In the world of software development, we often rely on other sets of program files called packages. Packages facilitate development time with code reusability. Another important feature is that they can be shared globally as external libraries for other developers, so they don't have to write everything from scratch. As the complexity grows, so does the number of packages we have to integrate. Packages get updated for various reasons; updating them might break our existing codebase. But we might miss out on important features or patches if we don't update them.

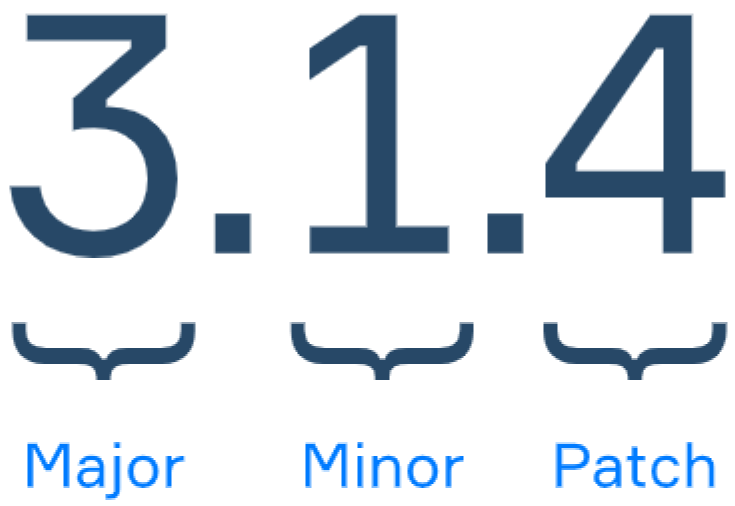
Luckily, we don't have to guess if an update breaks our code or not because semantic versioning is here to solve that problem. It is a way of versioning software releases so that we know exactly what kind of update it is.

Semantic versioning structure

The basic structure of semantic versioning, also known as semver, https://semver.org/

goes like this: Major.Minor.Patch.

For instance, let's have a look at a package called pie.



From the version number alone, we can get the following information.

- The major version release is 3. This shows that the major version was released 3 times. And each major version had at least 1 incompatible change.
- The minor version release is 1. It shows that one backward-compatible feature updates with the current major version.
- The Patch version is 4, which means it underwent four bug fixes that are backward-compatible with the current minor version.

Prefixing a semantic version with a "v" is a common way to indicate it is a version number. However, "v1.2.3" is not a semantic version; the letter "v" is a prefix, and the semantic version is "1.2.3".

Major version

The first digit of the version number must be a positive integer number unless it is the pre-release or unstable version. In this case, it starts from 0.

```
1.0.0       // a new release
```

```
0.1.0-alpha // pre release
```

When an update introduces backward-incompatible changes that break the compatibility of the code, the major version is incremented, and minor and patch versions are reset to 0.

If our pie package goes through a major update, its version will be:

```
3.1.4 // before
```

```
4.0.0 // after the major update
```

Minor version

The second digit of the version number starts from zero if the major version is greater than zero. If the major version is zero, then the minor version starts from one.

`1.0.0`

`0.1.0`

The minor version is incremented when backward-compatible feature updates are introduced. In this case, the package functions the same way as before but with additional features. Incrementing the minor version will reset the patch version to 0.

If our pie package gets a feature update.

`3.1.4 // before`

`3.2.0 // after the feature update`

Patch version

The third digit of the version number starts from zero. It gets incremented for every patch introduced to the package. Patches are a way to address security vulnerabilities and bug fixes.

Any change that breaks compatibility will increment the major version, even if it is just a patch.

A patch for our pie package.

`3.1.4 // before`

`3.1.5 // after the patch`

Conclusion

To sum up, in this topic, we've learned how to version packages according to semantic versioning specifications:

- Major version: incremented when backward-incompatible changes are introduced to the package.

