ChatGPT

Rust's `gen` Keyword: Generators and Coroutines in Evolution

Introduction to `gen` and Rust Generators

Rust is introducing a reserved keyword `gen` (as of the 2024 edition) to enable **generator blocks** in the language [1]. A generator (in this context) refers to a function-like construct that can **yield** a sequence of values lazily, similar to how Python's generators work. The design is analogous to Rust's existing `async` blocks: just as an `async` block produces a value that can be awaited with `.await`, a `gen` block produces a value that can be iterated with a `for` loop [1]. This feature aims to simplify writing iterators by allowing developers to write straightforward looping code with `yield` statements, instead of manually implementing the `Iterator` trait state machine. The `gen` keyword has been officially reserved in the 2024 Edition of Rust for this purpose [1], paving the way for stabilized generator support.

Motivation: Writing complex iterators by hand can be error-prone and verbose. The Rust community has long desired a more ergonomic way to create iterators (often seen in other languages as "generator functions"). The new `gen` blocks address this by letting developers yield values on the fly, creating an iterator lazily. For example, the Rust RFC 3513 demonstrates how run-length encoding (RLE) can be implemented with a `gen` block versus a manual iterator. The `gen` version uses simple control flow with `yield`, whereas the manual version requires a dedicated struct and intricate state management [2] [3]. This aligns with Rust's goal of high performance *and* high-level ergonomics, by generating the necessary iterator boilerplate under the hood.

Importantly, *"generator"* in Rust is being defined in a specific way: as a special case of a more general coroutine. According to the Rust team, "a generator is just a convenient way to write `Iterator` implementations," essentially **a coroutine with no explicit arguments and no explicit return type** (the iteration ends when it returns) [4]. In other words, when a Rust generator finishes execution (hits a return or the end of its block), it signals that iteration is complete (just like returning `None` ends a normal iterator) [4]. This definition clarifies how `gen` fits into Rust's broader plans for coroutines.

Rust's Design for Generators (`gen` Blocks)

Under Rust's current design, a `gen { ... }` block allows you to write code that can yield multiple values over time. Each `yield` statement inside the block produces one value of the generator's output sequence. The block as a whole evaluates to a generator object that can be iterated. In practice, you can use it by iterating directly or converting it into a standard iterator. For instance:

```
// Pseudo-code illustrating a generator block usage (post-Rust 2024 edition)
let my_iter = gen {
    for i in 1..=5 {
        yield i * 2;  // yield even numbers up to 10
    }
};  // my_iter is a lazy iterator producing 2,4,6,8,10

for val in my_iter {
    println!("Got {val}");
}
```

In this example, the code inside `gen { ... }` is not executed immediately. Instead, it's stored as a suspended computation. Only when the generator is iterated (e.g. in the `for` loop) does the code run, yielding values one by one. This lazy execution model mirrors how Rust's `async` blocks work (they don't run until polled) and matches the behavior of generators in other languages like Python or C# – the code inside doesn't execute until you begin iteration [5].

Rust generator blocks can use `yield` anywhere within the block to emit a value. The yielded values must all have the same type (the output type of the iterator). A simple example from RFC 3513 illustrates usage in a real scenario. The following function uses a `gen` block to implement run-length encoding lazily, yielding byte counts and values as pairs:

```
fn rl_encode<I: IntoIterator<Item = u8>>(xs: I) -> impl Iterator<Item = u8> {
    gen {
        let mut xs = xs.into_iter();
        let (Some(mut cur), mut n) = (xs.next(), 0) else {
            return; // end generator if input is empty
        };
        for x in xs {
            if x == cur && n < u8::MAX {
                n += 1;
            } else {
                yield n;
                yield cur;       // yield a count and a byte value
                (cur, n) = (x, 0);
            }
        }
        yield n;
        yield cur;               // yield the last count and value
    }.into_iter()
}
```

In this code, each `yield` produces a `u8` value, and the final `return` (implicit at end or explicit `return;`) signifies no more values (ending the iteration). The `.into_iter()` call converts the generator block into a concrete `Iterator` (the generator object itself can implement `IntoIterator` to allow this) [3] [6]. The net effect is that `rl_encode(xs)` returns a lazy iterator. Internally, the Rust compiler will transform the `gen` block into a state machine struct that implements iteration, managing local variables (`cur`, `n`, etc.) across suspensions.

