the usage scenarios of these features with usage differences, and give suggestions on coding, error detection, and optimization accordingly.

RQ3: Why are certain language features used frequently and how are they used?

Different language features can have different expression intentions and use occasions. There are always more than one available choice of language features for one circumstance. Why do the developers of popular Python projects prefer certain language features over others? Do these language features have the unique advantages of improving the safety and performance of projects or truly bring much convenience and flexibility for them? And do these language features have wider uses beyond their original use? In this RQ, we hope to dig more valuable and extensive using patterns of frequently used language features and figure out why such usage can improve the quality of Python projects. It is significant for the entire Python community to understand the extensive use of these language features. With the guidance of using patterns derived from popular Python projects, developers are more likely to make better choices when choosing language features.

RQ4: How does this empirical study help improve the design and quality of Python tasks?

As a popular dynamic programming language, Python is used among different user groups. They may focus on different characteristics of this language, but they all have to interact with its language features. How can our study help these people improve their work? In this RQ, we summarize implications from empirical results and in-depth manual analysis of language features. We hope to provide advice to different types of Python users such as Python compiler designers, Python application developers.

B. Common Language Features We Concern

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. We want to understand the characteristics and usage of common language features of these programming paradigms in Python. By analyzing the Python language specification, we focus on the 22 common language features listed in Table I that reflect the special characteristics of Python, which are divided into the following 6 categories:

- **Function**. Functions are the basis for encapsulating a group of operations to perform a task, which closely connect to the functionality of the program. We choose 9 language features related to functions, including 5 API-related (arguments and multiple return), 3 function-definition-related and 1 exception-related. The API affects the correctness and safety of use and interoperability, the function definition of special features affects the

TABLE I
SIX CATEGORIES OF LANGUAGE FEATURES IN PYTHON

Category	Language Feature	Scanning Strategy	Relative AST node
A. Function	Keyword Argument [18]	L-AST	*argument*
	Keyword-only Argument [11]	L-AST	*argument*
	Positional-only Argument [18]	L-AST	*argument*
	Multiple Return	L-AST	*Return*
	Packing and Unpacking Argument	L-AST	*argument*, *Call*
	Decorator	L-AST	*FuncDef*
	Exception	L-AST	*Call*,*Raise*, *Try*
	Recursion [23]	G-AST	*FuncDef*, *Call*
	Nested Function [39]	G-AST	*FuncDef*
B. Type System	First-class Function [2]	L-AST & Type & Std	*argument Return*, *Assign*
	Gradual Typing [41, 45, 52]	L-AST	*FuncDef*
C. Loop & Evaluation Strategy	Loop	L-AST	*For*,*While Continue*, *Break*
	Generator	L-AST	*Yield*
D. Object-Oriented Programming	Inheritance	G-AST	*ClassDef*
	Polymorphism [8]	L-AST & Type & Std	*argument*
	Encapsulation [37]	L-AST	*Name*, *ClassDef*
	Nested Class [40]	G-AST	*ClassDef*
E. Data Structure	List Comprehension [17]	L-AST	*ListComp*
	Heterogeneous List and Tuple	L-AST & Type & Std	*SubScript*
F. MetaProgramming	Introspection [12]	L-AST	*Call*
	Reflection	L-AST	*Call*
	Metaclass [16]	L-AST	*ClassDef*

complexity of analysis and implementation, and the use of exceptions can reflect the robustness of the code.

- **Type system**. As a dynamic programming language, Python has a more flexible type system which allows variables to change types at runtime. However, such feature can also increase the occurrence of type errors, and it may be difficult to build checking and inference tools to find such errors. In this category, we focus on 2 language features including first-class functions whose arguments or return values can be functions, and gradual typing [38, 47] that allows one to annotate only part of a program. The former increases the difficulty of program analysis, while the latter can leverage desirable aspects of both dynamic and static typing.
- **Loop & Evaluation Strategy**. Performance of programs is always an important topic in both industry and academia. For programs written in any language, the use of loops largely affects the performance of programs. For Python, the lazy evaluation strategy introduced by *generators* can have performance implications, both for memory management and function run time. However, such strategy may confuse users and lead to logical errors.
- **Object-Oriented Programming (OOP)**. Python mainly uses classes to implement the concept of OOP. Inheritances of classes occur frequently but some of them such as diamond inheritance can lead to class initialization problem. Polymorphism and nested class have extensive usage in fields such as testing while encapsulation increases the safety by hiding some information in class.
- **Data Structure**. Python's dynamic feature allows the existence of heterogeneous data structures. However, using such data structures is highly risky since we have no sense about the result when we index these data structures. Users may forget to handle different possible types generated from them and introduce type errors. Therefore, this paper focuses on Python-specific *list comprehension* and *heterogeneous lists or tuples*, which increase the difficulty of type inference and program analysis.
- **Metaprogramming**. Metaprogramming is a programming technique in which programs have the ability to treat other programs as their data. This technique is popular with code framework developers. Metaprogramming in Python relies on type introspection, reflection and metaclass *etc.* These features enhance the flexibility and expressivity of the program, but also increase the difficulty of static analysis *etc.* So we want to recognize their use in real-world Python projects.

C. PYSCAN: *Python Language Feature Scanner*

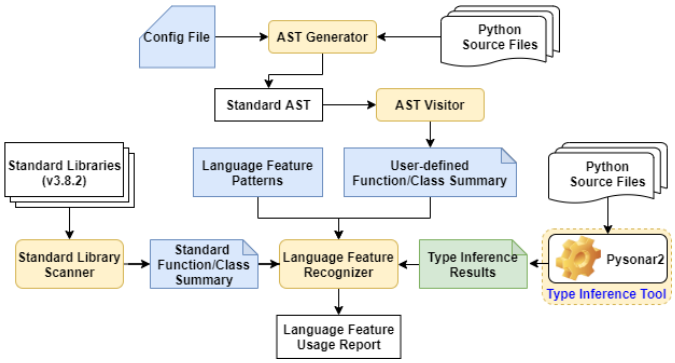

Fig. 1. Overview architecture of PYSCAN

In order to collect language feature usage information from real-word Python projects, we propose an automatic language feature scanner named PYSCAN, whose architecture is shown in Fig. 1. There are five major parts in PYSCAN:

AST Generator, which parses Python source code in the Python project to generate corresponding abstract syntax tree (AST) by simply calling the standard module *ast*.

