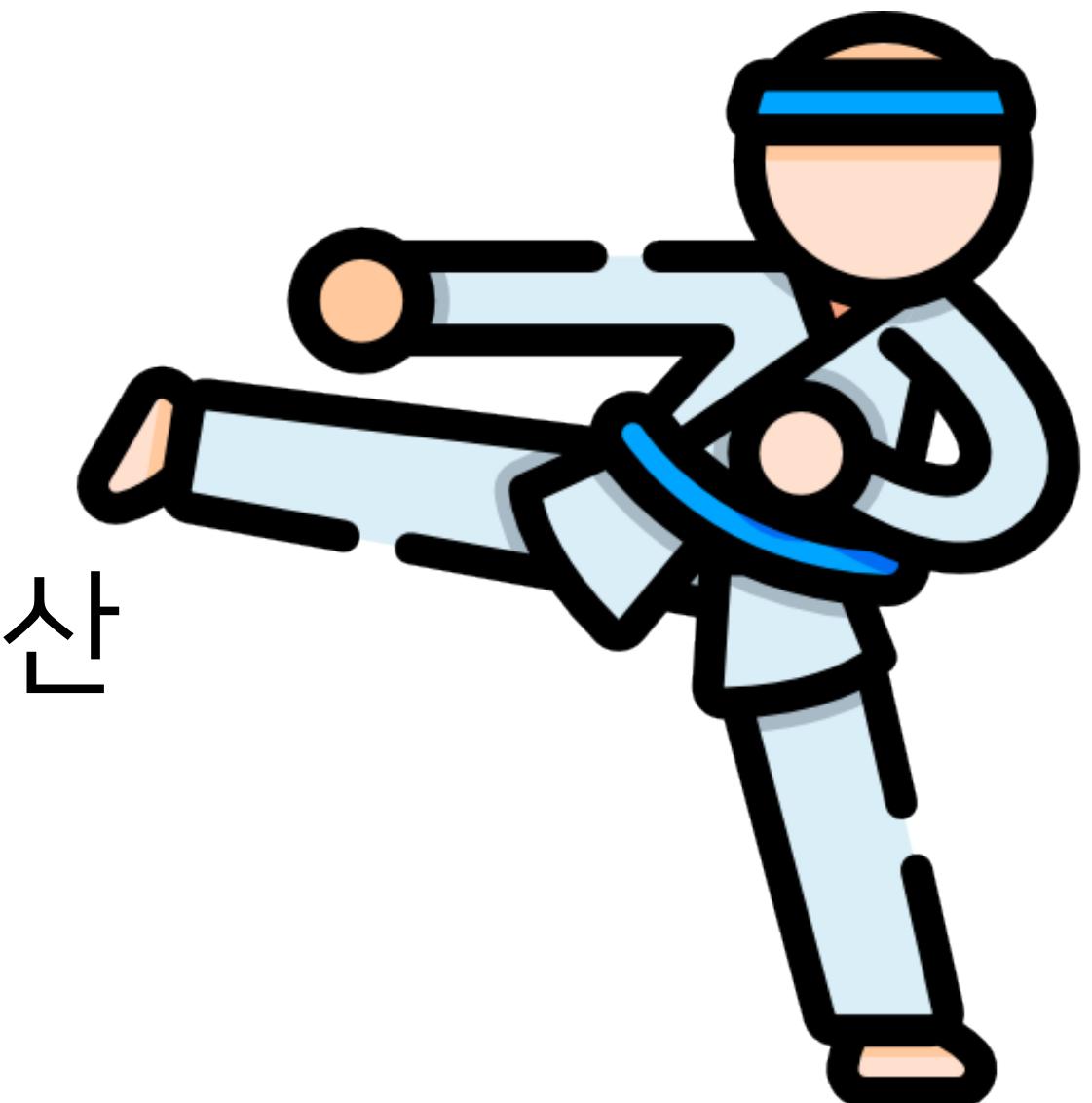


PyTER: 파이썬 타입 오류 자동 수정 기술

오원석, 오학주
고려대학교

06 Jul 2023

ERC 23 여름 워크샵 @ 머큐어 앰배서더, 울산



KOREA
UNIVERSITY

들어가기 전에

- 연구2그룹: 런타임 SW재난 유발 오류 신속 대응



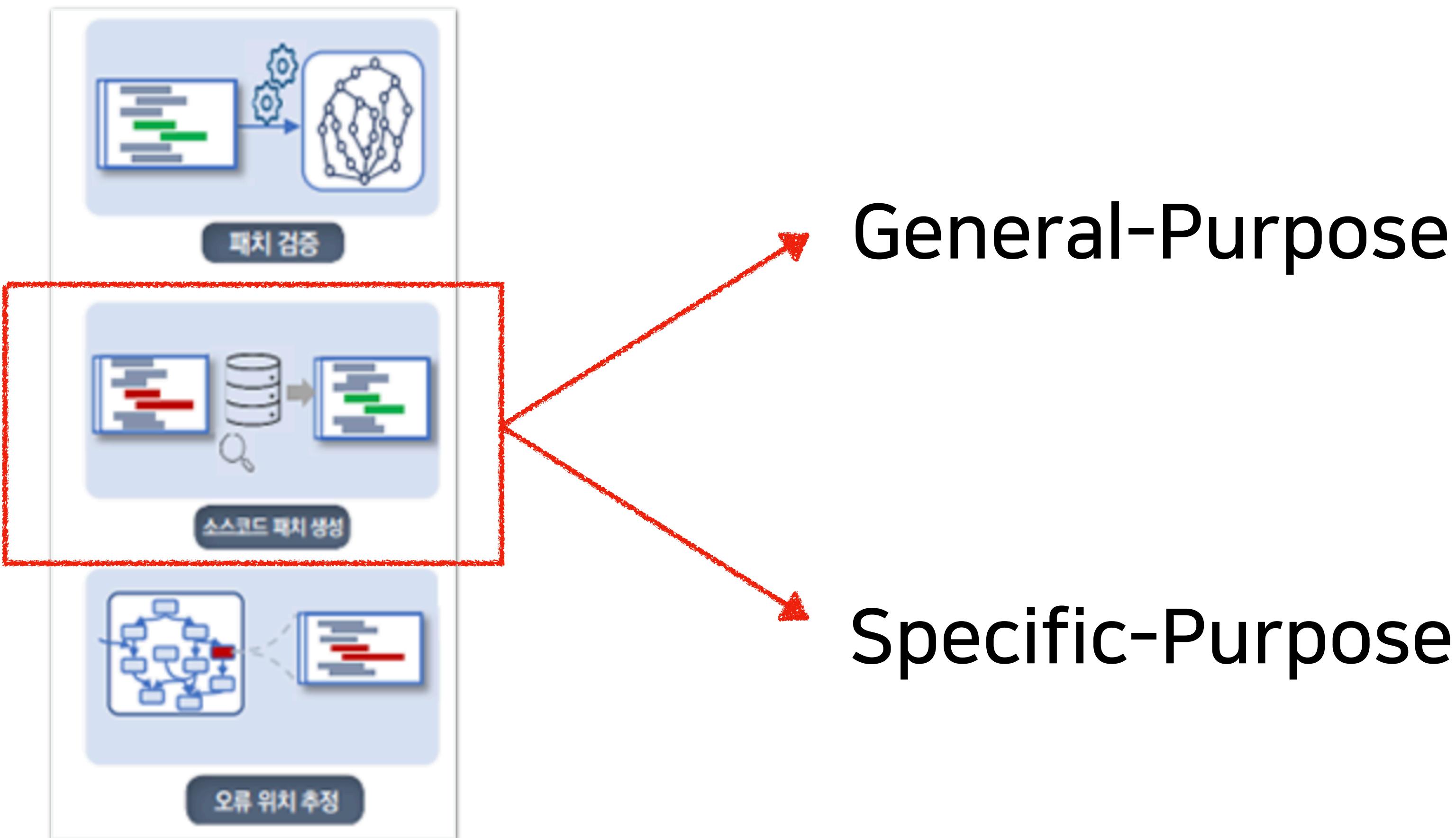
들어가기 전에

- 연구2그룹: 런타임 SW재난 유발 오류 신속 대응



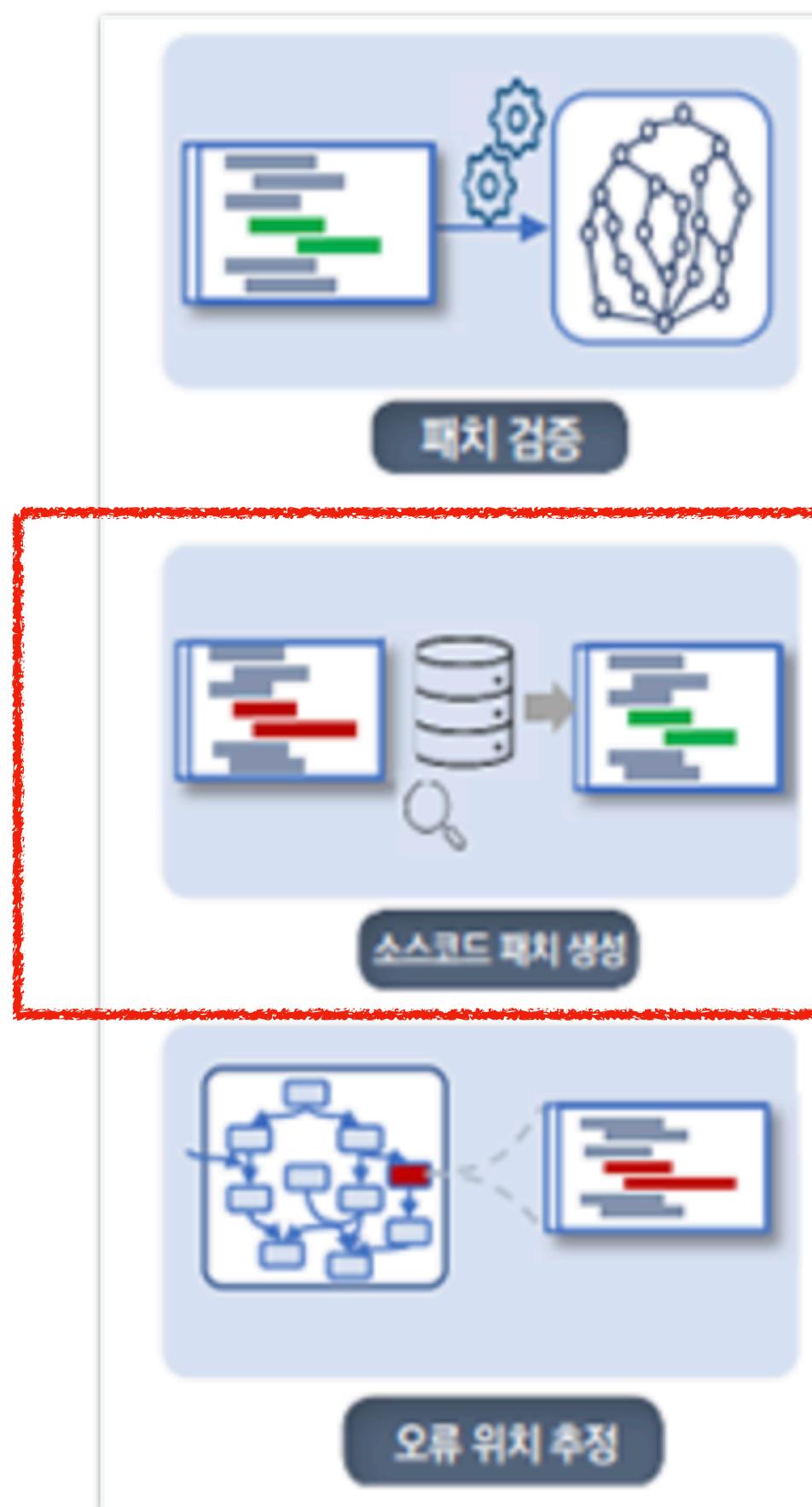
들어가기 전에

