requires-python = ">=3.14"

Overview of new features in Python 3.14

Contents

PEP 649 and PEP 749: Deferred evaluation of annotations

Eager evaluation of type annotations

```
1   def add(a: int, b: int) -> int:
2     return a + b
```

the type annotations are processsed when the function or class is defined.

PEP 649 and PEP 749: Deferred evaluation of annotations

__annotations__

```
1  def add(a: int, b: int) -> int:
2    return a + b
3
4  print(add.__annotations__)
5
6  {'a': <class 'int'>, 'b': <class 'int'>, 'return':
   <class 'int'>}
```

PEP 649 and PEP 749: Deferred evaluation of annotations

Foward Reference Problem

```
1  class Node:
2  # This would normally raise a NameError because
   'Node'
3      # isn't fully defined yet!
4      def add_neighbor(self, neighbor: Node):
5          ...
```

annotation for node is referred before processed

PEP 649 and PEP 749: Deferred evaluation of annotations

from __future__ import annotations

```python
from __future__ import annotations

class Node:
    def add_neighbor(self, neighbor: Node):
        ...
```

make the type annotations are not processed as object, but as a literal.

PEP 649 and PEP 749: Deferred evaluation of annotations

from __future__ import annotations

```
1  print(add.__annotations__)
2
3  {'a': 'int', 'b': 'int', 'return': 'int'}
```

make the type annotations are not processed as object, but as a literal.

PEP 649 and PEP 749: Deferred evaluation of annotations

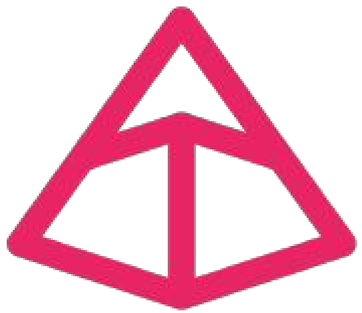
from __future__ import annotations

```
1  from __future__ import annotations
2
3  def add(a: int, b: int) -> int:
4    return a + b
5
6  print(add.__annotations__)
7
8  {'a': 'int', 'b': 'int', 'return': 'int'}
```

make the type annotations are not processed as object, but as a literal.

PEP 649 and PEP 749: Deferred evaluation of annotations

evaluation overhead



It makes overhead for the python packages having type validation logic.

with ***inspect.get_annotations()*** or ***typing.get_type_hints()***

PEP 649 and PEP 749: Deferred evaluation of annotations

Python3.14: lazy evaluation

```
1  from annotationlib import get_annotations, Format
2  def func(arg: Undefined):
3      pass
4  get_annotations(func, format=Format.VALUE)
5  get_annotations(func, format=Format.FORWARDREF)
6
7  get_annotations(func, format=Format.STRING)
```

storing annotations as Annotate Function executed only in special case.

PEP 750: Template strings

Prefixed strings in Python

```python
# 1. f-string: 변수 삽입 및 연산 (가장 많이 사용)
name = "Python"
version = 3.12
f_message = f"Hello, {name} {version}! 1 + 1은 {1 + 1}입니다."

# 2. r-string: 백슬래시(\)를 일반 문자로 취급 (경로, 정규식에 유용)
r_path = r"C:\users\name\documents\notes"

# 3. b-string: 바이너리(바이트) 데이터 처리
# 영문/숫자 등 ASCII 범위 내 문자만 가능하며, 타입이 'bytes'가 됩니다.
byte_data = b"Hello, Binary World"
```

PEP 750: Template strings

f-string

```python
# 1. f-string: 변수 삽입 및 연산 (가장 많이 사용)
name = "Python"
version = 3.12
f_message = f"Hello, {name} {version}! 1 + 1은 {1 + 1} 입니다."

f_message

"Hello, Python 3.12! 1 + 1은 2입니다."
```

PEP 750: Template strings

eagar evaluation of f-string

```
1    # 1. f-string: 변수 삽입 및 연산 (가장 많이 사용)
2    name = "Python"
3    version = 3.12
4    f_message = f"Hello, {name} {version}! 1 + 1은 {1 + 1}
     입니다."
5
6    f_message
7
8    "Hello, Python 3.12! 1 + 1은 2입니다."
```

