Building a cron parser in Python

D Batten · Follow
9 min read · Oct 24, 2023

If you've ever worked with crontabs before, chances are you've been on https://crontab.guru/ or similar to decipher someone else's cron schedule expression, or to verify the expression you've just written will be running every day for the next 3 months as required, not every month for the next 3 years.

In this article we'll first introduce cron, then we'll explore what a cron schedule expression is and what all the symbols and special characters mean, and finally we'll design a simple Python application to parse a cron expression and return it human readable form.

What is cron and what is it used for?

cron is a Linux utility that allows tasks to be automatically run in the background at predefined time intervals. A cron job or cron schedule is a set of instructions that specifices when a particular task (or tasks) should be run. A crontab (CRON TABle) is a file which contains all the scheduled cronjobs on the system.

Let's take a look at an example cron job:

```
*/15 0 1,15 * 1-5 /usr/bin/find
```

In this instance, `/usr/bin/find` is the task and everything that comes before it (`*/15 0 1,15 * 1-5`) defines when the task will be executed. The part of the expression that defines when the task is executed is known as the cron schedule expression (CSE).

Breaking down the cron schedule expression

As mentioned earlier, the CSE defines when the task will be executed. The CSE looks very uninviting by itself, so let's break it down.

The CSE is made up of 5 space separated components. These components refer to the minute, hour, day of month, month, and day of week of the schedule in order, see Figure 1.

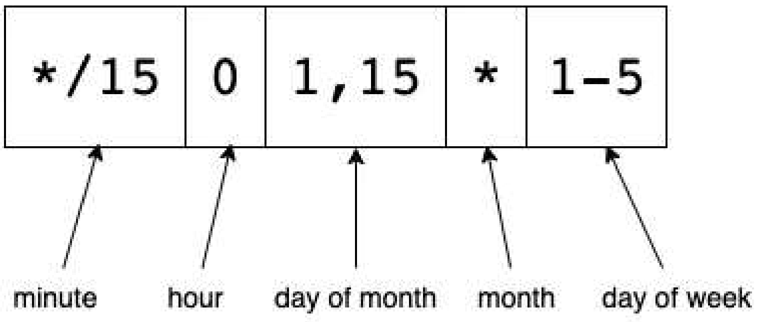

Fig 1: components of a CSE

Now let's break the expression down further to explore what the numbers and special characters mean.

If a component is made up of a single number, as is the case for the hour component in this example, then the task will be scheduled for the value of the corresponding component. For example, if `hour = 9` then the task will run in hour 9 (i.e. between 9am and 10am), if `day of week = 3` then the task will run on the 3rd day of the week (i.e. Wednesday). In this example, `hour = 0`, so we know our task will run in the 0th hour i.e. some time(s) between midnight and 1am.

Now for the special characters:

```
+------+----------------------+-------------------------------------+
| Char |     Description      | Example (assuming minute component) |
+------+----------------------+-------------------------------------+
| *    | Any value            | Every minute of the hour            |
| ,    | Value list separator | 10,20: the 10th and 20th minute     |
| -    | Range of values      | 10-25: from 10 until 25 past hour   |
| /    | Step values          | */5: every 5th minute               |
+------+----------------------+-------------------------------------+
```

Note that these special characters can be applied to any of the different time components. Special care has to be taken to ensure that the value is valid for the given component, for example a `10-20` value wouldn't make sense for the month component because there are only 12 valid options for month: the numbers from 1–12.

We're now in a position to decipher the CSE from the example:

```
+--------------+-------+-----------------------------------+
|  Component   | Value |            Description            |
+--------------+-------+-----------------------------------+
| Minute       | */15  | Every 15th minute                 |
| Hour         | 0     | Hour 0                             |
| Day of month | 1,15  | The 1st and 15th days of the month |
| Month        | *     | Every month                        |
| Day of week  | 1-5   | Monday to Friday                   |
+--------------+-------+------------------------------------+
```

So in human readable form, the cron job in the example means:

Run `/usr/bin/find` at every 15th minute between 0:00 and 1:00 on the 1st and the 15th of every month as long as it's a weekday.

Writing a Python cron parser application

To help us improve our cron expression understanding going forward, let's write a Python application with a CLI that can take a cron expression, expand the schedule expression and return a nice readable format for us to understand. Interaction with the application should look like the following:

```
~$ ./cron-parser.py "*/15 0 1,15 * 1-5 /usr/bin/find"
minute        0 15 30 45
hour          0
day of month  1 15
month         1 2 3 4 5 6 7 8 9 10 11 12
day of week   1 2 3 4 5
command       /usr/bin/find
```

With the SOLID principles in mind, in particular single responsibility, we start by thinking about the concerns of the application. Two main concerns jump out straight away, one being the ability to parse the expression, and the other being the ability to display the information in a table format. To ensure our approach is maintainable, readable, testable, and extendable, we'll use an object orientated approach and encapsulate the responsibilities in classes.

