中国科学技术大学
University of Science and Technology of China

An Empirical Study for Common Language Features Used in Python Projects

Yun Peng, Yu Zhang*, Mingzhe Hu

University of Science and Technology of China

Background

- **Python is extremely popular in recent years**
  - Dynamic type system → fast prototyping
  - Lots of libraries and powerful language features
- **However, dynamic features have costs:**
  - Performance: interpreter → longer execution time
  - Safety: dynamic type system → type errors

python™

Solutions

- **Pysonar2,** a Python type inference tool

```
def train_nlu(args: argparse.Namespace, train_path: Optional[Text] = None
              ) -> Optional["Interpreter"]:
    frc (?, ?) -> None / (?, None) -> None n_nlu

    output = train_path or args.out

    config = get_validated_path(args.config, "config", DEFAULT_CONFIG_PATH)
    nlu_data = get_validated_path(args.nlu, "nlu", DEFAULT_DATA_PATH)

    return train_nlu(config, nlu_data, output, train_path)
```

Some dynamic features (e.g. meta-programming) make it hard for Pysonar2 to statically infer types

- **Numba,** a JIT Python compiler aims to accelerate the execution of Python code

```
from numba import jit
import random

@jit(nopython=True)
def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples
```

It only supports a subset of Python and many dynamic features are excluded (e.g. function as return value)

Motivation

- Current solutions to improve Python's performance and safety often encounter new problems or challenges for certain language features

Question:

- **How are language features distributed in real-world Python projects?**
  - We may not pay many efforts to handle them if they are rarely used...
  - If they are commonly used, how are they used?

Research Questions

中国科学技术大学
University of Science and Technology of China

- **RQ1:** What is the general distribution of language features in real-world Python projects?
- **RQ2:** What are the differences of language feature distribution among different domains of Python projects?
- **RQ3:** Why are certain language features used frequently and how are they used?

Language Features

6 categories of 22 language features:

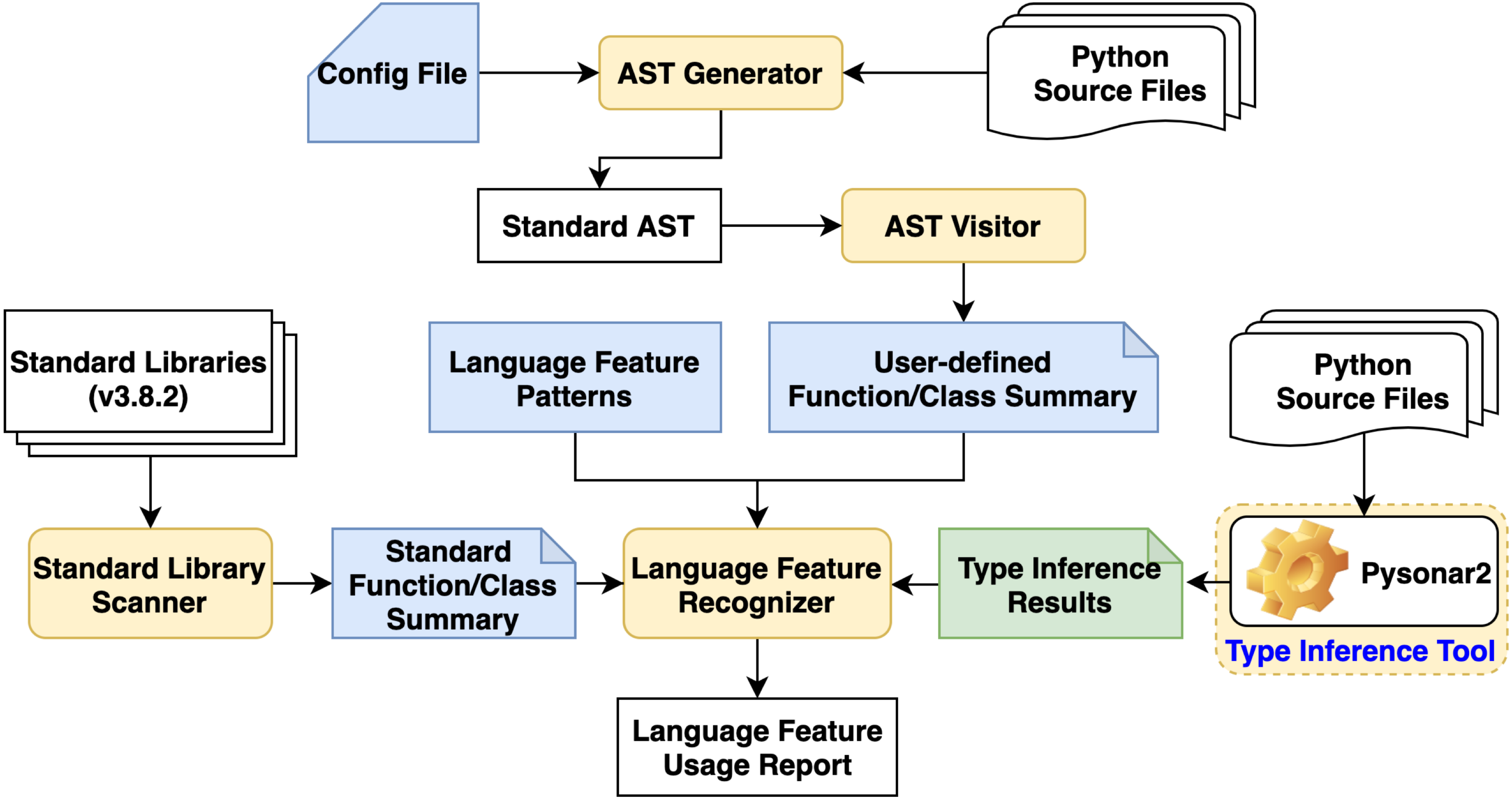
- **Function:** keyword/keyword-only/position-only arguments, multiple return, etc.
- **Type System:** first-class function, gradual typing, etc.
- **Loop & Evaluation Strategy:** loop, generator, etc.
- **Object-Oriented Programming:** inheritance, encapsulation, etc.
- **Data Structure:** list comprehension, heterogeneous list, etc.
- **Metaprogramming:** introspection, reflection, etc.