PEP 750: Template strings

eagar evaluation of f-string

```
# The "Template"
user_id = "123"
query = f"SELECT * FROM users WHERE id = {user_id}"
# Result: "SELECT * FROM users WHERE id = 123" (Looks fine)
```

WHAT IF... ***user_id = 123; DROP TABLE users; --***

PEP 750: Template strings

eagar evaluation of f-string

```
1  pi = 3.14
2  t_message = t't-strings are new in Python {pi!s}!'
3
4  t_message
5
6  Template(
7     strings=('t-strings are new in Python ', '!'),
8     interpolations=(Interpolation(3.14, 'pi', 's',
   ''),)
9  )
```

- string: **static** part of template string
- interpolation: **dynamic** part of template string

PEP 750: Template strings

render string with renderer function

- **sql**

```
# The 'db.execute' renderer knows exactly how to handle
# the 'input_val' without you touching it.
input_val = "배인수"
db.execute(t"SELECT * FROM users WHERE name = {input_val}")
```

- **html**

```
attributes = {"src": "shrubbery.jpg", "alt": "looks nice"}
template = t"<img {attributes} />"
assert html(template) == '<img src="shrubbery.jpg" alt="looks nice" />'
```

PEP 758: Allow except and except* expressions without brackets

```python
# Before Python 3.14 (brackets are necessary)
try:
    ...
except (ValueError, TypeError) as e:
    ...

# After Python 3.14

try:
    ...
except ValueError, TypeError as e:
    ...
```

error handling is possible without brackets

PEP 765: Control flow in finally blocks

```python
import sqlite3

def update_user_points(user_id, points):
    conn = sqlite3.connect('example.db')
    try:
        cursor = conn.cursor()
        # 데이터 처리 중 에러 발생 가능성 상존
        cursor.execute("UPDATE users SET points = ? WHERE id = ?", (points, user_id))
        conn.commit()
    except Exception as e:
        conn.rollback()  # 에러 시 변경사항 되돌리기
        print(f"트랜잭션 실패: {e}")
    finally:
        # 커밋 성공 여부와 상관없이 DB 연결은 반드시 닫아야 함
        conn.close()
        print("DB 연결을 종료했습니다.")
```

***finally* in python**

PEP 765: Control flow in finally blocks

silent cancellation

```
1  def dangerous_code():
2      try:
3          raise ValueError("Error raised!")
4      finally:
5          return "done"
6
7
8  dangerous_code()
9
10 # result
11 done
```

the error is canceled.

PEP 765: Control flow in finally blocks

Python 3.14 raises SyntaxWarnging into slient cancellation

```
1  def dangerous_code():
2      try:
3          raise ValueError("Error raised!")
4      finally:
5          return "done"
6
7
8  dangerous_code()
9
10 # result
11 SyntaxWarning: 'return' in a 'finally' block
12   return "done"
13 done
```

The compiler emits a SyntaxWarning when
a return, break or continue appears in a finally block

PEP 784: Zstandard support in the standard library

Multiple data conpression algorithms in the world

Method	Speed	Compression Ratio	Best Use Case	ext.
zstd	Very Fast	High	Modern apps, real-time data	.zst
lzma	Slow	Highest	Maximum space-saving (Backups)	.7z, .xz
gzip	Fast	Medium	Web assets, general compatibility	.gz, .zip
bz2	Medium	High	Large text/log files	.bz2

PEP 784: Zstandard support in the standard library

Zstandard

Zstandard

fast compression algorithm, providing high compression ratios by Meta

PEP 784: Zstandard support in the standard library

Zstandard: code example

```python
from compression import zstd
import math

data = str(math.pi).encode() * 20
compressed = zstd.compress(data)
ratio = len(compressed) / len(data)
print(f"Achieved compression ratio of {ratio}")
```