- 연구2그룹: 런타임 SW재난 유발 오류 신속 대응



들어가기 전에

- 연구2그룹: 런타임 SW재난 유발 오류 신속 대응

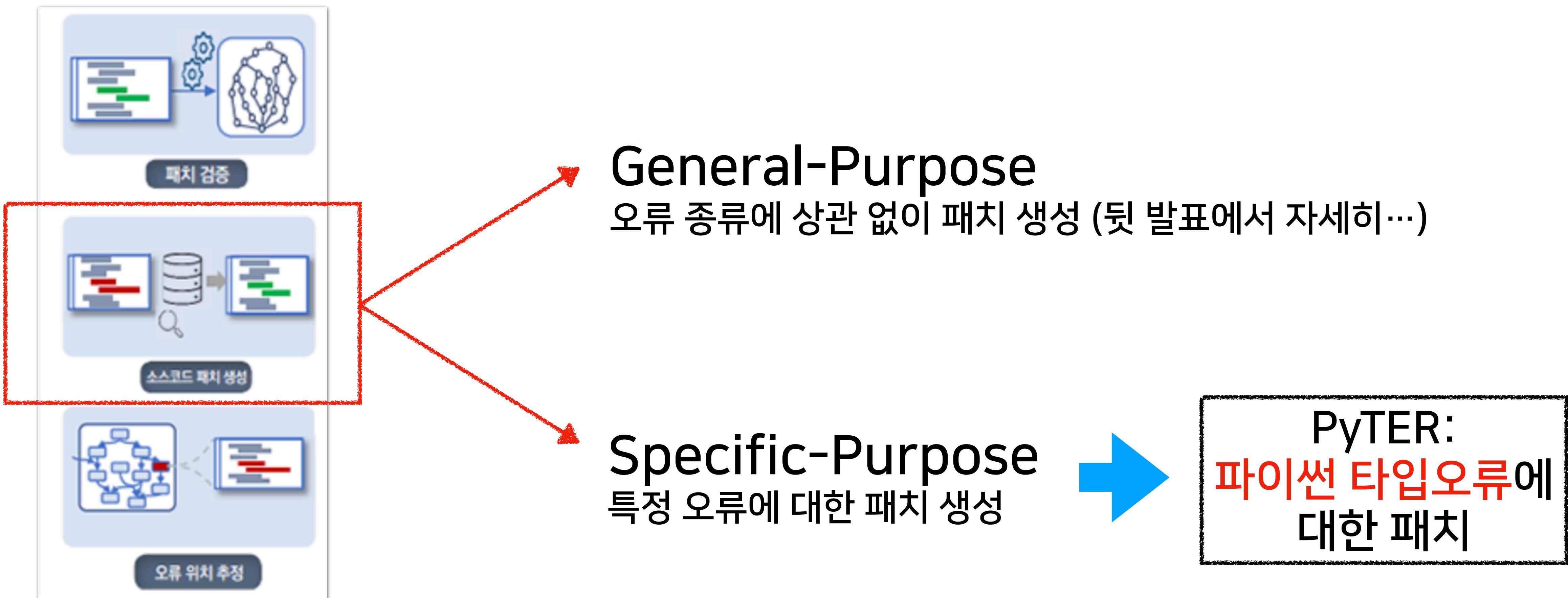


General-Purpose
오류 종류에 상관 없이 패치 생성 (뒷 발표에서 자세히…)

Specific-Purpose
특정 오류에 대한 패치 생성

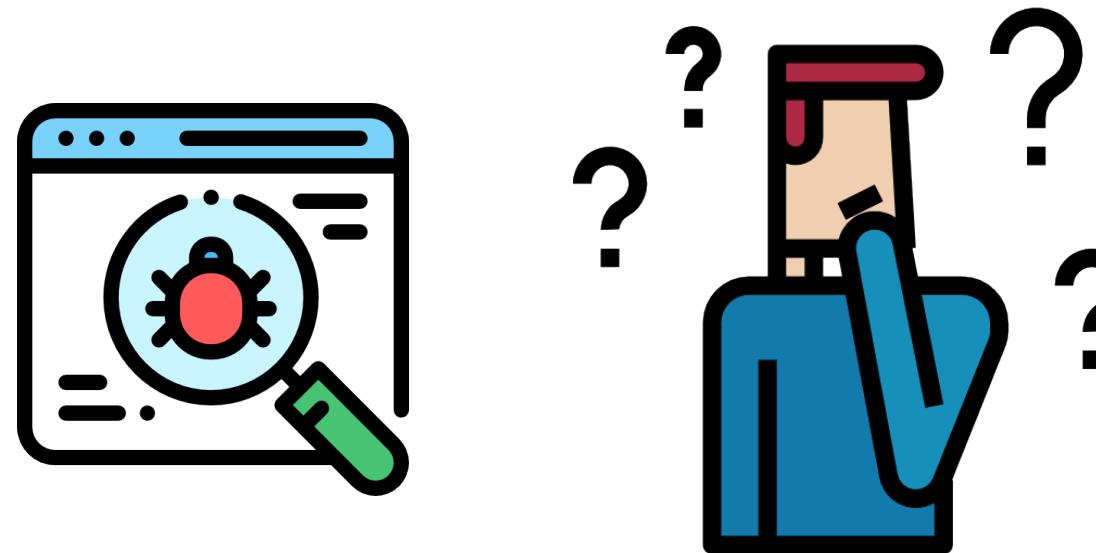
들어가기 전에

- 연구2그룹: 런타임 SW재난 유발 오류 신속 대응



동기: 타입오류는 중요한 문제이다

1. 파이썬에서 가장 흔한 오류는 무엇일까?