Traits and Types: Behind the scenes, a generator in Rust implements a special trait (currently unstable) in `std::ops`. Historically this was called the `Generator` trait, but it was recently renamed to `Coroutine` [7] to emphasize its generality. This trait has an associated **Yield type** (the type of values yielded) and a **Return type** (the type produced when the generator finishes) [8] [9], as well as a **Resume argument type** (the type of value that can be sent into the generator when resuming it, defaulting to `()` for generators that don't take input) [10]. For `gen` blocks intended as iterators, the resume type is `()` and the return type is effectively `()` as well (or unused), since ending a generator corresponds to an iterator finishing (returning `None`) [11]. The yield type, however, is set to the iterator's item type (like `u8` in the example above). Each call to `next()` on the iterator essentially calls the generator's internal `resume()` method with no argument, causing the generator's execution to continue from the last `yield` point and run until the next `yield` (or completion). The value passed to `yield` inside the generator is what gets produced as the iteration output [12] [13].

It's worth noting that Rust ensures memory and type safety across these suspension points. For example, if a generator holds a reference across a `yield` suspension, the compiler will enforce that the generator cannot be moved in memory unless it's pinned (similar to `async` futures). These details are mostly handled by the compiler and typically invisible to users writing simple `gen` blocks, but they highlight the complexity under the hood.

Finally, because `gen` blocks integrate with the `Iterator` trait, they can be directly used in `for` loops or with iterator adapters. The values are produced lazily and on-demand. This approach retains Rust's zero-cost abstractions: the generated code is essentially equivalent to a hand-written state machine, giving performance similar to manually implementing the iterator (with optimizations). In short, the `gen` keyword provides a **more ergonomic syntax** for something Rust could already do in theory, without compromising on performance or safety.

Evolution of Coroutines in Rust (Beyond Basic Generators)

Rust's adoption of `gen` is part of a broader story of **coroutine support** in the language. The term *coroutine* refers to a generalization of functions that can be suspended and resumed (of which generators are a subset). Rust initially introduced coroutines in an experimental form back in 2017 (RFC 2033) to power the async/await system [14] [15]. In that design, a **stackless coroutine** mechanism was added to the compiler: essentially, any function that could suspend (an `async fn` or a generator closure) would be transformed into a state machine. This is how `async fn` works under the hood, and it's also how the unstable generator feature (enabled via `#![feature(generators)]`) worked. Rust's internal **Generator trait** (now renamed to **Coroutine trait**) was the core of this system.

Until recently, the Generator/Coroutine feature remained unstable and was used mainly as an implementation detail for `Future`s (i.e., `async fn`) [8]. However, there has been renewed effort to stabilize and extend it for general use, such as for writing iterators more ergonomically (hence `gen`). In late 2023, the Rust compiler team renamed the internal `Generator` trait to `Coroutine` [7], acknowledging that it represents a more powerful concept than simple iterators. As the team noted: *"Our* `Generator` *trait was effectively a coroutine already... All nightly users using the* `Generator` *trait must now update their code to the new trait names."* [7]. This change frees up the term "generator" to refer to the higher-level user feature (the nice syntax for iterators), while using "coroutine" for the underlying mechanism. Put another way, **every Rust generator is a coroutine, but not every coroutine is just a simple iterator** [4].

`gen` **Blocks and Coroutine Capabilities:** The current `gen` block proposal purposely limits itself to the iterator use-case (no arguments, no distinct return type beyond signaling completion) [4]. This is by design – it covers the common need (lazy iteration) first, while leaving more advanced coroutine features for future proposals [16] [17]. Because it's built on the coroutine infrastructure, Rust can potentially extend generators to be more powerful. Some anticipated evolutions include:

- **Generator Functions (`gen fn`):** The ability to define a named function that yields values, rather than an inline block or closure. The RFC notes that the syntax for `gen fn` is still an open design question and was deliberately left out of the initial feature [16]. In the future, you might write something like `gen fn foo() -> T { ... }` or other syntax to declare a generator function that returns an iterator. Deciding how to spell the return/yield types in the signature is a part of that design space [18].