AST Visitor, which traverses the whole AST to identify specific types of AST nodes listed in column 4 of Table I, and further collect three kinds of information required for language feature recognition: (1) function and class names, (2) class inheritance relationship, (3) function call relationship.

Type Inference Tool, which infers the types of Python objects. Type information is essential for identifying language features such as first-class function, heterogeneous list and tuples. PYSCAN loosely couples with existing type inference tools to obtain type inference results of the Python code. Since we want to perform fast type inference on a large number of Python source files, we choose *PySonar2* [49] with relatively

high accuracy instead of the recent type inference method based on deep learning models [3, 33] with higher accuracy.

Standard Library Scanner, which is a simplified version of PYSCAN and scans three kinds of information in standard libraries in advance: 1) all function definitions and their locations; 2) all functions with parametric polymorphism and their locations; 3) all functions returning heterogeneous lists and tuples and their locations. Since standard libraries do not change frequently, such information can be stored in advance and used to increase accuracy of PYSCAN and avoid the loss of true-positives when scanning real-world Python projects.

Language Feature Recognizer, which accepts information from the above four parts, compares such information with language feature patterns designed in Section III and finally generates the language feature usage report. Columns 3∼4 in Table I list the scanning strategy and relative AST nodes of each language feature. There are 4 scanning strategies:

- L-AST, which means that scanning of a language feature is only related to a single AST node.
- G-AST, which means that scanning of a language feature needs global information of AST.
- Type, which means that scanning of a language feature needs information from type inference.
- Std, which means that scanning of a language feature needs information from standard libraries.

D. Result Analysis

To answer RQ1 and RQ2, we use PYSCAN to count the number of every language feature used in a project. Since some language features related to functions appear more frequently than those related to classes, we normalize the result by dividing the number of language features by the number of functions or classes to remove such difference. The normalized data can be seen as what proportion of functions or classes uses this feature. For RQ1 and RQ2, we separately combine the results of all projects and sub-domain projects, calculate the overall distribution and sub-domain distribution of language features, and analyze the difference between the sub-domain distribution and the overall distribution.

To answer RQ3, we conduct an in-depth analysis for some frequently and differently used language features. We manually check and summarize their using patterns in the source code and dig their extensive usage. For each frequently used language feature, we further collect more detailed data for a deeper understanding.

To answer RQ4, we make use of results collected in the first three RQs and read document of some popular Python projects to discover the advantages and disadvantages of using certain language feature. With all these information, we conclude some suggestions for certain Python user groups such as Python compiler designers, Python application developers.

E. Dataset

We collect 35 popular and influential Python projects from Github according to 2019 Annual Report of Github [20] and 2019 Python Survey of Jetbrains [27], and divide them into 8 domains shown in Table II. The first two columns show the domain name and number of Python projects in each domain. These projects do not always contain only Python source files, so we first filter all other files out and count the number of Python code lines and number of files as shown in columns 3∼5 and 6∼8. The whole dataset contains 25,059 Python source files and about 4.3 million lines of Python code. The average ratio of Python code is about 38%, which is calculated by dividing lines of Python code by lines of all code. The detailed ratios of Python code for projects in each domain are listed in columns 9∼11. To make our empirical results more representative, we list the Github stars for projects in each domains in the last three columns. As we know, Github star intuitively shows the contribution and influence of the project and the preference of other developers.

III. LANGUAGE FEATURES AND THEIR DISTRIBUTION

In this section, we first introduce the selected six categories of language features in turn, and then show the empirical results collected by PYSCAN when answering RQ1 and RQ2.

A. Functions

1) Keyword/Keyword-only/Positional-only Parameters: *Keyword* parameter can accept an argument preceded by an identifier (*e.g.,* name=) in a function call or a value in a dictionary preceded by "**" [18]. It always appears between separator "/" and "*". *Keyword-only* parameter is a named parameter placed after separator "*", which is introduced in Python 3 to avoid being automatically filled by a positional parameter [43]. *Positional-only* parameter appears to the left of separator "/", which has no externally-usable name and is passed in parameter order. It is introduced in Python 3.8 to obtain performance benefits and better API design [24].

2) Multiple Return: Functions in Python can return an object holding multiple values, which is similar to C/C++ and Java. They can also return multiple values by a tuple without parentheses or a list with square brackets.

3) Packing and Unpacking Arguments: Python provides some methods to pack and unpack arguments in function calls. If it is not sure how many parameters to pass in, add "*" before a parameter name in a function definition to pack the parameters. "*" (for list or tuple) and "**" (for dictionary) are used to unpack packed parameters and pass them to a function.

4) Decorator: Decorators allow developers to modify the behavior of function or class [50, 51]. They allow developers to wrap another function in order to extend the behavior of the wrapped function without permanently modifying it.

5) Exception: Exceptions are errors occurred during execution. The raise statement triggers exceptions which can be captured by the try...except statement to handle in Python. Moreover, parameters in exceptions can affect the type inference of some Python compilers like Numba [28], which only accepts exceptions with constant parameters to reduce the complexity of type inference.