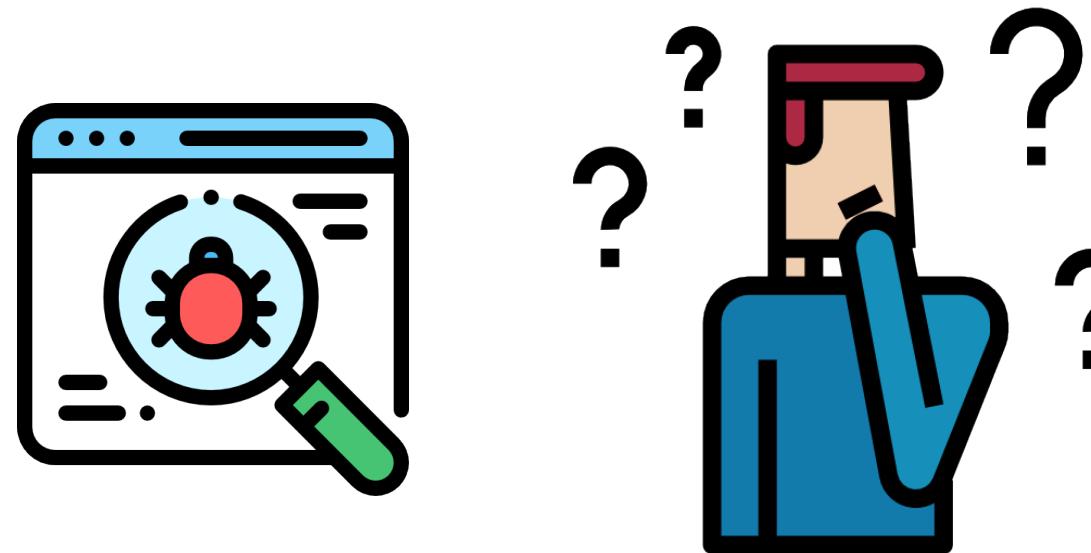


파이썬 내장 오류 Top 5 조사 결과

| | Type | Attribute | Value | Key | Import |
|---------------|-------|-----------|-------|-------|--------|
| StackOverflow | 31.5% | 19.4% | 27.8% | 8.3% | 13.0% |
| GitHub | 29.2% | 19.4% | 28.2% | 12.9% | 10.3% |

동기: 타입오류는 중요한 문제이다

1. 파이썬에서 가장 흔한 오류는 무엇일까?

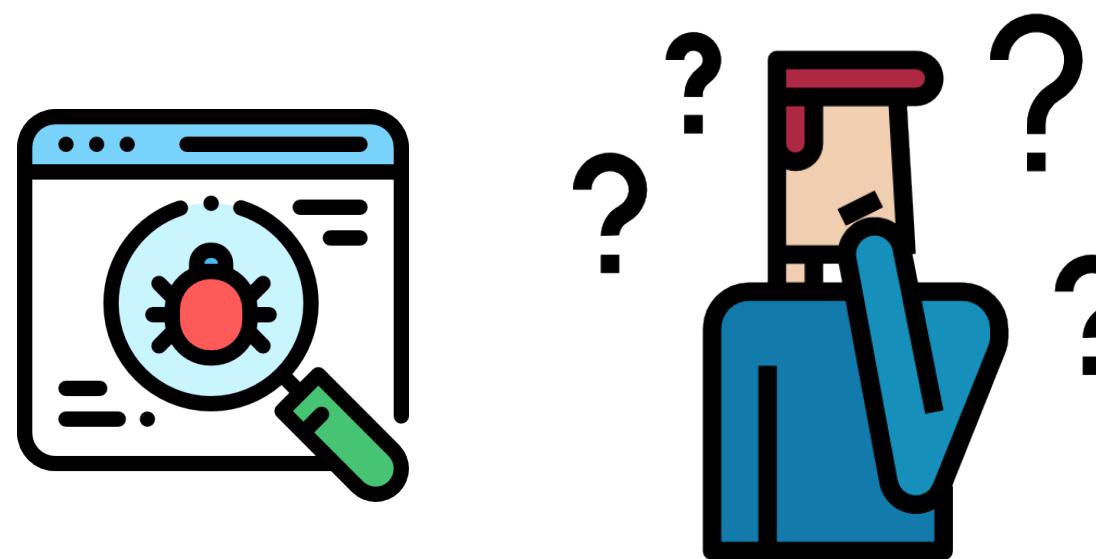


파이썬 내장 오류 Top 5 조사 결과

| | Type | Attribute | Value | Key | Import |
|---------------|-------|-----------|-------|-------|--------|
| StackOverflow | 31.5% | 19.4% | 27.8% | 8.3% | 13.0% |
| GitHub | 29.2% | 19.4% | 28.2% | 12.9% | 10.3% |

동기: 타입오류는 중요한 문제이다

1. 파이썬에서 가장 흔한 오류는 무엇일까?



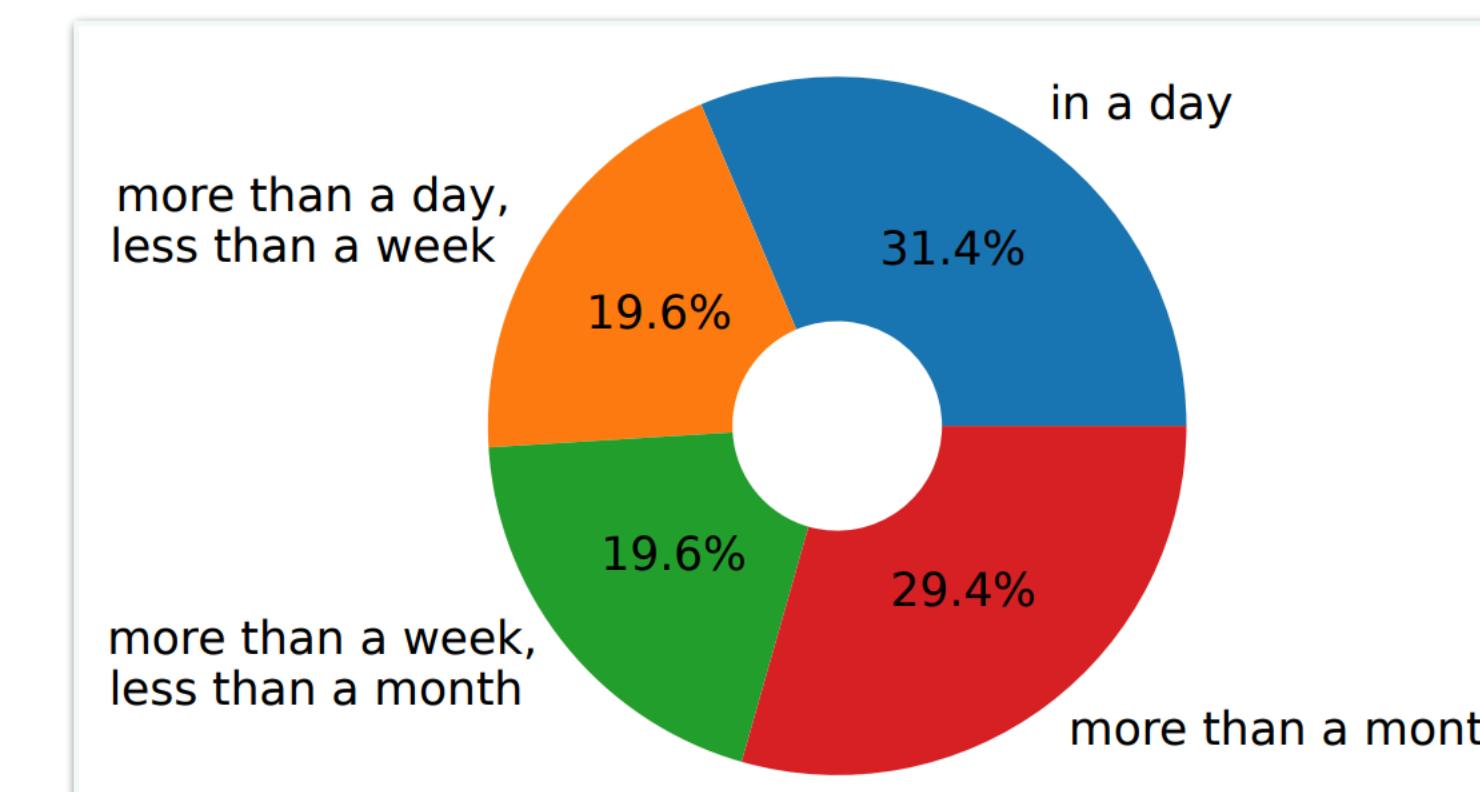
파이썬 내장 오류 Top 5 조사 결과

| | Type | Attribute | Value | Key | Import |
|---------------|-------|-----------|-------|-------|--------|
| StackOverflow | 31.5% | 19.4% | 27.8% | 8.3% | 13.0% |
| GitHub | 29.2% | 19.4% | 28.2% | 12.9% | 10.3% |

2. 개발자들이 타입오류를 고치는데 얼마나 시간이 소요될까?

