Five Ways to Write Better Python

Taken from Effective Python: 90 Specific Ways to Write Better Python, Second Edition by Brett Slatkin

Oscar Hill 04/11/2024

Be Defensive When Iterating Over Arguments

- An iterator will only produce its results once.
- Iterating over an exhausted iterator will not throw an error.
- Pass containers instead of iterators to functions.
- Raise a `TypeError` if an iterator is passed to a function.

```
def sum_twice(l):
    total = 0
    for i in l:
        total += i
    for i in l:
        total += i
    return total

def safe_sum_twice(l):
    if iter(l) is l:
        raise TypeError("Must supply a container.")

    total = 0
    for i in l:
        total += i
    for i in l:
        total += i
    return total

def numbers_below(x):
    for i in range(x):
        yield i

class NumbersBelow:
    def __init__(self, x):
        self.x = x

    def __iter__(self):
        for i in range(self.x):
            yield i
```

Consider `concurrent.futures` for True Parallelism

- Python's global interpreter lock prevents true parallelism in threads.
- `concurrent.futures` runs additional interpreters as child processes, allowing multiple cores to be used.
- `ProcessPoolExecutor` uses `pickle` and sockets internally.

Profile Before Optimising

- Because Python is dynamic, it is not always clear what is causing code to run slowly.
- Therefore it is important to profile a Python program before optimizing it.

```python
# Use cProfile rather than profile
# because it has less of an impact
# on the performance of the program
# being profiled.
from cProfile import Profile
from pstats import Stats

def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

def factorial_wrapper(n):
    return factorial(n)

profiler = Profile()
profiler.runcall(lambda: factorial_wrapper(950))

stats = Stats(profiler)
stats.strip_dirs()
stats.sort_stats('cumulative')
stats.print_stats()
stats.print_callers()
```

Consider Static Analysis via `typing` to Obviate Bugs

- Historically, Python has not provided compile-time type safety.
- Python has now introduced special syntax and the built-in `typing` module, which allow you to provide type information.
- Static analysis tools can then identify bugs.
- This allows gradual typing.

```python
from typing import Optional

def increment(value: Optional[int]) -> int:
    return value + 1
```

Validate Subclasses with `__init_subclass__`

- Metaclasses can be used to inspect or alter a class.
- A more lightweight alternative, `__init_subclass__`, can be used to check subclasses when they are defined, before they are used to create objects.

```
class Item:
    age = 0

    def __init_subclass__(cls):
        super().__init_subclass__()
        if cls.age < 0:
            raise ValueError("age must be greater than 0.")

class Table(Item):
    age = -1
```