TABLE II
EIGHT DOMAINS OF PYTHON PROJECTS SCANNED BY PYSCAN

Domain	Nums of Projects	KLOC in Python			Nums of Python Files			Ratio of Python Code			Github Star (k)		
		Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min
Web	5	6.3	24	0.6	550	2034	34	51.38%	84.17%	48.07%	30.7	52.7	1.5
Data Science	5	157	272	117	670	856	417	41.52 %	74.42%	20.60%	12.8	26.8	2.4
ML & DL Framework	6	197	605	19	949	2555	171	25.35%	85.00%	14.80%	55.7	149	19
AutoDrive	2	20	23	17	193	278	108	4.53%	4.57%	4.48%	17.1	-	-
Quantum Computing	6	45	91	18	375	667	197	72.53%	91.79%	32.33%	1.4	2.9	0.4
DevOps	5	331	947	4	1902	6336	45	73.82%	84.24%	48.48%	15.3	45.0	1.0
CV	3	14	22	2	100	182	12	7.62%	91.76%	3.45%	4.6	8.4	2.3
Image Processing	3	26	44	10	272	522	25	55.87%	66.94%	46.09%	5.4	7.8	4
Total	**35**	**4369**			**25059**			**38.51% (Avg)**			**683.7**		

6) Recursion: Recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem [23]. It can significantly reduce code complexity and is a common programming paradigm.

7) Nested Function: Python supports nested function, which is defined inside another function. Nested functions can be used to create closures, where outer functions returns inner functions. Closures can avoid the use of global values and provides some form of data hiding [39].

B. Type System

1) First-class Function: Python supports first-class functions, which can be passed as arguments, returned as return values, and assigned to variables [2].

2) Gradual Typing: Gradual typing is a type system that allows parts of a program to be dynamically typed and other parts to be statically typed [41]. Type hints and function annotations [45, 52] make gradual typing possible in Python. This feature may significantly reduce type errors at run-time and make Python much safer.

C. Loop & Evaluation Strategy

1) Loop: The for/while loops are the main time and memory consuming part of programs, so identifying them is important for optimization. In addition, the break and continue used to control the loop also need to be recognized.

2) Generator: Python uses eager evaluation in most cases. However, it also supports lazy evaluation in *generator*. Generator functions created using yield statement, allow developers to declare a function that behaves like an iterator.

D. Object-Oriented Programming

1) Inheritance: Python supports five types of inheritance, *i.e.,* single, multiple, multilevel, hierarchical and diamond. The former two are more common, so we only explain the others:

- Multilevel inheritance, which means a class inherits from a derived class.
- Hierarchical inheritance, which means more than one class is derived from the same class.
- Diamond inheritance, which occurs if classes B and C inherit from a superclass A, and another class D inherits from B and C. Such inheritance may affect the initialization and method call of classes and increase the complexity of code, making programs hard to maintain.

2) Polymorphism: Python supports parametric polymorphism, in which a function or a data type can be written generically regardless of the type of values. Both standard and user-defined functions may have parametric polymorphism.

3) Encapsulation: It is used to hide the value or state of structured data objects in a class to prevent unauthorized parties from directly accessing them. In Python, members in a class are public by default, and it is agreed that members prefixed with "_" are protected and those prefixed with "__" are private [46]. But it is just a convention and does not prevent instance variables from accessing or modifying such members of the instance, *e.g.,* "e1._a=20".

4) Nested Class: A nested class is a class declared entirely within the body of another class. It is usually used to group two or more classes or hide classes from the outside.

E. Data Structure

1) List Comprehension: As one of Python's most distinctive features, list comprehension provides a *concise* way to create lists based on existing lists [17]. However, using them too much or writing them too long may cause the code to be inefficient and hard to read.

2) Heterogeneous List and Tuple: Lists and tuples are commonly used data structures, but they are often used in different situations for different purposes. Lists are mutable and used to store homogeneous elements, while tuples are immutable and used to store heterogeneous elements. However, heterogeneous lists and tuples are hard to implement thus compilers such as Numba provide limited support for them.

F. Metaprogramming

1) Introspection/Reflection: Introspection is the ability of a program to examine the type or properties of an object at runtime [12]. Reflection is a step forward than introspection, indicating the ability to examine and modify the behavior of code at runtime. They both are dynamic features which are hard to analyze statically.

2) Metaclass: By default, classes are constructed using type(). The class body is executed in a new namespace and the class name is bound locally to the result of type(name, bases, namespace). Python provides a method to customize class definition process by passing the *metaclass* keyword parameter in the class definition line, or by inheriting from an existing class which includes such a parameter [16].

G. Distribution of Language Features in Python Projects

We have realized the recognition of the above six categories of language features according to the architecture of PYSCAN introduced in Section II-C. Then we scan 35 real-world Python projects mentioned in Section II-E using PYSCAN to quantitatively answer RQ1 and RQ2.

RQ 1: What is the general distribution of language features in real-world Python projects?

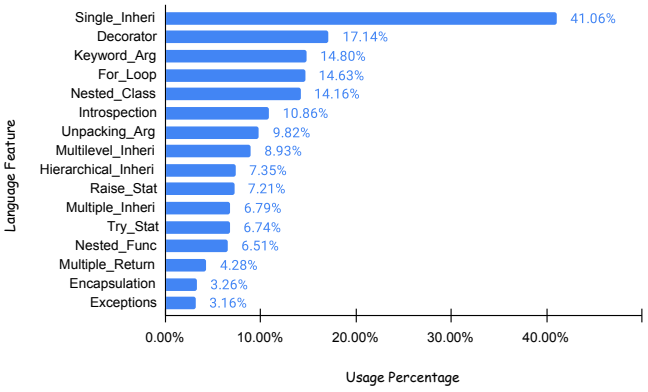

Fig. 2. General distribution of language feature usage in 35 Python projects

In total 35 Python projects from 8 different application domains, PYSCAN scans 25059 files, covering more than 4.3 million lines of code. Fig. 2 shows the 16 most frequently used language features. The percentage of each language feature shows the proportion of functions/classes used them.

> **Finding 1:** *Single inheritance*, *decorator*, *keyword parameters*, for *loop* and *nested class* are top 5 used language features in real-world projects, while *position-only parameters, heterogeneous list and tuple, keyword-only parameters*, *function assigned to another variable* are bottom 5 used language features.