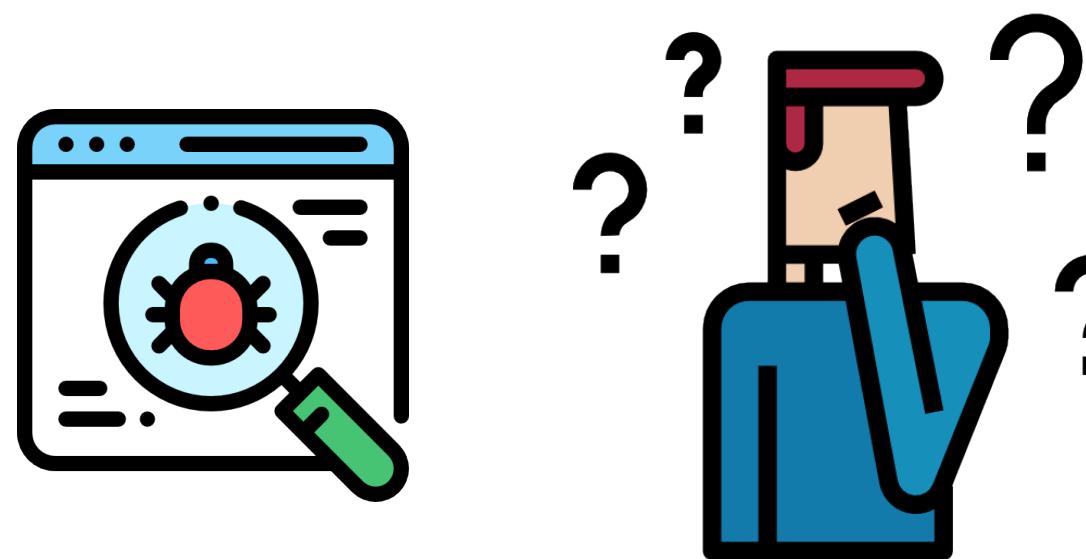


이슈가 처음 올라오고 고쳐질 때까지의 시간 조사 결과



동기: 타입오류는 중요한 문제이다

1. 파이썬에서 가장 흔한 오류는 무엇일까?



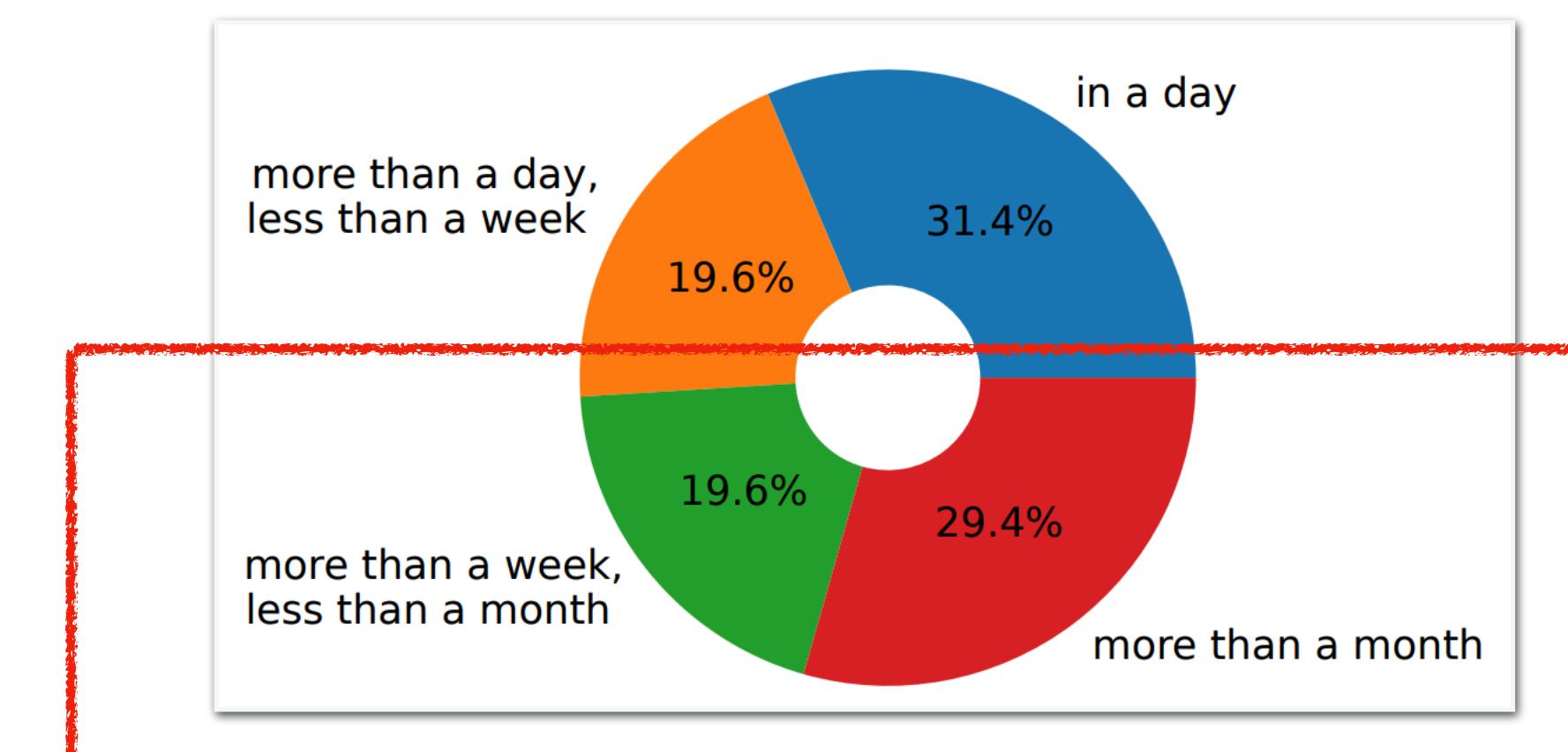
파이썬 내장 오류 Top 5 조사 결과

| | Type | Attribute | Value | Key | Import |
|---------------|-------|-----------|-------|-------|--------|
| StackOverflow | 31.5% | 19.4% | 27.8% | 8.3% | 13.0% |
| GitHub | 29.2% | 19.4% | 28.2% | 12.9% | 10.3% |

2. 개발자들이 타입오류를 고치는데 얼마나 시간이 소요될까?



이슈가 처음 올라오고 고쳐질 때까지의 시간 조사 결과



타입오류 절반 이상이
일주일 이상 시간이 걸렸다

동기: 타입오류는 중요한 문제이다

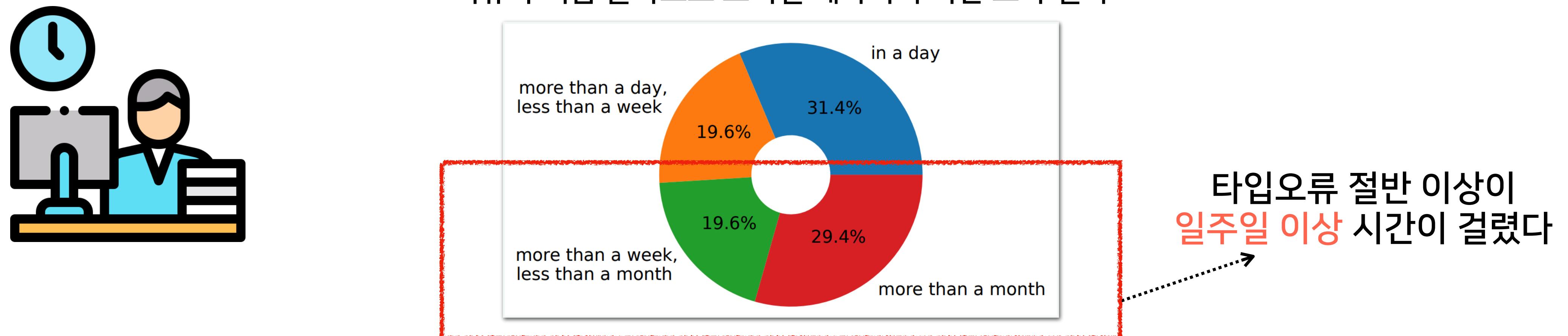
- 파이썬에서 가장 흔한 오류는 무엇일까?