we can use *zstd* with ***compression***, new built-in package

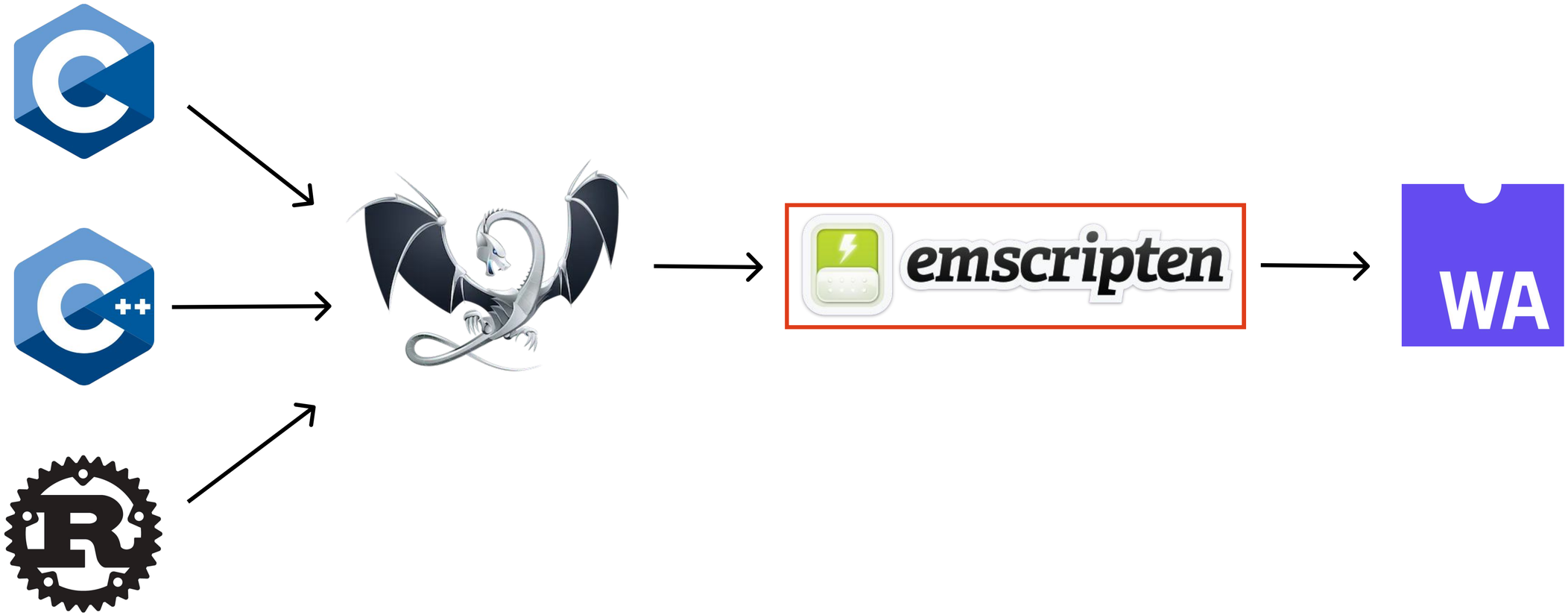
PEP 776: Emscripten is now an officially supported platform, at tier 3.

Emscripten

a complete compiler toolchain to WebAssembly, using LLVM,
with a special focus on speed, size, and the Web platform.

PEP 776: Emscripten is now an officially supported platform, at tier 3.

compiler process



PEP 776: Emscripten is now an officially supported platform, at tier 3.

Python Tier System

등급	핵심 요약	상세 내용
Tier 1	완전 보장	가장 대중적인 플랫폼(Windows, macOS, 현대적 Linux). CI(지속적 통합) 빌드가 실패하면 배포 자체가 중단
Tier 2	안정적 지원	Tier 1만큼은 아니지만, 개발팀이 신경을 많이 쓰는 Tier. 빌드 실패 시 배포를 막지는 않지만, 반드시 수정해야 할 문제로 간주
Tier 3	공식 인정 시작	동작은 확인하지만, 100% 보장하진 않음 단계. 공식 플랫폼으로 이름은 올렸지만, 빌드가 깨져도 릴리스를 중단하지는 않음

PEP 776: Emscripten is now an officially supported platform, at tier 3.

before supported

Python distribution for the browser and Node.js based on WebAssembly.

cpython

E.O.D