Let's start by considering the parsing class. Where possible, I like to consider classes as nouns rather than verbs. So for example, a `CronExpression` class which implements something like an `expand` instance method, as opposed to a `CronExpressionExpander` class. Here we have two different "flavours" of `CronExpression` class. The first is the raw, unexpanded form. The second is the expanded form. Given that both flavours are concretions of the same cron expression abstraction and will share a lot of the same behaviour (such as the `minute` , `hour` etc. properties) it makes sense to define the shared behaviours in an abstract class and have both the raw and expanded concretions inherit from the abstract class. Figure 2 shows the proposed class hierarchy diagram.

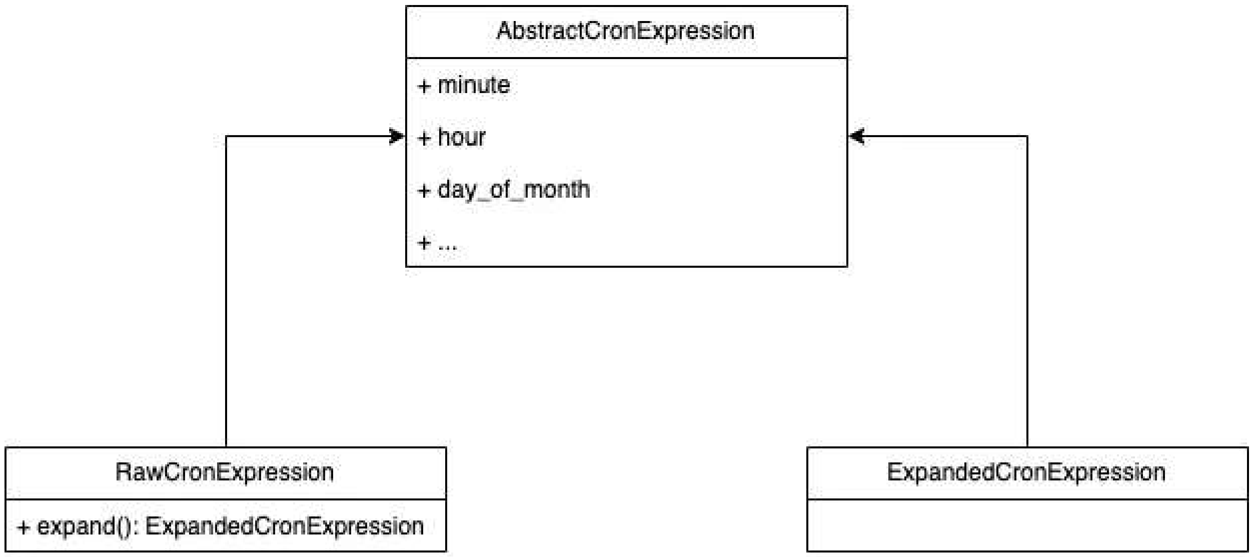

Fig 2: class hierarchy diagram for the AbstractCronExpression class and children.

The concretions can then be interacted with like so:

```
expression = "*/15 0 1,15 * 1-5 /usr/bin/find"
raw = RawCronExpression(expression)
raw.minute  # returns "*/15"
expanded = ExpandedCronExpression(expression)
expanded.minute  # returns [0, 15, 30, 45]
```

The public interface for both the raw and expanded cron expression instances will be largely the same and will be defined in the parent class, however the `RawCronExpression` class should implement an `expand` instance method which will return an instance of the `ExpandedCronExpression`. This design will allow for both the raw and expanded expressions to be passed around the application, so e.g. modifying the output table to instead display the unexpanded forms will be trivial. This helps us satisfy the Open/Closed principle.

The `ExpandedCronExpression` is then tasked with expanding the time components into list form. The logic for this expansion could live in a helper function in a `utils` module which could be triggered from the constructor of the `ExpandedCronExpression` class. I like to think of helper functions as mercenaries; they don't care about what the output of their work is used for, only that they do their job when called upon and do it well (and that they get paid, I guess that's where the analogy breaks down).

The expanding of special characters would be best handled using regular expressions so that the logic remains concise and compact. Given that the logic needs to know what the expression is as well as what the available options are (since e.g. a `*` for minute should return different values compared to a `*` for hour), the helper function signature could look like the following:

```
def expand_expression(
    expression: str, options: Union[List[str], List[int]]
) -> Union[List[str], List[int]]:
    """Expand a cron schedule expression component."""

# * for minute
expand_expression("*", list(range(60))  # should return [0, 1, ..., 59]
# dash interval for day of week
expand_expression("1-5", list(range(1, 8))  # should return [1, 2, 3, 4, 5]
# slash interval for day of month
expand_expression("*/2", list(range(1, 31))  # should return [1, 3, ..., 31]
```

To fill out the body of the function, we'll go one-by-one through the special characters.