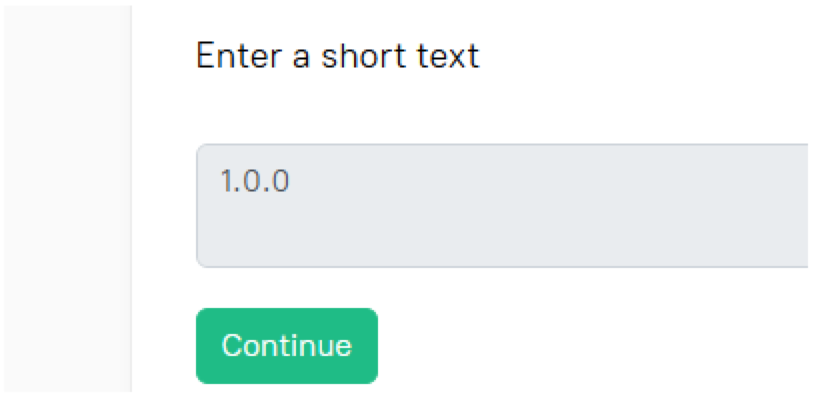
- **Minor version: incremented when backward-compatible feature updates are made.**
- **Patch versions: incremented when backward-compatible bug fixes occur.**

We can update without breaking compatibility with the help of semantic versioning.

Now, let's practice semantic versioning!

You have been working on a very interesting project under the pre-release version, and now you are ready to launch a stable version.

What should your version number be according to semantic versioning specification?


Enter a short text

1.0.0

According to the **Semantic Versioning (SemVer) specification**:

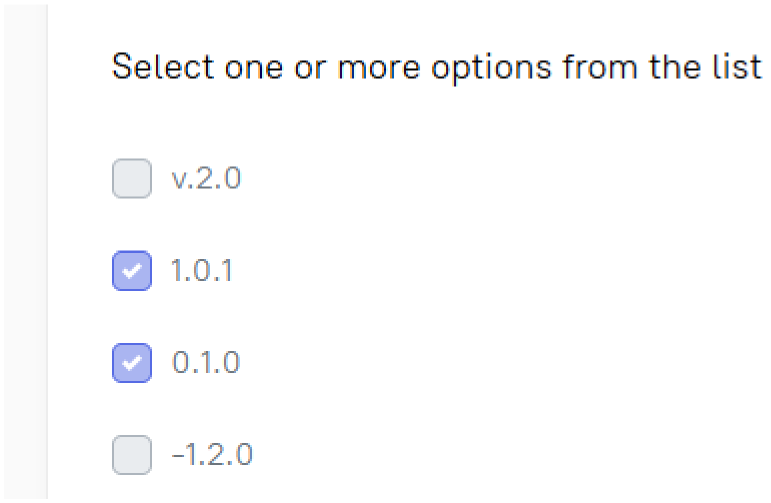
- Pre-release versions are typically denoted as `1.0.0-alpha`, `1.0.0-beta`, `1.0.0-rc.1`, etc.

- When you are ready to launch the **first stable release**, you **drop the pre-release label** and release it as:

```
1.0.0
```

Select the
 semantic versioning pattern.
Major.Minor.Patch

Please select all
valid
 versions according to semantic versioning specifications.



Select one or more options from the list

- [ ] v.2.0
- [✓ 1.0.1]
- [✓ 0.1.0]
- [ ] -1.2.0

`v.2.0` → Invalid (the `v` and dot are not part of SemVer; should be `2.0.0`).

`-1.2.0` → Invalid (negative numbers are not allowed).

`1.0.1` → Valid (follows MAJOR.MINOR.PATCH).

`0.1.0` → Valid (also valid for initial development).

Arrange the following version numbers according to semantic versioning specifications in ascending order.