Inheritance: Compared to multilevel inheritance (8.93%), multiple inheritance (6.79%), hierarchical inheritance (7.35%) and diamond inheritance (0.90%), single inheritance (41.06%) is the main type developers choose in all 5 types of inheritances. This result matches our intuition because single inheritance forms the basic usage of classes and is the foundation of the other four. However, we want to emphasize that single inheritance may occur with diamond or multilevel inheritances at the same time. Thus developers are advised to carefully check the inheritance relationship even if a class has only one base class. Diamond inheritance itself actually is hardly used, which may be because it is complex and can easily lead to class initialization problems. The using frequency of the remaining three inheritances are almost equal, and accounts for a relatively high percentages (6.79%∼8.93%). This result shows the diversity of classes in popular Python projects.

Parameters: Regarding parameters in all function definitions, 14.80% of functions use keyword parameters, while 0.14% of functions use keyword-only parameters and no function uses position-only parameters. The benefits of keyword parameters are obvious: not only can it be assigned default values to avoid being passed every time, but also maintain interface consistency so that changes of the position of parameters in the interface will not affect the client code. However, position-only parameter is newly introduced in Python 3.8 released in Oct. 2019, while the use of keyword-only parameter is a bit complicated and some existing Python compilers or analyzers may not support it. This may be the reason why these two kinds of parameters are not widely adopted in the Python community.

Other Features of Higher Usage: Safety checks such as exception handling statements (17.11%), which contains Exceptions and try,raise statements, testing techniques such as nested class (14.16%) / nested function (6.51%), decorators (17.14%) and dynamic features such as introspection (10.86%) bring more energy to Python programs compared to traditional static languages. We will discuss the former three in Section IV with code examples in detail. For dynamic features such as introspection, after manually checking the source code we find that developers usually use function calls with introspection to check attributes and avoid possible errors. For example, super() is used in the initialization function of classes to avoid initializing a base class two or more times. isinstance() is used to get the type information of variables and add a type check.

> **Finding 2:** Developers of popular Python projects tend to use relatively simple language features focused on safety checks, testing and some dynamic features, but avoid using those complex and error-prone features such as heterogeneous list and tuple, diamond inheritance, *etc.*

We find that those seldom-used language features have commonalities: they are often complex to implement or error-prone. As mentioned earlier, heterogeneous lists and tuples pose challenges to compiler design and are prone to type errors. Corresponding to our inference, they are actually rarely used (both less than 0.1%). Metaclasses (0.3%) are hard to implement and understand. For first-class functions, developers prefer to use function as argument (2.1%), rather than as return value (0.3%) or assign to another variable (0.3%). The reasons may be as follows: 1) language features such as decorators or callback design pattern use functions as parameters, 2) developers usually encapsulate common operations as functions and pass them to other functions to avoid code redundancy.

Type inference is important for early detection of type errors and better compilation to produce reliable and efficient code. Although recent type inference methods based on deep learning can improve the accuracy of inference to a certain extent, the effect of such methods depends on the dataset used. In order to give more auxiliary information to type inference,

Python introduces gradual typing feature for developers to annotate types. However, in our empirical study we find that popular Python projects slightly use gradual typing, with a usage rate of only 0.6%. We advise developers to add more type annotations to their programs, which can not only help deepen type-related research, but also enhance the type safety of projects since these annotations can be used by type checkers to find type errors. Similarly, keyword-only parameter has been introduced to avoid misuse caused by rapid API changes since 2006 [43]. However, current Python projects seem not to adopt this feature and still focus on keyword parameters to reduce interface changes.

Finding 3: Some language features designed to enhance the safety of Python projects such as gradual typing and keyword-only parameters are not widely used in real-world Python projects.

RQ 2: What are the differences of language feature usage distribution among different domains of Python projects?

We further analyze the impact of the application domain of the project on the use of language features. Fig. 3 shows the usage rate of ten language features in Python projects from eight domains. Some commonly used language features discussed in RQ1 are not included . From this figure, we can see the differences in the use of language features of Python projects from different domains.

Finding 4: Apart from those commonly-used language features, Python projects from different domains use different language features according to their domain characteristics.

We summarize the differently used language features and possible reason as follows:

- *Autodrive*: Projects in this domain have the highest proportion of using **for loops** (21.86%), no matter compared with the same feature in other domains (less than 16%) or other features in the same domain (less than 10%). After a manual check of the source code, we conclude a possible reason that these projects repeatedly receive information from sensors and handle them to guide driving.
- *CV*: Projects in this domain uses more **gradual typing** and **decorators**. Similar to the following ML & DL framework, decorators similar are mainly used to handle computations. However, the use of gradual typing is surprising. Unlike the following Quantum Programming projects which introduce some new types, CV projects do not seem to have a strong motivation than other domains. Combined with the lack of exception handling statements and nested classes, a possible explanation is that developers in CV may focus more on the type errors.
- *Data Science & Image Processing*: Language feature distribution in these two domains does not show typical differences compared with others. Actually their distribution looks closely to the general distribution in RQ1.
- *DevOps*: Projects in this domain typically pay more attention to exception handling as they use the most **try statements**. This may be because they try to achieve program safety and stability through runtime exception capture. In addition, compared to other domains, projects in this domain use the most **argument unpacking**, partly because they use more exception handling and decorators. Such two kinds of features always accept a function and its packed parameters as parameters.
- *ML & DL Framework*: Compared with other domains, projects in this domain use the most **decorators** and **nested functions**. Such projects are computationally intensive and implement a lot of algorithms and models to abstract features from the dataset. Some common operations and even compilation strategies (*e.g.,* @torch.jit.* in PyTorch [32]) can be defined as functions or decorators to reuse in different algorithms. This not only reduces the code size, significantly improves performance and compatibility, but also makes the code easier to understand.
- *Quantum Programming*: Projects in this domain create new types (*e.g.,* qubit) that do not exist in Python. Python compilers or checkers do not check these new types, so adding annotations is necessary to avoid type errors. Thus they use **gradual typing** which is rarely used in other projects, as well as more **decorators** and **introspection**.
- *Web*: Projects in this domain use the most **nested classes** compared with other domains. One of the important uses of nested classes is testing. We manually analyze source code and find there are indeed a lot of code in these projects performing tests. Unlike DevOps projects requiring dynamic exception handling, Web projects prefer to write more test code to find most bugs before deployment.