파이썬 내장 오류 Top 5 조사 결과

| Type | Attribute | Value | Key | Import |
|----------------|-----------|-------|-----|--------|
| IndexError | | | | |
| NameError | | | | |
| TypeError | | | | |
| ValueError | | | | |
| AssertionError | | | | |

자동으로 타입오류를 고쳐보자!

- 개발자들



예제: 실제 타입오류 상황

Negative testcase

ret["comment"] = [1,2,3] (List of int)

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 )
```

예제: 실제 타입오류 상황

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

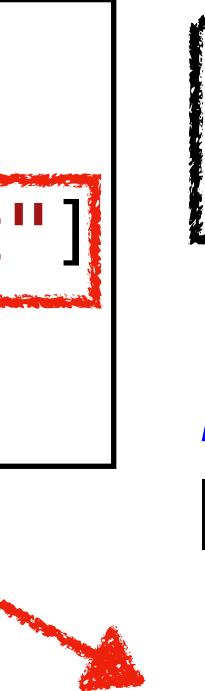
`ret["comment"] = [1,2,3] (List of int)`

`not ret["comment"]` 표현식이 `False`이므로,
line 3의 전체 표현식의 타입은 `List`

예제: 실제 타입오류 상황

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List



Negative testcase

```
ret["comment"] = [ 1,2,3 ] (List of int)
```

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, join 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함
=> Type Error

예제: 실제 타입오류 상황

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

```
ret["comment"] = [ 1,2,3 ] (List of int)
```

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, *join* 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함
=> Type Error

패치 생성

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

예제: 실제 타입오류 상황

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

패치 생성

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

Negative testcase

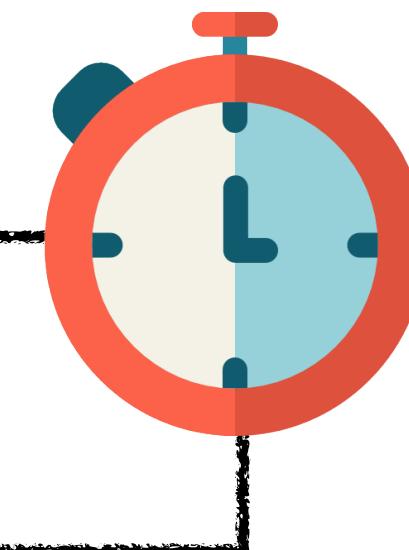
```
ret["comment"] = [ 1,2,3 ] (List of int)
```

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, join 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함
=> Type



181일



이슈가 처음 발생하고 고쳐지기까지의 시간

예제: 실제 타입오류 상황

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

```
ret["comment"] = [ 1,2,3 ] (List of int)
```

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, join 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함
=> Type Error

패치 생성

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

181일

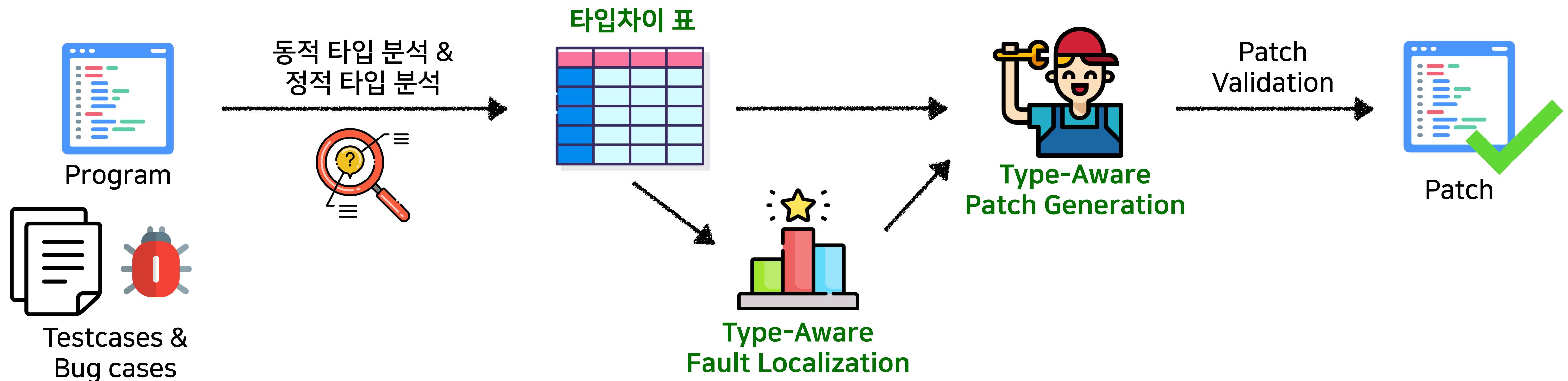
이슈가 처음 발생하고 고쳐지기까지의 시간

120초

11/56381

PyTER 개요

- 타입분석으로부터 타입차이 표 생성
- 타입차이 표를 이용하여 Fault Localization과 Patch Generation 진행



아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

```
ret["comment"] = [ 1,2,3 ] (List of int)
```

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, *join* 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함

패치 생성
어떻게?

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

ret["comment"] = [1,2,3] (List of int)

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

그러나, join 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | str |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

패치 생성
어떻게?

요약



Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else ret["comment"]  
4 ]  
5 )
```

False List

Negative testcase

ret["comment"] = [1,2,3] (List of int)

not ret["comment"] 표현식이 False이므로,
line 3의 전체 표현식의 타입은 List

패치 생성
어떻게?

요약

그러나, join 함수의 파라미터 타입은...
List of str여야함. List of List면 안됨.
따라서, line 3의 타입은 반드시 str이어야 함

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | str |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2 [  
3     "" if not ret["comment"] else str(ret["comment"])  
4 ]  
5 )
```

str

ret["comment"]의 타입을
List에서 str으로 바꿔보자

아이디어 1: 타입차이 찾기

Negative testcase

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 )
```

False List

ret["comment"] = [1,2,3] (List of int)

패치 생성
어떻게?

타입차이 표

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | str |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 )
```

str

ret["comment"] 의 타입을
List에서 str 으로 바꿔보자

아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 )
```

False List

Negative testcase

ret["comment"] = [1,2,3] (List of int)

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 )
```

str

타입차이 표

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | | |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 )
```

False List

Negative testcase

ret["comment"] = [1,2,3] (List of int)

동적 타입 분석

타입차이 표

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 )
```

str

아이디어 1: 타입차이 찾기

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 ) str False List
```

Negative testcase

ret["comment"] = [1,2,3] (List of int)

개발자는 타입을 유지하려고 한다

정적 타입 분석

타입차이 표

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | str |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 ) str
```

아이디어 1: 타입차이 찾기

Negative testcase

ret["comment"] = [1,2,3] (List of int)

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 ) str False List
```

패치 생성

타입차이 표

| 변수이름 | Negative type | Positive type |
|----------------|---------------|---------------|
| ret["comment"] | List | str |

Negative type : 타입오류 발생시의 타입

Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 ) str
```

ret["comment"] 의 타입을
List에서 str 으로 바꿔보자

아이디어 1: 타입차이 찾기

Negative testcase

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else ret["comment"]  
4     ]  
5 ) str False List
```

ret["comment"] = [1,2,3] (List of int)

아이디어 1
적절한 패치를 만들기 위해
타입분석을 이용하여 **타입차이** 표를 만들고
타입차이를 찾아낸다

Positive type
str

Negative type : 타입오류 발생시의 타입
Positive type : 프로그램이 올바르게 동작할 때의 타입

```
1 ret["comment"] = " ".join(  
2     [  
3         "" if not ret["comment"] else str(ret["comment"])  
4     ]  
5 ) str
```

ret["comment"] 의 타입을
List에서 str 으로 바꿔보자

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6
7
8     if self.binary:
9         return to_bytes(value)
10    else:
11        return to_unicode(value)
12
13 def to_unicode(value):
14     if isinstance(value, str):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18     ...
19
20 def to_bytes(value):
21     if isinstance(value, bytes):
22         return value
23     if not isinstance(value, str):
24         raise TypeError
25     ...
```

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6
7
8     if self.binary:
9         return to_bytes(value)
10    else:
11        return to_unicode(value)
12
13 def to_unicode(value):
14     if isinstance(value, str):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18 ...
19
20 def to_bytes(value):
21     if isinstance(value, bytes):
22         return value
23     if not isinstance(value, str):
24         raise TypeError
25 ...
```