The implementation for `*` is trivial:

```
if expression == "*":
    return options
```

For dash intervals we can use the regex `r"^(\d{1,2})-(\d{1,2})$"` which will match a one or two digit number anchored at the start, followed by a dash, followed by another one or two digit number anchored at the end. The brackets allow us to extract the numbers via the match group:

```
dash_matches = re.search(r"^(\d{1,2})-(\d{1,2})$", expression)
if dash_matches:
    mini, maxi = int(dash_matches.group(1)), int(dash_matches.group(2))
    return list(range(mini, maxi + 1))
```

We can catch both a single one or two digit number and a value separator list by using the regex `r"^\d{1,2}(?:,\d{1,2})*$"` . This will match a one or two digit number anchored at the start of the expression, followed by an optional amount of one or two digit numbers preceeded by a comma. So for example, all of `1`, `10`, `1,10,20` will be matched. Note that we need to split the string by `","` to ensure we return a list of strings, and not a single string of comma separated values.

```
comma_matches = re.findall(r"^\d{1,2}(?:,\d{1,2})*$", expression)
if comma_matches:
    return comma_matches[0].split(",")
```

The slash interval is a little more tricky. We want to capture either a `*` or a dash interval followed by a slash followed by one or two digits. This can be done with the regex `r"^(\*|\d{1,2}-\d{1,2})/(\d{1,2})$"` . The tricky bit here is that our options are now affected by whether it's a `*` or a dash interval. To solve this we can call the `expand_expression` function from within the function, passing the character preceeding the slash and the original options to get the new options. We can then step through the new options using the number after the slash. The comments in the snippet below show the variable values as we step through the function for the example described in the first line of the snippet.

```
# expression = 1-5/2, options = [1, 2, 3, 4, 5, 6, 7]
interval_matches = re.search(r"^(\*|\d{1,2}-\d{1,2})/(\d{1,2})$", expression)
if interval_matches:
    new_options = expand_expression(
        interval_matches.group(1),  # 1-5
        options  # [1, 2, 3, 4, 5, 6, 7]
    )
```

```
        # new_options = [1, 2, 3, 4, 5]
        interval = int(interval_matches.group(2))
        # interval = 2
        return new_options[::interval]  # [1, 3, 5]
```

It would be wise to cover some errors here as well, for example if an unrecognised or invalid (e.g. `5-1`) expression was passed. In this case it would be best to raise some custom exceptions, one for unrecgonised and one for invalid so that it can be caught further up the program and bubbled up to the client.

Now that all the components of the CSE can been expanded, all that's left to do is format the expanded values into a multi-line string table structure. A simple class that accepts a list of tuples in the format `(name of field, field values)` and implements a render method to return the multiline string could look like the following:

```
def _generate_buffered_col(name: str, length: int) -> str:
    return name + " " * (length - len(name))


class TableOutput:
    def __init__(
        self,
        table_data: List[Tuple[str, Union[str, List[Any]]]],
        name_col_length: int = 14,
    ):
        self.table_data = table_data
        self.name_col_length = name_col_length

    def render(self) -> str:
        """Render the data as a multi-line string."""
        out = ""
        for name, value in self.table_data:
            if isinstance(value, list):
                value = " ".join([str(x) for x in value])
            row = f"{_generate_buffered_col(name, self.name_col_length)} {value}
            out += row
        return out.rstrip()
```

It accepts a `name_col_length` argument to adjust the width of the field name column.

The final piece of the puzzle would be to implement a concrete method in the `AbstractCronExpression` class which would convert the cron expression to the required list of tuples format. Better yet, this could be done in a dedicated class which could accept an instance of `AbstractCronExpression` and return the list of tuples. This would better adhere to single responsibility as one could argue that it's not the responsibility of the `AbstractCronExpression` class to output its own data into a specific format.

For the sake of simplicity though, let's implement a concrete method in the `AbstractCronExpression` class:

```
def to_table_format(self) -> List[Tuple[str, Union[str, List[Any]]]]:
    """Return the argument in a table format."""
    return [
        ("minute", self.minute),
        ("hour", self.hour),
        ("day of month", self.dom),
        ("month", self.month),
        ("day of week", self.dow),
        ("command", self.command),
    ]
```

All the components of the application are now available to be linked together and exposed through a public API:

```
def expand_cron_expression(cron_expression: str) -> str:
    """Expand and return a formatted cron expression."""
    table_data = RawCronArgument(cron_expression).expand().to_table_format()
    return TableOutput(table_data).render()
```

This could then be invoked by a command line interface which would ultimately be responsible for displaying the output to the screen.

Of course, all classes and helper functions should be thoroughly unit tested. It may also be helpful to have some integration tests to ensure that the end-to-end flow is working as expected.

Written by D Batten

0 Followers