Finding 5: Language features used most differently are gradual typing (used in only 2 domains), exception handling statements (the maximum and average usage rates are 16.36% and 6.74%), nested classes (the maximum and average usage rates are 25.60% and 14.16%), decorators (the maximum and average usage rates are 25.68 and 17.14%). Such differences always reveal extensive using patterns and deserve in-depth analysis.

IV. In-depth Analysis and Discussion

In this subsection, we focus on answering RQ3 and RQ4.

RQ 3: Why are certain language features used frequently and how are they used?

In the RQ2 study, we find that *decorators*, *exception handling statements*, and *nested classes* are used most differently, thus deserving in-depth analysis. We next discuss these three language features and try to find why and how they are used frequently and differently in the following three subsections.

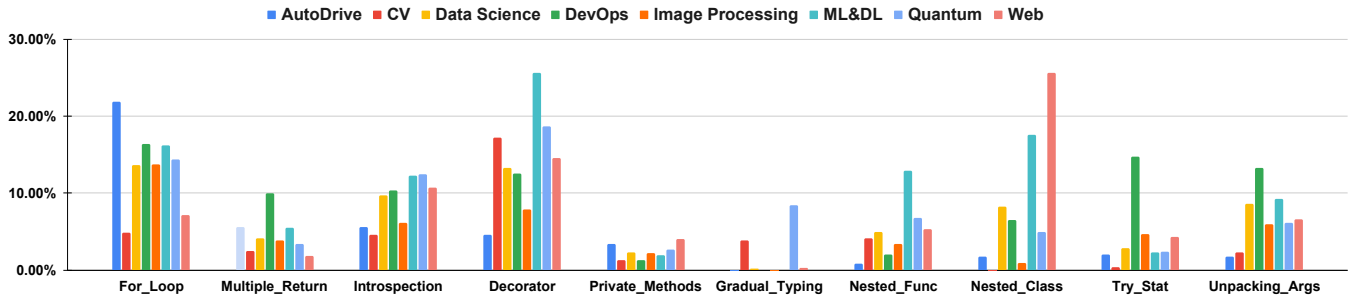

Fig. 3. Language feature usage of Python projects from eight domains

TABLE III
USAGE INFORMATION OF BUILT-IN AND USER DEFINED DECORATORS

Built-in			User-defined
@staticmethod	@classmethod	@property	46019 (**74.66%**)
2259 (14.46%)	1903 (12.18%)	11456 (73.36%)	
15620 (**25.34%**)			

A. Decorators

As a way to modify the behavior of function or class, decorators allow users to wrap another function to extend the behavior of wrapped function without permanently modifying it. To understand the use of decorators in real-world projects, we further collect the usage information of three built-in decorators [1] and user-defined decorators, as shown in Table III. The @classmethod and @staticmethod decorators are used to define methods inside a class namespace that are not connected to a particular instance of that class. The @property decorator is used to customize getters and setters for class attributes.

> **Finding 6:** Developers prefer to define their own decorators instead of using built-in decorators, and the former is about 3× the latter. And the use of @property accounts for 3/4 of the total number of built-in decorators.

We then conduct a manual check to see how the user-defined decorator is used in the real-world Python projects. Code 1 shows seven typical using patterns (UP-D1∼UP-D7) of user-defined decorators we find.

Decorators bring much convenience to the *testing* process, such as setting up the testing environment (UP-D1), skipping the test unless the module has certain features (UP-D2), and setting input arguments for testing (UP-D3). Decorators can also label the *deprecated* function (UP-D4), not only to remind users to change to the new version, but also to maintain backward compatibility. Decorators like UP-D5 can realize *overloading*, so that functions such as equal support not only regular NumPy arrays but also other arrays derived from them. This usage significantly reduces the code size and greatly improves the compatibility while almost not affecting performance [26]. Decorators like UP-D6 can filter and convert different types of function arguments to the same type, *e.g.,* string. Decorators like UP-D7 can control the compilation strategy of Python code, *e.g.,* functions annotated with @torch.jit.script_method will be translated into TorchScript [42] for optimization.

```
# UP-D1 from Django v3.0.4
@setup({'if-tag01': '{% if foo %}yes{% else %}no{% endif %}'})
    def test_if_tag01(self):
        ...
# UP-D2 from Django v3.0.4
@skipUnlessDBFeature('can_create_inline_fk')
    def test_inline_fk(self):
        ...
# UP-D3 from Pandas v1.0.3
@pytest.mark.parametrize("cache", [True, False])
    def test_to_datetime_dt64s(self, cache):
        ...
# UP-D4 from Tensorflow v2.2.0-rc3
@deprecation.deprecated(
    "2016-12-30",
    "`tf.mul(x, y)` is deprecated; use `tf.math.multiply(x, y)` or `x * y`")
def _mul(x, y, name=None):
    ...
# UP-D5 from Numpy v1.8.3
@array_function_dispatch(_binary_op_dispatcher)
def equal(x1, x2):
    ...
# UP-D6 from Django v3.0.4
@stringfilter
def addslashes(value):
    ...
# UP-D7 from Pytorch v1.5.0
@torch.jit.script_method
def forward(self, input):
```

Code. 1. Typical using patterns of decorators

B. Exception Handling

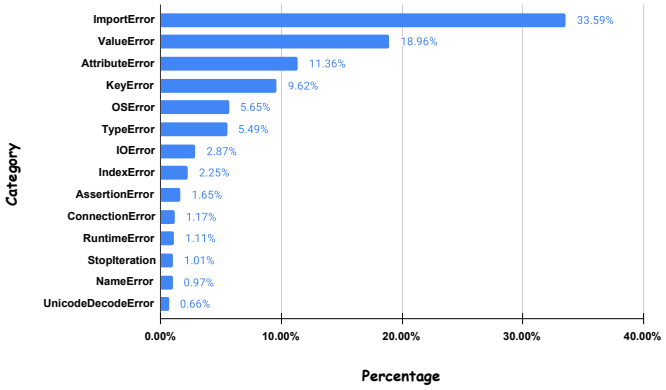

Fig. 4. Standard errors exceptions raised in real-world Python projects

For exception handling, in order to understand what errors developers care most, we further collect the standard errors raised by exception handling statements from the evaluated projects. The results are shown in Fig. 4.