Language Features Recognition

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*

PYSCAN

A Python language feature scanner



DataSet

- 35 popular Python projects from 8 domains at GitHub
- 4 million lines of code and 25 thousand files

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**		

RQ1: General distribution

中国科学技术大学
University of Science and Technology of China

Top 5 used:

- Single inheritance
- Decorator
- Keyword argument
- For loop
- Nested class

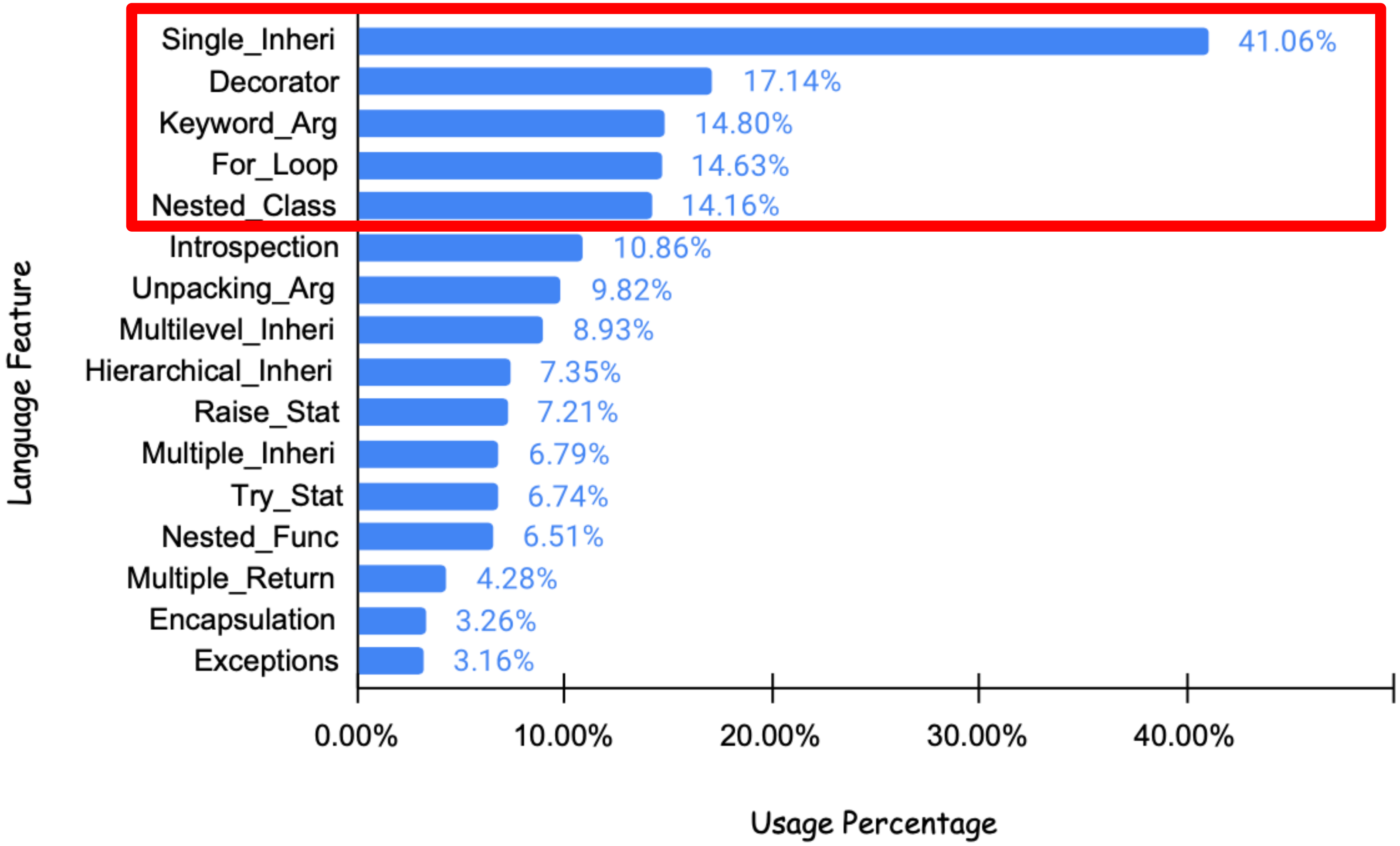


Fig. 2. General distribution of language feature usage in 35 Python projects

RQ1: General distribution

中国科学技术大学 University of Science and Technology of China

Least 5 used:

- Position-only argument (0%)
- Heterogeneous list (0.05%)
- Heterogeneous tuple (0.05%)
- Keyword-only argument (0.14%)
- Function as variables (0.26%)

Finding:

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.

RQ1: General distribution

2 Special features:

- Gradual typing (0.6%)
  - Aims to enhance type safety
- Keyword-only arguments (0.14%)
  - Aims to avoid misuse caused by rapid API changes

Finding:

Some language features designed to enhance the safety of Python programs are not widely used in real-world Python projects.

RQ2: Distribution in domains

中国科学技术大学 University of Science and Technology of China

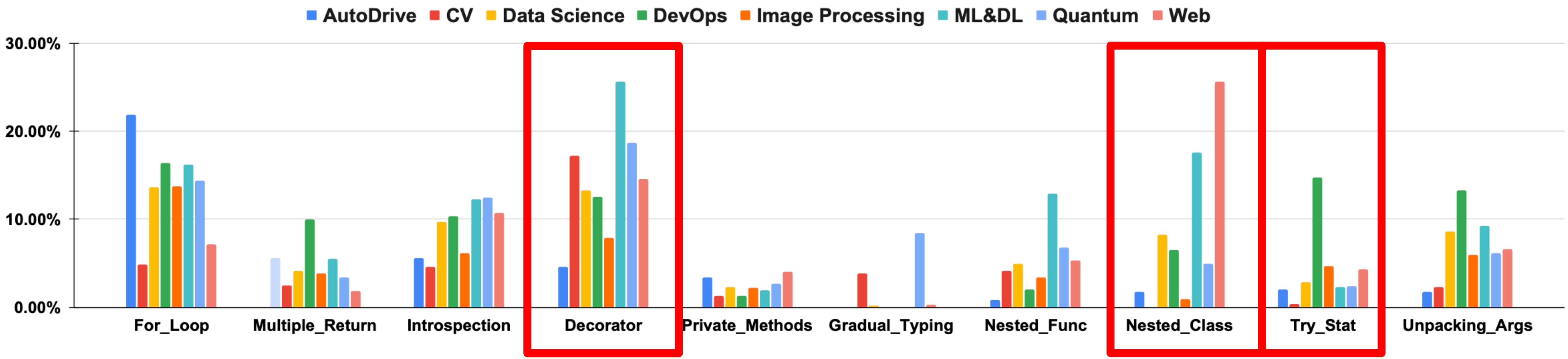


Fig. 3. Language feature usage of Python projects from eight domains

Used frequently but differently across domains:

Decorator (min: 17.14% → max: 25.68%)

Nested class (min: 14.16% → max: 25.60%)

Exception handling statements (min: 6.74% → max: 16.36%)

RQ3: In-depth analysis

❑ Decorator

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%**)			

Finding:

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

RQ3: In-depth analysis

❑ Decorator

```
# 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):
```

UP-D1: Set up testing environment

UP-D2: Skip feature in testing

UP-D3: Set inputs for tests

UP-D4: Label deprecated functions

UP-D5: Realize overloading

UP-D6: Convert the types of arguments

UP-D7: Control compilation strategy

RQ3: In-depth analysis

University of Science and Technology of China

❑ Exception handling statements

Top 5 Used Errors:

(80% of total usage)