- **Resume Arguments (Coroutines with inputs):** In the full coroutine model, a coroutine can receive values each time it resumes (similar to how Python's generators can receive sent values). The underlying `Coroutine` trait already supports a resume argument type [10], but in today's `gen` blocks it's fixed to `()` (no input) for simplicity. There have been RFC discussions (e.g. *"Generator resume arguments"* RFC PR 2781) on how to unify coroutines with closures by allowing arguments to be passed in on each resume [19] [20]. If Rust enables this, you could have more general coroutines that act like Python's `gen.send(value)` or even implement cooperative multitasking patterns. However, this is complex to stabilize – it involves borrow checker considerations and how `yield` expressions would capture incoming values – so it remains a future possibility. For now, Rust generators do **not** support receiving values from the caller on the fly (only yielding out), apart from maybe using shared mutable state as a workaround.

- **Asynchronous Generators (Streams):** Another likely evolution is combining `async` with `gen` to yield values asynchronously (sometimes called **streams** in async programming). Python, for instance, supports *asynchronous generators* with `async def ...: yield ...` (and these produce async iterators that you consume with `async for`). Rust could introduce something analogous, perhaps an `async gen` block or an `async gen fn` that yields values which are produced asynchronously. The RFC hints at this: it notes that using `.await` inside a `gen` block (making it an async generator) is left for future work [21]. The main challenge is handling **self-referential** suspension – holding references across both yield points and await points is tricky. The RFC 3513 states that the design is forward-compatible with such extensions, but solving it is non-trivial and will be addressed later [22]. In practical terms, a future Rust might have a first-class `Stream` type (the async analog of `Iterator`) powered by `async gen` constructs, simplifying how we create asynchronous iterators (streams) without external crates.

- **Coroutine-based APIs & Effects:** Looking further ahead, one can imagine Rust exposing more coroutine capabilities for advanced use cases, such as cooperative tasks, generators that can terminate with a value (and that value accessible to the caller in some way, akin to Python's generator `return`), or integrating with error handling (`?` is already allowed in `gen` blocks to propagate errors as an early return of the generator [21], and there's mention of a potential `gen try fn` in the future [23] to yield values until an error occurs). All these indicate that Rust's core coroutine mechanism could underpin multiple language features.

In summary, the introduction of `gen` is a significant step in Rust's coroutine story: it *reintroduces* generators for iteration in a safe and ergonomic form [24], and lays groundwork for broader coroutine support. The Rust team's phrasing "coming full circle" [24] highlights that Rust had internal generators (for async) and even earlier experiments with coroutines, and now those ideas are coming to fruition in the language proper. While the initial stabilized feature will focus on the common use-case (synchronous iterators), it is part of a path toward more powerful coroutine features that Rust developers have been exploring for years.

Comparison: Rust's Generators/Coroutines vs Python's Generators/Coroutines

Rust's approach to generators and coroutines has been heavily informed by experiences in other languages like Python, but the two languages differ in syntax, philosophy, and capabilities. Here we compare them in key areas:

Syntax and Usage

- **Defining a Generator:** In Python, any function that contains a `yield` statement automatically becomes a generator function. You define it with normal `def` syntax and use `yield` inside; calling that function returns a generator object. For example, `def count(): yield 1; yield 2` defines a generator that yields 1 and 2. In Rust, by contrast, generator syntax is explicit: you either use a `gen { ... }` block as an expression or (eventually) a `gen fn`. Rust chose to introduce a new keyword and block form (`gen`) to clearly delineate generator logic, rather than having any function with a `yield` become special. This fits Rust's style of being explicit with keywords like `async` and (in the future) `gen`. The `yield` keyword itself is used *inside* Rust `gen` blocks to produce values, similar to Python's `yield`. One consequence of Rust's design is that you cannot accidentally create a generator – it has to be marked with `gen` – whereas in Python simply writing a `yield` makes the function a generator.

- **Iterating and Awaiting:** Both languages allow using generators in loops, but the syntax differs. In Rust, a generator block yields values that implement `IntoIterator`, so you can write `for x in gen { ... } { ... }` (or use `.into_iter()` explicitly as seen in examples) to consume the yields. In Python, you use the generator directly in a loop: `for x in count(): ...` will implicitly call `__iter__`/`__next__` on the generator. Under the hood, both are doing something analogous – repeatedly resuming the generator until it's exhausted. For asynchronous coroutines, Python uses `async for` to loop over an async generator, whereas Rust will likely integrate with the existing `Stream` trait (from the `futures` library) or a similar `AsyncIterator` if/when `async gen` becomes available. Rust's eventual async streams might use a similar approach (polling in a loop under the hood), but at present Rust doesn't have built-in syntax for `async for` (one would use `.await` in a loop manually or external crates to bridge futures to streams).