Finding 7: Developers care most about ImportError, ValueError, AttributeError, KeyError and OSError, which totally account for 80% of all errors.

In Code 2 we show five typical using patterns of exception handling statements, *i.e.,* UP-E1∼UP-E5.

```
# UP-E1 from Ansible v2.9.7
try:
    basestring
except NameError:
    basestring = string_types
# UP-E2 from Ansible v2.9.7
resp = self.client.api.get(uri)
try:
    response = resp.json()
except ValueError as ex:
    raise F5ModuleError(str(ex))
# UP-E3 from Ansible v2.9.7
try:
    from ansible.module_utils.common._json_compat import json
except ImportError as e:
    print('\n{{"msg": "Error: ansible requires the stdlib json: {0}", "failed":
        true}}'.format(to_native(e)))
    sys.exit(1)
# UP-E4 from Ansible v2.9.7
try:
    return int(self._values['priority_to_client'])
except ValueError:
    return self._values['priority_to_client']
# UP-E5 from Ansible v2.9.7
try:
    return check_type_str(value, allow_conversion)
except TypeError:
    common_msg = 'quote the entire value to ensure it does not change.'
```

Code. 2. Typical using patterns of exception handling statements

The first three patterns handle errors caused by differences between Python versions (UP-E1), interaction with other devices or services (UP-E2), and module importing (UP-E3). Such usage provides a good way to improve the compatibility and adaptability of Python programs when changing in Python versions, devices or services, modules, etc. Python is popular for its rich modules. A large project always has many submodules with different functions, and the function in each submodule relies on other submodules. The interfaces between modules keep changing over time, causing the most ImportError errors. The last two patterns are both about types, which deal with errors when converting one type to another (UP-E4), and checking the type of certain variables (UP-E5).

C. Nested Classes and Nested Functions

```
# UP-N1 from Django v3.0.4
class ModelFormBaseTest(TestCase):
    def test_no_model_class(self):
        class NoModelModelForm(forms.ModelForm):
            pass
        with self.assertRaisesMessage(ValueError, 'ModelForm has no model class
        specified.'):
            NoModelModelForm()
# UP-N2 from Pyquil v2.22.0
class QuilParser ( Parser ):
        class QuilContext(ParserRuleContext):
        def quil(self):
        class AllInstrContext(ParserRuleContext):
        def allInstr(self):
        ...
# UP-N3 from Django v3.0.4
class ModelFormMetaclass(DeclarativeFieldsMetaclass):
    def __new__(mcs, name, bases, attrs):
    ...
class PriceForm(forms.ModelForm):
    class Meta:
        model = Price
        fields = '__all__'
# UP-N4 from Pillow v7.1.2
def load_signed_rational(self, data, legacy_api=True):
        ...
        def combine(a, b):
            return (a, b) if legacy_api else IFDRational(a, b)
    return tuple(combine(num, denom) for num, denom in zip(vals[::2], vals
        [1::2]))
```

Code. 3. Typical using patterns of nested classes or functions

Code 3 shows four typical using patterns of nested classes and functions. By manually checking the source code, we find that test files use the most nested classes (UP-N1), where the code to be tested is placed in a single class. Actually nested functions also have similar usage. This can not only hide the test details from outside but also make testing more convenient. Another common pattern for nested classes is to implement different components of a module, for example in pattern UP-N2, each part of the parser is implemented as a nested class. Developers also use nested class to customize the derived class from a metaclass. They define a small inner class with metadata to guide the generation of new classes from a metaclass as in UP-N3. The last pattern UP-N4 is used frequently for nested functions, which occurs when developers want to hide a repeated operation inside a function. **RQ 4**: How does this empirical study help improve the design and quality of Python tasks?

In this RQ study, we hope to provide certain user groups with some implications and advice based on the results of the first three RQs. We divide users into 3 groups: Python designers, Python compiler designers, Python application developers.

For **Python designers,** we greatly appreciate their work for bringing such a powerful programming language to the world. Python has been widely used in many domains. It evolves fast and each version brings some new features, for example, the position-only parameter is introduced in Python 3.8 to tackle certain problems raised by Python community. However, in our study we find that some new features have not been widely adopted by developers. The typical one is *gradual typing*. Gradual typing allows users to annotate only part of a program to statically type and enhance the type safety. Lack of human type annotations as ground truth also hinders some use of deep learning techniques to help improve type safety. We think a possible reason may be that adding type annotations is not easy in Python and greatly enlarges code size. A similar case happens on *keyword-only parameter*, which was introduced in Python3, but has not been supported by some Python compilers even now. Therefore, Python designers can pay more attention to simplifying the use of features which truly brings benefits to programs.

For **Python compiler designers** including program analysis designers, they have proposed different compilers or analyzers for different goals such as performance optimization and bug finding. However, such compilers and analyzers only support a subset of Python, so how to choose language features becomes crucial. In RQ1 study, we find that developers focus on relatively simple language features on safety checks, testing and some dynamic features. We suggest Python compiler/analyzer designers focus more on these features, rather than complex features which easily cause errors such as heterogeneous lists and diamond inheritance. We hope the general distribution of language feature usage we show in RQ1 can help the design of some Python compilers.

For **Python application developers**, we list some typical using patterns of common language features in RQ3. These patterns are not trivial and can improve the safety, compati-

bility and performance of Python programs. Thus we highly recommend the application developers to adopt these patterns.