- ImportError
- ValueError
- AttributeError
- KeyError
- OSError

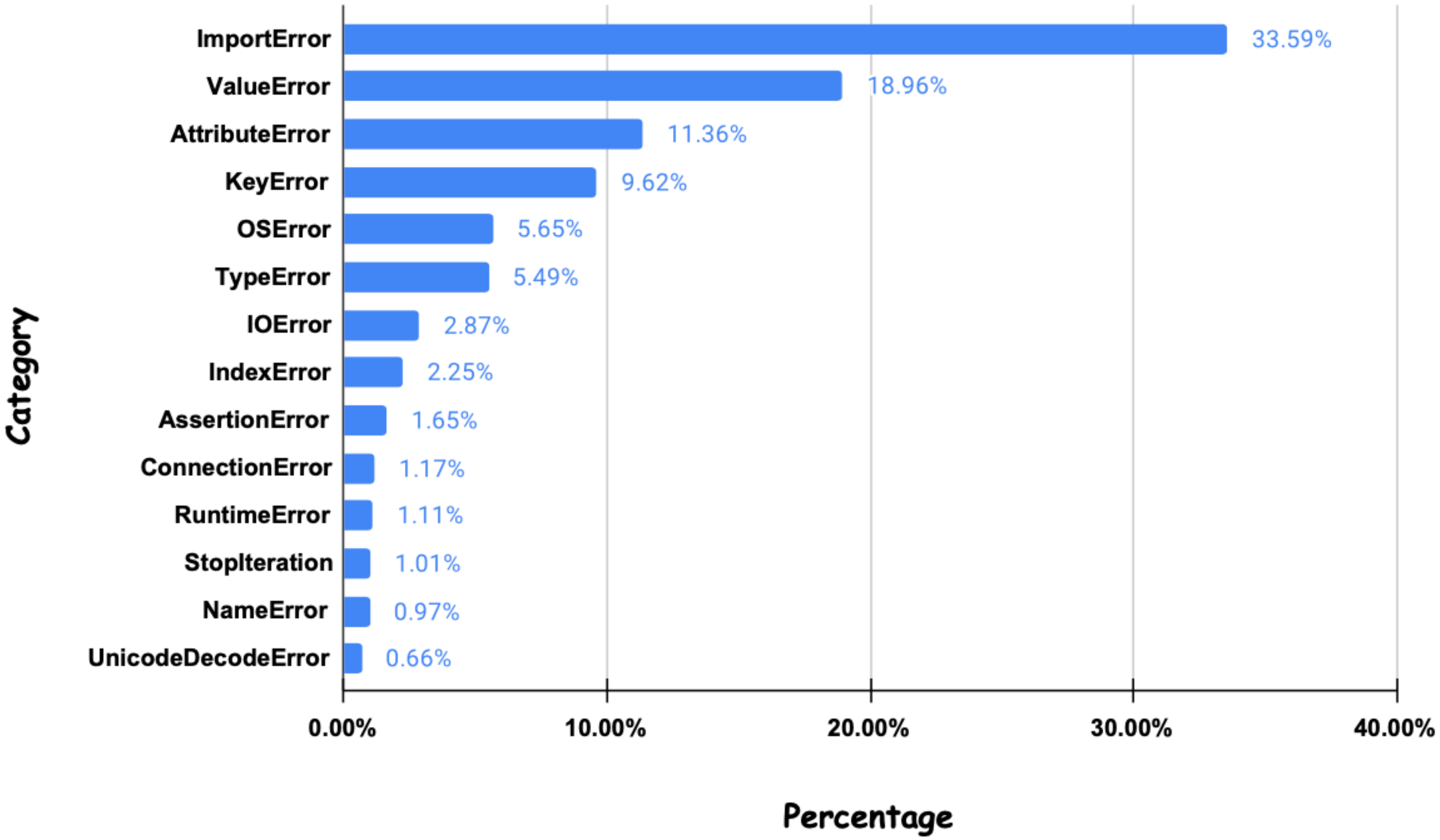


Fig. 4. Standard errors exceptions raised in real-world Python projects

RQ3: In-depth analysis

❑ Exception handling statements

```
# 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.'
```

UP-E1: Differences between Python versions

UP-E2: Interaction with other modules or devices

UP-E3: Module importing

UP-E4: Type Conversion

UP-E5: Type Check

RQ3: In-depth analysis

中国科学技术大学
University of Science and Technology of China

Nested Class/Function

```
# 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]))
```

UP-N1: Test a certain module

UP-N2: Implement different parts of a module

UP-N3: Define metadata of a class

UP-N4: Define frequently used inner operations

Conclusion

中国科学技术大学 University of Science and Technology of China

- We summarize 22 kinds of common language features, which are divided into 6 categories.
  - An automatic language feature scanner named PYSCAN
  - Analysis of their general distributions, specific distributions across different domains
  - In-depth analysis on exception handling statements, decorators and nested classes/functions

- We conclude some implications and findings for developers and researchers targeting Python from the empirical results

Thanks

Q&A