Negative testcase

```
value = 1
self.binary = False
```

Assertion

value가 int이면,
value를 return 해라

라인 11, 16, 17을 따라 타입오류 발생

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6
7
8     if self.binary:
9         return to_bytes(value)
10    else:
11        return to_unicode(value)
12
13 def to_unicode(value):
14     if isinstance(value, (str, int)):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18     ...
19
20 def to_bytes(value):
21     if isinstance(value, bytes):
22         return value
23     if not isinstance(value, str):
24         raise TypeError
25     ...
```

Negative testcase

```
value = 1
self.binary = False
```

Assertion

value가 int이면,
value를 return 해라

라인 11, 16, 17을 따라 타입오류 발생

만약 value가 int이면, value를 return
올바른 패치일까?

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6
7     if self.binary:
8         return to_bytes(value)
9     else:
10        return to_unicode(value)
11
12 def to_unicode(value):
13     if isinstance(value, (str, int)):
14         return value
15     if not isinstance(value, bytes):
16         raise TypeError
17     ...
18
19 def to_bytes(value):
20     if isinstance(value, bytes):
21         return value
22     if not isinstance(value, str):
23         raise TypeError
24     ...
25
```

Negative testcase

value = 1
self.binary = False

Assertion

value가 int이면,
value를 return 해라

라인 11, 16, 17을 따라 타입오류 발생

self.binary 가 True이면,

라인 9, 23, 24를 따라 타입오류 발생

만약 value가 int이면, value를 return
올바른 패치일까?

Overfitting Patch

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6
7
8     if self.binary:
9         return to_bytes(value)
10    else:
11        return to_unicode(value)
12
13 def to_unicode(value):
14     if isinstance(value, str):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18     ...
19
20 def to_bytes(value):
21     if isinstance(value, bytes):
22         return value
23     if not isinstance(value, str):
24         raise TypeError
25     ...
```

Negative testcase

```
value = 1
self.binary = False
```

Assertion

value가 int이면,
value를 return 해라

라인 11, 16, 17을 따라 타입오류 발생

| 함수이름 | 변수이름 | Negative type | Positive type |
|---------------------------|-------|---------------|---------------------|
| _serialize_value (line 1) | value | int | BaseItem, dict, str |
| to_unicode (line 13) | value | int | str |

_serialize_value 함수에서 타입차이가 더 많이 관찰됨

더 의심스럽다!

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6         if isinstance(value, int):
7             return value
8     if self.binary:
9         return to_bytes(value)
10 else:
11     return to_unicode(value)
12
13 def to_unicode(value):
14     if isinstance(value, str):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18     ...
19
20 def to_bytes(value):
21     if isinstance(value, bytes):
22         return value
23     if not isinstance(value, str):
24         raise TypeError
25     ...
```

Negative testcase

```
value = 1
self.binary = False
```

Assertion

```
value가 int이면,
value를 return 해라
```

라인 11, 16, 17을 따라 타입오류 발생

| 함수이름 | 함수이름 | Negative type | Positive type |
|----------------------------|-------|---------------|---------------------|
| _serialize_value (line 11) | value | int | BaseItem, dict, str |
| to_unicode (line 13) | value | int | str |

_serialize_value 함수에서 타입차이가 더 많이 관찰됨

더 의심스럽다!

아이디어 2: Type-Aware Fault Localization

```
1 def _serialize_value(self, value):
2     if isinstance(value, BaseItem):
3         ...
4     if isinstance(value, dict):
5         ...
6         if isinstance(value, int):
7             return value
8     if self.binary:
9         return ...
10    else:
11        return ...
12
13 def to_unicode():
14     if isinstance(value, str):
15         return value
16     if not isinstance(value, bytes):
17         raise TypeError
18     ...
19
20 def to_bytes(value):
21     if not isinstance(value, bytes):
22         raise TypeError
23     if not self.binary:
24         raise TypeError
25     ...
```

Negative testcase

```
value = 1
self.binary = False
```

라인 11, 16, 17을 따라 타입오류 발생

Assertion

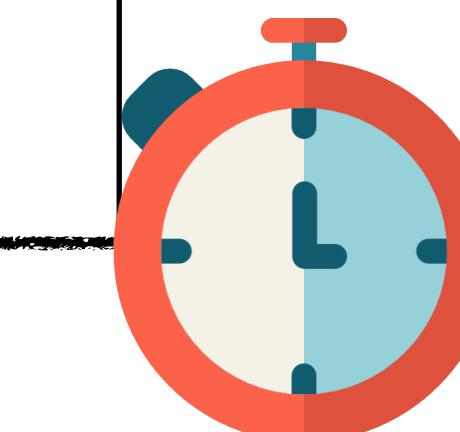
```
value가 int이면,
value를 return 해라
```

아이디어 2
타입차이가 더 많이 관찰된 곳 일수록,
더 의심스럽다!

| Positive type |
|---------------------|
| BaseItem, dict, str |
| str |

발됨

더 의심스럽다!



15초



81일



아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5
6
7     j = self.index_col[i]
8     ...
9 
```

Negative testcase

```
self.index_col = None
```

Assertion

타입오류 없음

아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5     if isinstance(self.index_col, None):
6         return self.index_col
7
8     j = self.index_col[i]
9     ...
```

Negative testcase

```
self.index_col = None
```

Assertion

타입오류 없음

아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5     if isinstance(self.index_col, None):
6         return self.index_col
7
8     j = self.index_col[i]
9     ...
```

Negative testcase

`self.index_col = None`

Assertion

타입오류 없음

Overfitting Patch

아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5     if isinstance(self.index_col, None):
6         return self.index_col
7
8     j = self.index_col[i]
9     ...
```

Negative testcase

`self.index_col = None`

Assertion

타입오류 없음

Overfitting Patch

이 함수의 `return` 타입은 아마 `bool` 타입일 것이다...
이것으로부터 이 패치의 순서를 뒤로 미룰 수 있음

아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5     if isinstance(self.index_col, None):
6         j = i
7     else:
8         j = self.index_col[i]
9     ...
```

Negative testcase

self.index_col is None

Assertion

오류 없음

아이디어 3: Type-Aware Patch Generation