V. Threats to Validity

Our study mainly suffers from three kinds of threats:

Threats to dataset. We build our dataset by choosing 35 popular Python projects. We try to make our dataset more representative by selecting the most influential projects in Python community. However, these projects may not contain all typical language feature patterns. We mitigate this problem by choosing the most popular and influential Python projects.

Threats to type inference. We use *PySonar2* as our type inference tool since it infers types quickly with relatively high accuracy. However, there are some functions or variables that *PySonar2* fails to infer, which might affect the recognition of language features such as polymorphism. We make a trade-off between the accuracy and time consumption of inference techniques in this study. However, PySCAN can be improved by replacing *PySonar2* with more advanced in the future.

Threats to language feature identification. There exists some dynamic features in Python projects which can hardly be identified in static analysis. We mitigate this problem by identifying typical function calls with these features. Although this can catch most dynamic features, we may still miss some complex implementations of them.

VI. Related Works

A. Analysis of Language Features

In recent years, researchers have conducted several studies on certain language features of Python. Åkerblom *et al.* study dynamic features [34] and polymorphism [35] in Python programs based on traces of run-time data. Lin *et al.* study fine-grained source code changes of Python software by developing an automatic change extraction tool PyCT [29]. Vitousek *et al.* present Reticulated Python, a system for experimenting with gradual-typed dialects of Python. These dialects are syntactically identical to Python 3, but provide static and dynamic semantics for type annotations that already exist in Python 3. Malloy *et al.* develop a Python compliance analyser, PyComply, and use it to measure and quantify the degree to which Python developers were making the transition from Python 2 to Python 3 [30, 31]. Biswas *et al.* create a dataset to enable MSR (mining software repositories) research on *Data Science* programs written in Python [6]. They extract such projects from GitHub, and then map their Python AST to Boa AST, and reuse Boa infrastructure [13] to generate datasets and provide public queries.

Apart from Python, language features of other programming languages also get focused. Dyer *et al.* analyze a large number of open source Java projects to study the usage of Java language features [14]. There are also some studies separately analyzing overloading features in both Java [19] and C++ programs [48]. Rodrigues *et al.* investigate how developers use dynamic features based on 28 open-source Ruby projects [36]. Dilley *et al.* analyze how massage passing concurrency is used in Go projects from GitHub [10].

B. Type Inference

As a special case of language feature analysis in Python, type inference of Python becomes a hot topic recently. For Python type inference tools, *PySonar2* is a type inference tool in Python implemented by Yin Wang [49]. It can infer 49.47% types of variables in real-world Python programs according to the experiment conducted by Xu *et al.* [53]. Google proposes a type checker *Pytype*, which firstly infers the types of objects in Python source files and then conducts a type check [21]. *Pyre* proposed by Facebook also has an independent inference model for users to statically infer types [15].

For research focusing on type inference, Xu *et al.* observe that type hints in Python programs can help infer types of variables and propose a novel approach of probabilistic type inference [53]. Gorbovitski *et al.* propose a flow-sensitive may-alias analysis based on type inference to optimize the execution of Python programs [22]. Cannon *et al.* implement a type inference algorithm for Python without changing the semantics of it and study the benefit of adding type annotations to help type inference [7]. Aycock *et al.* propose aggressive type inference, which determine the types of variables in the absence of explicit cues [5]. Pradel *et al.* implement a neural network model with a search based validation to do type inference on Python projects [33]. Allamanis *et al.* address the out of vocabulary problem of type inference tasks and implement a graph model to further improve the accuracy [3]. Dash *et al.* introduce conceptual types which developers have in mind while writing the program. Variables with different conceptual types may share the same regular type. For example, email and url types share string type [9].

VII. Conclusion

In this paper, we conduct an empirical study of language feature usage on 35 Python projects from 8 domains by designing a tool named PySCAN to automatically scan language features. We observe the following key findings: 1) Developers tend to use language features on safety check, testing and some dynamic features while avoiding the use of complex features which easily lead to errors. 2) Some language features designed to enhance the safety of Python such as gradual typing and keyword-only arguments have not been widely adopted by developers. 3) Developers prefer to define their own decorators instead of using built-in decorators. 4) Developers care most about ImportError errors when handling exceptions. Along with these findings we manually check using patterns of exception handling, decorators and nested classes / functions. We believe such findings and patterns can help Python users better design applications.

Acknowledgment

This work was partially funded by the National Natural Science Foundation of China (No. 61772487), Anhui Provincial Natural Science Foundation (No. 1808085MF198) and Anhui Provincial Development and Reform Commission 2020 New Energy Vehicle Industry Innovation Development Project "Key System Research and Vehicle Development for Mass Production Oriented Highly Autonomous Driving".

References

[1] The Python standard library: Built-in functions. https://docs.python.org/3/library/functions.html (Accessed: Oct 2020).

[2] Harold Abelson and Gerald Jay Sussman. Formulating abstractions with higher-order procedures. In Structure and Interpretation of Computer Programs. MIT Press, 1984.

[3] Miltiadis Allamanis, Earl T. Barr, Soline Ducousso, and Zheng Gao. Typilus: Neural type hints. In 41st PLDI, page 91–105. Association for Computing Machinery, 2020.

[4] Anaconda. Numba, 2019. http://numba.pydata.org/.

[5] J. Aycock. Aggressive type inference. In International Python Conference, 2000.

[6] Sumon Biswas, Md Johirul Islam, Yijia Huang, and Hridesh Rajan. Boa meets Python: A Boa dataset of data science software in Python language. In 16th MSR, page 577–581. IEEE Press, 2019.

[7] B. Cannon. Localized type inference of atomic types in Python. In Master's Thesis, 2005.

[8] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. In ACM Computing Surveys, 1985.

[9] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. RefiNym: Using names to refine types. In 26th ESEC/FSE, page 107–117. Association for Computing Machinery, 2018.

[10] N. Dilley and J. Lange. An empirical study of messaging passing concurrency in Go projects. In 26th SANER, pages 377–387, 2019.