- **Awaiting vs Yielding:** In Python, an *async coroutine* (created with `async def`) cannot use `yield` directly for producing values (except in the form `yield` inside `async def` which creates an async generator). Python distinguishes between generators (for iteration) and coroutines (for async) at the syntax level (`def ... yield ...` vs `async def ... await ...`). Rust's approach is more uniform under the hood – the same coroutine mechanism can do both – but in syntax Rust also uses distinct keywords (`gen` vs `async`). One difference is that Rust currently doesn't allow mixing `await` and `yield` in the same function easily (there is no stable concept of an async generator yet), whereas Python *does* allow `await` inside an async generator function. For example, in Python you can write:

```python
async def fetch_and_yield(session, urls):
    for url in urls:
        data = await session.get(url)
        yield data  # yields results asynchronously
```

  This is an async generator (denoted by the combination of `async def` and `yield`). Rust would require future language support to do something analogous, like an `async gen fn` or `gen async` block [21]. The Rust RFC authors have indicated that such a feature is intended eventually, but it will come with limitations (e.g. you might not be allowed to hold certain references across an `.await` and a `yield` without pinning) [22].

Design Philosophy

- **Static vs Dynamic Typing:** Perhaps the biggest difference is that Rust's generators are compiled with static types for yields and returns. The type of value yielded from a `gen` block must be known and consistent for every `yield` (or use a sum type like an enum if you need to yield different variants). In Python, a generator can yield values of any type (and even vary types across yields), since Python is dynamically typed. Rust's design brings the benefits of compile-time type checking – for instance, the compiler will enforce that a `gen` block's yield type matches the function's return `impl Iterator<Item=T>` signature, etc. – and potentially better performance (no boxing of yield values unless needed, and no dynamic dispatch unless you choose to hide the type). The flip side is a bit more verbosity in Rust's declarations and sometimes the need for type annotations, whereas Python's approach favors brevity and flexibility at runtime.

- **Explicitness and Clarity:** Rust tends to favor explicit markers for special behavior. By requiring the `gen` keyword (just as it requires `async` for async functions), Rust makes it clear when a function or block will have suspend/resume behavior. Python's simplicity – where `yield` doubles as both an "exit" and "entry" point in the function's execution [25] – is very convenient, but it can sometimes lead to surprising behavior if one isn't expecting a function to be a generator. Rust's approach ensures that the presence of a generator is syntactically visible at the call site (if using a `gen` block inline, it's obvious; if using a `gen fn`, that would be in the signature). This design philosophy aligns with Rust's general emphasis on making control flow and side-effects explicit.

- **Performance and Implementation:** Rust's generators are implemented as **zero-cost abstractions**. The state of a Rust generator is stored in a struct (often on the stack or heap, depending on usage) with fields for each local variable that needs to persist across yields. The Rust compiler's generator transformation creates an efficient state machine with jumps (very much like how it implements async/await). This means that a Rust generator's runtime performance is comparable to hand-written iterator code in most cases. Python's generators, by contrast, are higher overhead: each `next()` call in Python re-enters the interpreter, executes bytecode until a `yield` is encountered, and so on. They are quite efficient for Python, but Python's interpreter and dynamic typing have inherent overhead that Rust's compiled, optimized code doesn't have. The difference reflects philosophy: Rust aims to make even high-level constructs as performant as possible, whereas Python opts for maximal ease-of-use with the understanding that it runs in a higher-latency interpreter environment.

- **Memory Safety and Concurrency:** Rust's design ensures that *generators are memory-safe* without a garbage collector. The borrow checker works with generators to prevent, for example, yielding a reference that could be invalidated when the generator is resumed later. This often necessitates using `Pin` when dealing with the low-level `Coroutine` trait – to prevent moving a self-referential generator – but the high-level `Iterator` interface abstracts that away for common usage. Python, on the other hand, relies on its garbage collector and runtime to handle lifetimes. For instance, if a Python generator yields a reference to some object, the Python GC must keep that object alive until the generator is done. Rust's compile-time checks provide stronger guarantees up front, but also make certain patterns (like holding a reference across a yield) statically illegal unless carefully managed. In terms of concurrency, Rust generators (if they don't capture non-`Send` data) can be sent across threads freely, and multiple generators can be run truly in parallel on different threads. Python's generators are bound by the Global Interpreter Lock (GIL), meaning even if you use threading, two Python generator executions won't truly run in parallel in one process. Rust's fearless concurrency model extends to generators as well — they compose with threads and `Send`/`Sync` as any other type would, which is a significant difference from Python's single-threaded coroutine execution model.