```
1 def _should_parse_dates(self, i):
2     ...
3     if isinstance(self.parse_dates, bool):
4         return self.parse_dates
5     if isins
6         j =
7     else:
8         j =
9     ...
```

Negative testcase

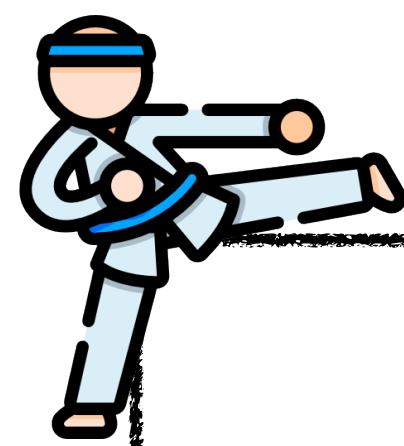
`self.index_col = None`

Assertion

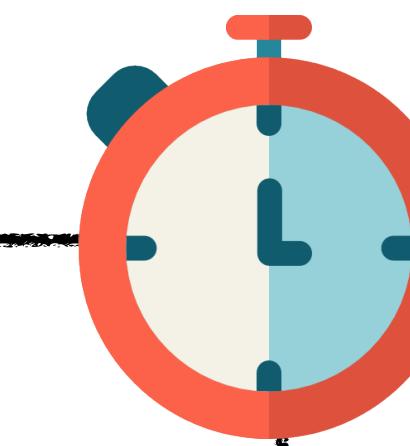
오류 없음

아이디어 3

타입 정보를 활용하여 패치 우선순위를 정할 수 있다



20초

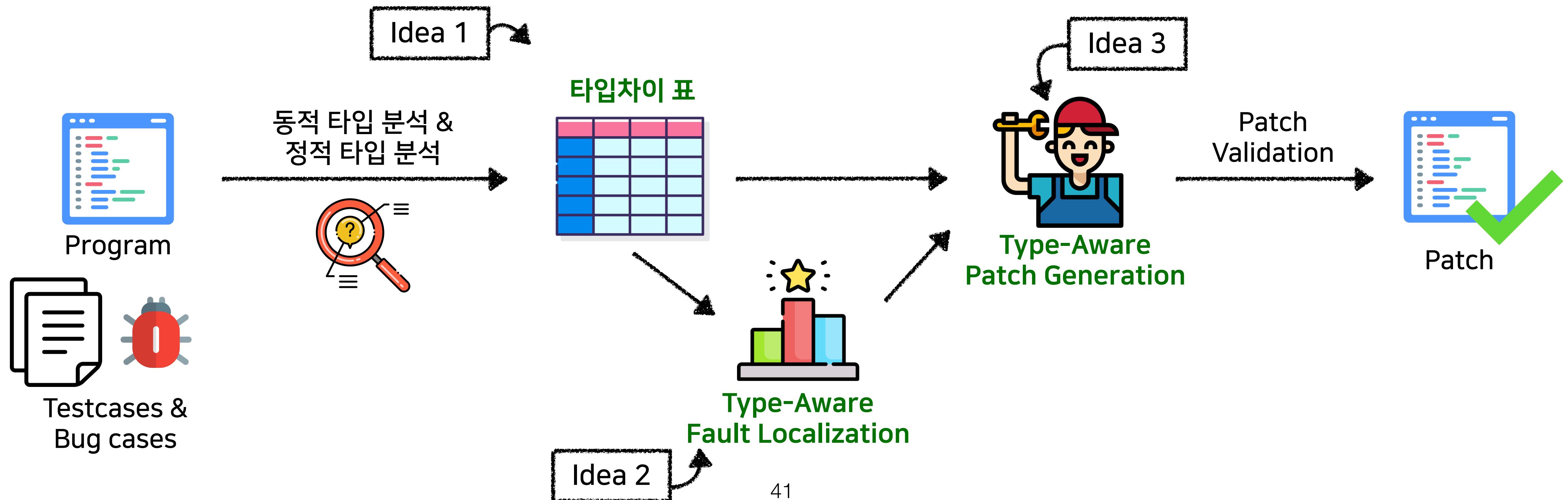


236일



PyTER 개요

- 타입분석으로부터 타입차이 표 생성
- 타입차이 표를 이용하여 Fault Localization과 Patch Generation 진행



성능 평가

- RQ1: PyTER가 얼마나 효율적으로 실제 타입오류를 고치는가?
- RQ2: PyTER에 사용된 핵심 기술이 얼마나 중요한가?
- **실험 Setting**
 - 벤치마크: TypeBugs (93), BugsInPy (57)
 - TypeBugs : Github 오픈소스 프로젝트를 대상으로 수집
 - BugsInPy (FSE 20, Ratnadira Widyasari et al.) : 타입오류 케이스만 이용
 - 베이스라인
 - Enumerate 기반 자동패치 기술 (= without PyTER 기술)
 - 시간 제한: 1h

| Program | #B | Avg. KLoC | NegTest | | PosTest | | BASELINE | | | PyTER | | | | | | | |
|----------|--------------|--------------|-------------|--------------|-------------|--------------|----------|----|-------------|--------|--------------|--------|----|-------------|--------|--------------|--------|
| | | | Avg. Num | Avg. Time | Avg. Num | Avg. Time | #G | #C | Fix Rate | Prec | Avg. Time | #G | #C | Fix Rate | Prec | Avg. Time | |
| TypeBugs | airflow | 7 | 73.9 | 1.0 | 53.4 | 13.3 | 43.9 | 3 | 0 | 0.0% | 0.0% | 232.0 | 4 | 3 | 42.9% | 75.0% | 50.5 |
| | beets | 1 | 21.4 | 1.0 | 4.0 | 9.0 | 7.0 | 1 | 0 | 0.0% | 0.0% | 8.1 | 1 | 0 | 0.0% | 0.0% | 3.6 |
| | core | 9 | 230.5 | 1.4 | 11.6 | 23.7 | 14.7 | 6 | 4 | 44.4% | 66.7% | 303.7 | 6 | 5 | 55.6% | 83.3% | 88.7 |
| | kivy | 1 | 44.5 | 1.0 | 6.0 | 4.0 | 26.0 | 1 | 0 | 0.0% | 0.0% | 400.57 | 1 | 0 | 0.0% | 0.0% | 54.3 |
| | luigi | 1 | 13.2 | 4.0 | 4.0 | 1.0 | 2.0 | 1 | 0 | 0.0% | 0.0% | 20.6 | 1 | 0 | 0.0% | 0.0% | 1.5 |
| | numpy | 3 | 53.4 | 2.0 | 6.0 | 72.3 | 4.7 | 1 | 0 | 0.0% | 0.0% | 1356.8 | 2 | 0 | 0.0% | 0.0% | 225.5 |
| | pandas | 45 | 86.9 | 7.0 | 42.6 | 141.1 | 125.8 | 21 | 11 | 24.4% | 52.4% | 1478.2 | 24 | 19 | 42.2% | 79.2% | 1063.4 |
| | rasa | 1 | 45.8 | 1.0 | 9.0 | 7.0 | 6.0 | 1 | 1 | 100.0% | 100.0% | 7.3 | 1 | 1 | 100.0% | 100.0% | 7.3 |
| | requests | 4 | 9.4 | 1.5 | 8.0 | 184.8 | 33.3 | 3 | 1 | 25.0% | 33.3% | 942.4 | 4 | 4 | 100.0% | 100.0% | 167.7 |
| | rich | 1 | 16.1 | 1.0 | 9.0 | 7.0 | 6.0 | 0 | 0 | 0.0% | n/a | 3600.0 | 0 | 0 | 0.0% | n/a | 3600.0 |
| | salt | 9 | 364.8 | 1.0 | 31.0 | 12.7 | 165.6 | 3 | 2 | 22.2% | 66.7% | 1265.5 | 6 | 6 | 66.7% | 100.0% | 93.8 |