[11] Stephanie Douglas, Adrian Price-Whelan Stuart Mumford, Nathan Goldbaum, Tom Robitaille, and Erik Tollerud. Python3 advanced features, 2016. https://python-3-for-scientists.readthedocs.io/en/latest/python3_advanced.html.

[12] Michael Driscoll. Welcome to Python 101! Chapter 12 Introspection, 2020. https://python101.pythonlibrary.org/chapter12_introspection.html.

[13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In 35th ICSE, page 422–431. IEEE Press, 2013.

[14] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of Java language features. In 36th ICSE, page 779–790, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Facebook. Pyre: Performant type checker for Python.

[16] Python Software Foundation. Brief introduction to metaclass, 2020. https://docs.python.org/3/reference/datamodel.html.

[17] Python Software Foundation. List comprehension, 2020. https://docs.python.org/3/tutorial/datastructures.html.

[18] Python Software Foundation. Official documentation of Python3, 2020. https://docs.python.org/3.

[19] Joseph (Yossi) Gil and Keren Lenz. The use of overloading in Java programs. In Theo D'Hondt, editor, ECOOP 2010 – Object-Oriented Programming, pages 529–551, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] Github. Github Annual Report 2019: The State of the Octoverse, 2019. https://octoverse.github.com/.

[21] Google. Pytype: Static type analyzer for Python code.

[22] Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. SIGPLAN Not., 45(12):27–42, October 2010.

[23] Ronald Graham, Donald Knuth, and Oren Patashnik. Recurrent problems. In Concrete Mathematics, 1990.

[24] Larry Hastings, Pablo Galindo, Mario Corchero, and Eric N. Vander Weele. Python PEP 570 - Python positional-only parameters, 1 2018. https://www.python.org/dev/peps/pep-0570/.

[25] Raymond Hettinger. What's new in Python 3.8, 2019. https://docs.python.org/3/whatsnew/3.8.html.

[26] Stephan Hoyer, Matthew Rocklin, Marten van Kerkwijk, and Hameer Abbasi. Numpy NEP 18: A dispatch mechanism for NumPy's high level array functions, 2019. https://numpy.org/neps/nep-0018-array-function-protocol.html.

[27] Jetbrains. Only 13 percent of developers keep using Python2 in 2019, 2019. https://www.jetbrains.com/lp/devecosystem-2019/python/.

[28] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. Numba: A LLVM-based Python JIT compiler. In 2nd LLVM, pages 7:1–7:6, 2015.

[29] Wei Lin, Zhifei Chen, Wanwangying Ma, Lin Chen, Lei Xu, and Baowen Xu. An empirical study on the characteristics of Python fine-grained source code change types. In 32nd ICSME, pages 188–199, 2016.

[30] Brian A. Malloy and James F. Power. Quantifying the transition from Python 2 to 3: An empirical study of Python applications. In 11th ESEM, pages 314–323, 2017.

[31] Brian A. Malloy and James F. Power. An empirical analysis of the transition from Python 2 to Python 3. In Empirical Software Engineering, volume 24, pages 751–778, 2019.

[32] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In 32th NeurIPS, pages 8024–8035. Curran Associates, Inc., 2019.

[33] Michael Pradel, Georgios Gousios, Jason Liu, and Satish Chandra. TypeWriter: Neural type prediction with search-based validation. In 2020 ESEC/FSE, pages 241–250, 2020.

[34] Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. Tracing dynamic features in Python programs. In 11th MSR, page 292–295, New York, NY, USA, 2014. Association for Computing Machinery.

[35] Beatrice Åkerblom and Tobias Wrigstad. Measuring polymorphism in Python programs. In 11th DLS, page 114–128, New York, NY, USA, 2015. Association for Computing Machinery.

[36] Elder Rodrigues and Ricardo Terra. How do developers use dynamic features? The case of Ruby. Computer Languages, Systems & Structures, 53:73 – 89, 2018.

[37] Wm. Paul Rogers. Encapsulation is not information hiding, May 2001. https://www.javaworld.com/article/2075271/encapsulation-is-not-information-hiding.html.

[38] Guido van Rossum and Ivan Levkivskyi. Python PEP 483, 2014. https://www.python.org/dev/peps/pep-0483/.

[39] Nicholas Samuel. Python nested functions, 2020. https://stackabuse.com/python-nested-functions/.

[40] Hafeezul Kareem Shaik. Inner classes in Python, 2020. https://www.datacamp.com/community/tutorials/inner-classes-python.

[41] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In Scheme and Functional Programming Workshop, pages 81–92, 2006.

[42] PyTorch software. TorchScript. https://pytorch.org/docs/stable/jit.html (Accessed: 2020).

[43] Talin. Python PEP 3102 - keyword-only arguments, April 2006. https://www.python.org/dev/peps/pep-3102/.

[44] Laurence Tratt. Dynamically typed languages. Advances in Computers, 77:149–184, July 2009.

[45] Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. PEP 484 – Type Hints, 2014.

[46] Guido van Rossum, Barry Warsaw, and Nick Coghlan. Python PEP 8 – style guide for Python code, 8 2013. https://www.python.org/dev/peps/pep-0008/, created in July 2001.

[47] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. page 45–56, 2014.

[48] C. Wang and D. Hou. An empirical study of function overloading in C++. In 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pages 47–56, 2008.

[49] Yin Wang. PySonar2: An advanced semantic indexer for Python. https://github.com/yinwang0/pysonar2.

[50] Collin Winter. Python PEP 318 - decorators for functions and methods, June 2003. https://www.python.org/dev/peps/pep-318/.

[51] Collin Winter. Python PEP 3129 - class decorators, 5 2007. https://www.python.org/dev/peps/pep-3129/.

[52] Collin Winter and Tony Lownds. Python PEP 3107 - function annotations, December 2006. https://www.python.org/dev/peps/pep-3107/.

[53] Zhaogui Xu, Xiangyu Zhang, Lin Chen, Kexin Pei, and Baowen Xu. Python probabilistic type inference with natural language support. In 24th FSE, pages 607–618, 2016.