Capabilities and Features

- **Bidirectional Communication:** Python generators support sending values *into* the generator. A Python generator's `yield` can be used as an expression that receives a value sent by the caller on resumption (using the `generator.send(val)` method). This allows a form of two-way communication: the generator yields a value out, and can accept a new value in on the next resume. Rust's current `gen` blocks do **not** support this kind of direct bidirectional exchange in stable code. There is no stable API to send a value into a running generator each time you resume it – the `Iterator` abstraction only pulls values out. However, the underlying coroutine mechanism is designed to allow this (the resume argument `R` in `Coroutine<R>` [10]). In fact, if you use the unstable `Coroutine` trait directly, you *can* resume with a value of a given type. Stabilizing that in a nice syntax is still to be decided (perhaps via `gen fn` arguments or some `yield` expression form). In short, Python currently has the edge in ergonomics for coroutines that need to both send and receive data in each step, whereas Rust's model is at the moment one-directional for stable generators (from generator to caller). We expect Rust to narrow this gap as the coroutine feature evolves.

- **Exception Handling and Cancellation:** Python generators have methods like `throw()` and `close()` which allow the caller to inject exceptions into the generator or prematurely terminate it.

For example, `gen.throw(MyError)` will make the `yield` expression in the generator raise `MyError` at that suspension point, allowing the generator to handle or propagate it. Rust's generators don't have an exact equivalent mechanism. If a Rust generator needs to handle errors, it would likely yield `Result<T, E>` types or use the `?` operator internally to return early on error (the `gen` design does allow using `?` to break out of the generator early with an error, which would end the iteration)[21]. If a caller wants to stop a Rust generator early, typically they just stop iterating (drop the iterator), which will drop the generator value – any cleanup in the generator's destructor (`Drop` impl) would then run. There isn't a way to force a generator to throw an exception to an inner scope because Rust doesn't use exceptions; instead, Rust would encourage error handling through results or panics. A panic inside a Rust generator will unwind just as it would in normal code, possibly aborting iteration. Overall, Python provides more direct control to externally influence a generator's execution (for better or worse), while Rust treats a generator more like any other iterator where control flow is driven by yielding and returning rather than external intervention. Rust's approach is more static and structured; for example, to simulate Python's `close()`, you might design your generator to check a flag and return if it's set, but you can't asynchronously inject that signal without cooperation.

- **Use in Async Context:** Both Rust and Python have the concept of *asynchronous coroutines*, but they manifest differently. Python's `async def` coroutines and `asyncio` use `await` and can be thought of as specialized generators under the hood (historically, Python's `await` was built on top of `yield` with the `@asyncio.coroutine` decorator and `yield from` semantics before Python had native syntax). Python also has **async generators** (`async def` with `yield` statements), which allow yielding multiple times in an async context. Rust's equivalent to `async def` is `async fn`, and it produces a `Future` that can be `.await`ed. A Rust `async fn` can only return a single value (or Result) when awaited, not a stream of values; to produce multiple async values you currently use the `Stream` trait from the `futures` crate or manual methods. The proposed Rust `gen` feature could bridge that gap by allowing a natural syntax for streams. The main difference in philosophy is that Python built both sync and async generators into the language (with unified semantics for `yield` and `await` being separate but composable concepts), whereas Rust first built the low-level primitives (the generator trait for futures) and is now gradually exposing a high-level syntax for both patterns (sync generators now, async generators likely later). One similarity is that *neither* language begins executing a coroutine until it's awaited or iterated. In Python, calling a generator function or an async function doesn't run its body immediately – it returns a generator/coroutine object that you must iterate or schedule. Rust is the same: `gen { ... }` returns an inert generator that does nothing until iteration, and calling an `async fn` returns a future that does nothing until polled/awaited[5]. This lazy execution model is common to both, avoiding doing work until the user explicitly drives the coroutine forward.