| | sanic | 3 | 5.6 | 3.3 | 5.7 | 51.0 | 9.3 | 1 | 1 | 33.3% | 50.0% | 1406.6 | 3 | 3 | 100.0% | 100.0% | 43.5 |
| | scikit-learn | 5 | 63.5 | 1.0 | 8.0 | 75.8 | 13.2 | 3 | 2 | 40.0% | 66.7% | 1561.6 | 3 | 2 | 40.0% | 66.7% | 408.7 |
| | tornado | 1 | 12.6 | 1.0 | 8.0 | 171.0 | 20.0 | 1 | 1 | 100.0% | 100.0% | 1307.9 | 1 | 1 | 100.0% | 100.0% | 199.2 |
| | Zappa | 2 | 4.1 | 1.0 | 7.0 | 46.5 | 748.5 | 1 | 0 | 0.0% | 0.0% | 1876.3 | 1 | 1 | 50.0% | 100.0% | 1663.9 |
| | Total | 93 | 112.7 | 4.1 | 30.7 | 91.9 | 101.2 | 47 | 23 | 24.7% | 48.9% | 1196.0 | 58 | 45 | 48.4% | 77.6% | 651.2 |
| BugsInPy | ansible | 1 | 126.5 | 1.0 | 21.0 | 30.0 | 13.0 | 0 | 0 | 0.0% | n/a | 3600.0 | 0 | 0 | 0.0% | n/a | 2549.7 |
| | fastapi | 2 | 7.4 | 1.0 | 5.5 | 7.0 | 3.0 | 0 | 0 | 0.0% | n/a | 10.8 | 0 | 0 | 0.0% | n/a | 1.5 |
| | keras | 5 | 26.5 | 1.0 | 23.8 | 14.8 | 50.4 | 1 | 0 | 0.0% | 0.0% | 2162.9 | 1 | 1 | 20.0% | 100.0% | 1769.8 |
| | luigi | 7 | 12.3 | 1.1 | 9.3 | 54.9 | 23.0 | 4 | 1 | 14.3% | 25.0% | 207.1 | 5 | 3 | 42.9% | 60.0% | 15.1 |
| | matplotlib | 1 | 94.9 | 1.0 | 8.0 | 214.0 | 41.0 | 0 | 0 | 0.0% | n/a | 0.0 | 0 | 0 | 0.0% | n/a | 0.0 |
| | pandas | 25 | 87.9 | 9.2 | 67.4 | 250.8 | 130.5 | 11 | 4 | 16.0% | 36.4% | 2040.3 | 13 | 7 | 28.0% | 53.8% | 1636.7 |
| | scrappy | 10 | 12.7 | 1.6 | 8.7 | 21.0 | 6.5 | 6 | 2 | 20.0% | 33.3% | 62.2 | 6 | 5 | 50.0% | 83.3% | 12.0 |
| | spacy | 1 | 78.6 | 1.0 | 7.0 | 6.0 | 7.0 | 1 | 0 | 0.0% | 0.0% | 9.7 | 1 | 0 | 0.0% | 0.0% | 79.8 |
| | tornado | 2 | 13.3 | 1.0 | 3.5 | 44.0 | 3.0 | 1 | 0 | 0.0% | 0.0% | 1.2 | 1 | 1 | 50.0% | 100.0% | 0.8 |
| | tqdm | 1 | 1.9 | 1.0 | 6.0 | 54.0 | 4.0 | 0 | 0 | 0.0% | n/a | 0.0 | 0 | 0 | 0.0% | n/a | 0.0 |
| | youtube-dl | 2 | 112.0 | 1.0 | 9.0 | 81.5 | 8.5 | 1 | 1 | 50.0% | 100.0% | 1815.2 | 1 | 1 | 50.0% | 100.0% | 1802.9 |
| | Total | 57 | 54.6 | 4.7 | 35.7 | 131.7 | 67.3 | 25 | 8 | 14.0% | 32.0% | 1248.4 | 28 | 18 | 31.6% | 64.3% | 968.5 |

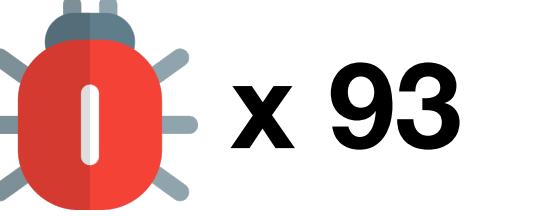
성능 평가: RQ1

#Correct : 올바른 패치 수
#Plausible : 테스트 통과 패치 수
#Total : 총 버그 개수

패치율
#Correct / #Total 정확도
#Correct / #Plausible

평균 시간

TypeBugs



PyTER

Baseline

48.4%

77.6%

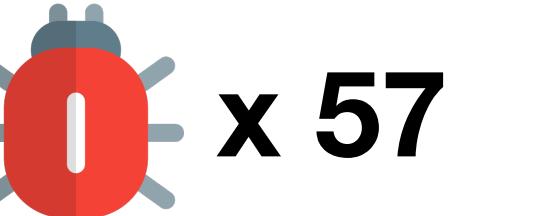
651.2s

24.7%

48.9%

1196.0s

BugsInPy



PyTER

Baseline

31.6%

64.3%

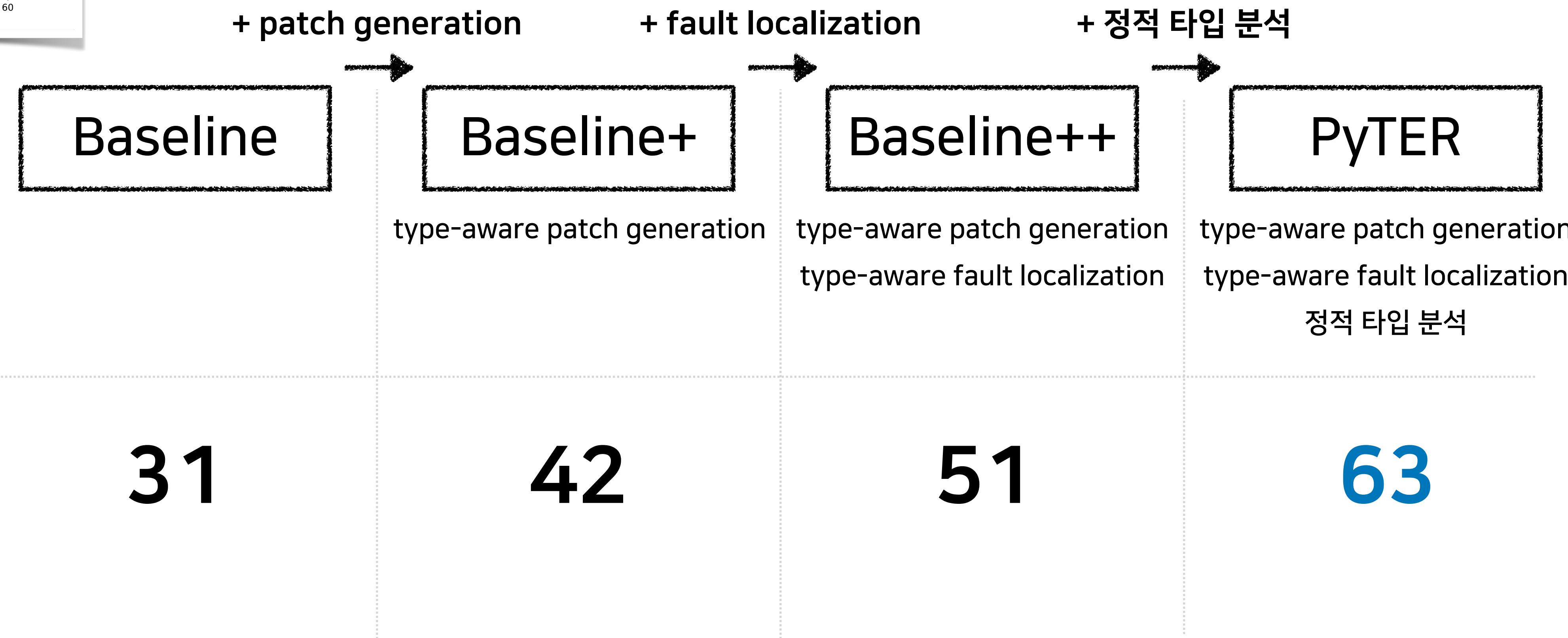
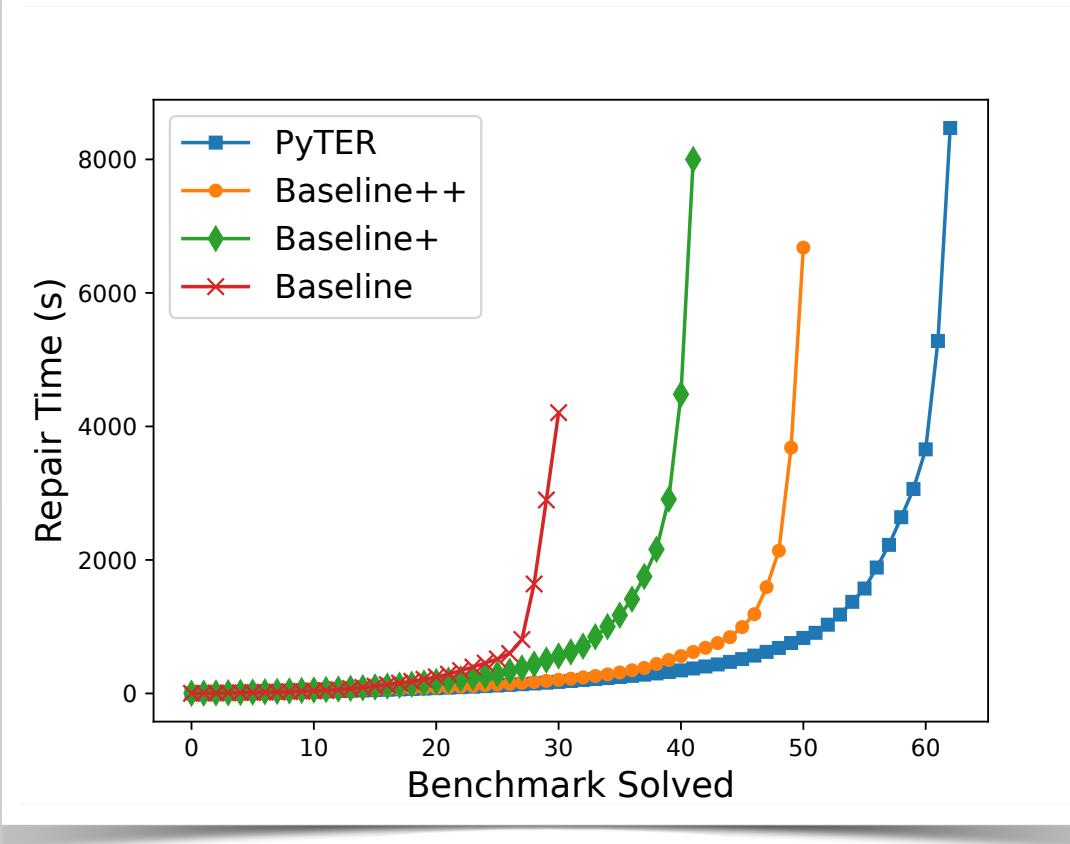
968.5s

14.0%

32.0%

1248.4s

성능 평가: RQ2



정리

- 실제 파이썬 프로그램의 타입오류를 고치는 첫번째 기술인 PyTER 제안
- 타입오류를 고치기 위해 타입차이를 활용하는 기술들을 제안

감사합니다!