Summary of Similarities and Differences

Both Rust and Python recognize the utility of generators and coroutines for writing clearer code that produces or handles sequences of values. They share the concept of suspending and resuming execution (via `yield` in Python and Rust, and additionally `await` in async contexts). A Rust generator and a Python generator both allow a function's execution state to be preserved between outputs, and both integrate with the respective language's iteration constructs (Rust's `for`, Python's `for` loop). In fact, Rust's motivation for `gen` blocks is very much like Python's motivation for generator functions: to make lazy iteration simple and intuitive[26].

However, the two differ in implementation and scope. Python's approach is dynamic and highly flexible – a generator is a sort of all-in-one object that can yield values, receive values, be closed, etc., with the trade-off of runtime overhead and less compile-time error checking. Rust's approach requires more upfront syntax and thought (deciding on yield types, perhaps dealing with lifetimes if yields involve references), but in return you get compile-time guarantees and performance. The design philosophies are reflected even in the naming: Python doesn't explicitly call them "coroutines" for the synchronous case, they're just generator functions by virtue of `yield`. Rust splits the terminology: it uses **"generator"** for the restricted form meant for iterators, and **"coroutine"** for the broader feature that includes asynchronous tasks and potentially more powerful generators [4]. This means experienced Rust developers will think of `gen` blocks as a convenient iterator builder, and see coroutines as the underlying mechanism that also powers `async/.await`. An experienced Python developer, on the other hand, sees no difference in mechanism between a generator function and, conceptually, a coroutine – they're all just functions that can pause (indeed, in Python, generator-based coroutines were a thing before native async syntax).

In capability, right now Python's generators are actually more general in some ways (due to `send`/`throw` and easy async generator support), but Rust is catching up fast. The Rust community's active RFCs and proposals suggest that soon Rust will have its own answers to those features (for example, an ergonomic `Stream` in std for async generators, and possibly syntax to allow resume arguments or `gen fn` definitions). The end result will be that Rust developers can enjoy writing generator functions in a style not too dissimilar from Python, but with Rust's signature twist of compile-time safety and efficiency. As the Rust Coroutine RFC notes, coroutines (generators) in Rust are intended not only as a foundation for async/await, but also to provide "an ergonomic definition for iterators and other primitives" [8]. The `gen` keyword is a concrete step toward that goal, bringing Rust's iterator convenience closer to that of Python's, while maintaining Rust's distinct approach to reliability and performance.

References:

- Rust RFC 3513 – *"gen blocks"* (reserving the `gen` keyword and introducing generator blocks) [27] [3].
- Inside Rust Blog (Oct 2023) – *"Generators are dead, long live coroutines, generators are back"* (discusses renaming `Generator` trait to `Coroutine` and plans for new generator syntax) [28] [7].
- Rust Unstable Book – *"generators"* (explains the experimental generator feature and yield semantics) [29] [13].
- Rust std::ops::Coroutine trait documentation [8] [10].
- Rust Lang discussions on async & generators (RFC 2033 and others) [14] [21].
- Python reference on generators (behavior of `yield`) [25] and official Python documentation on generator functions [30].

---

[1] [2] [3] [5] [6] [16] [17] [18] [21] [22] [23] [26] [27] 3513-gen-blocks - The Rust RFC Book
https://rust-lang.github.io/rfcs/3513-gen-blocks.html

[4] [7] [11] [24] [28] Generators are dead, long live coroutines, generators are back | Inside Rust Blog
https://blog.rust-lang.org/inside-rust/2023/10/23/coroutines/

[8] [9] [10] Coroutine in std::ops - Rust
https://doc.rust-lang.org/std/ops/trait.Coroutine.html

[12] [13] [29] generators - The Rust Unstable Book
https://dev-doc.rust-lang.org/beta/unstable-book/language-features/generators.html

[14] [15] 2033-experimental-coroutines - The Rust RFC Book
https://rust-lang.github.io/rfcs/2033-experimental-coroutines.html

[19] [20] Unified coroutines a.k.a. Generator resume arguments by semtexzv · Pull Request #2781 · rust-lang/rfcs · GitHub
https://github.com/rust-lang/rfcs/pull/2781

[25] generator — Python Reference (The Right Way) 0.1 documentation
http://python-reference.readthedocs.io/en/latest/docs/generator/

[30] Generators - Python Wiki
https://wiki.python.org/moin/Generators