Put the items in the correct order

- 1.0.0
- 1.4.16
- 1.5.0
- 2.0.0

Feature update

Report a typo

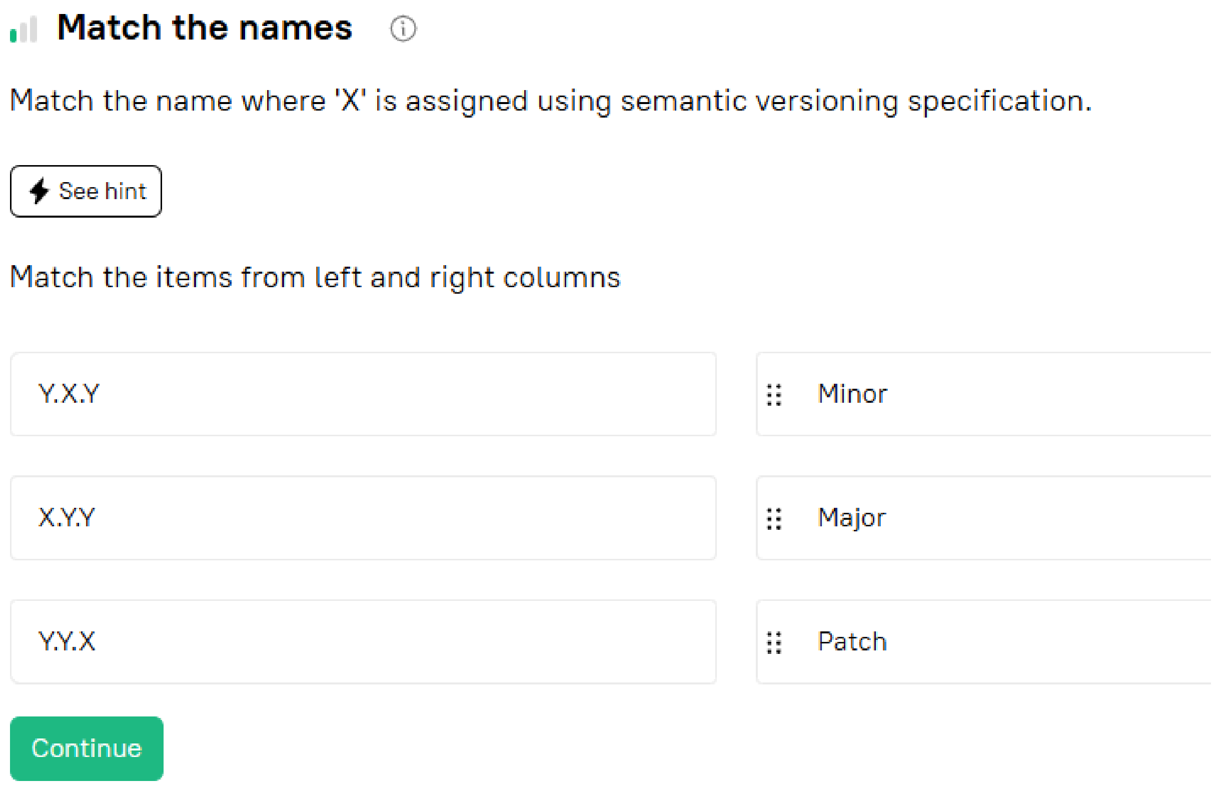
A package called pie is currently at version `4.3.2`, and it went through a feature update that doesn't break compatibility.

What will be the updated version of the package?

- [✓ 4.4.0]
- 4.3.3
- 4.4.3
- 5.0.0

Continue

Match the name where 'X' is assigned using semantic versioning specification.



Match the names

Match the name where 'X' is assigned using semantic versioning specification.

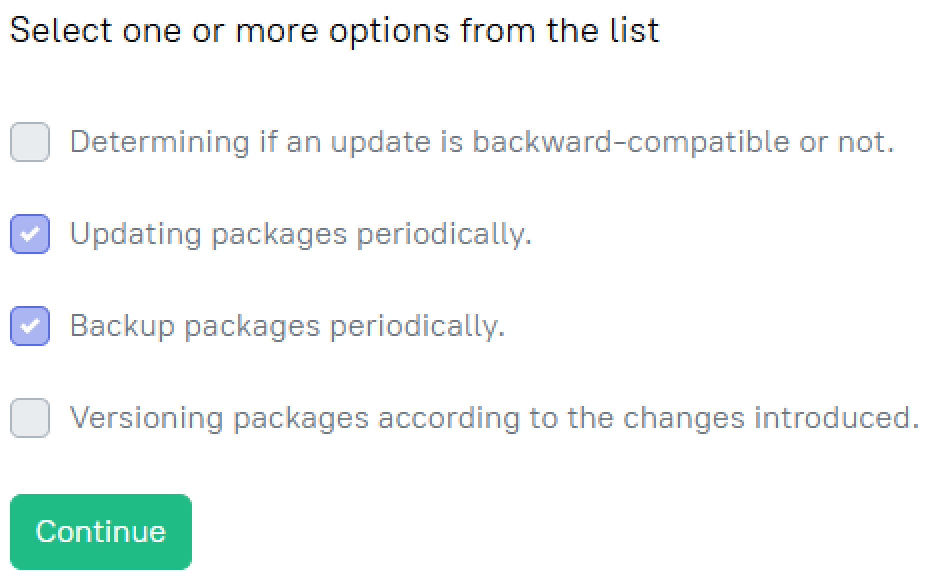
See hint

Match the items from left and right columns

Y.X.Y	Minor
X.Y.Y	Major
Y.Y.X	Patch

Continue

Which of the following options is semantic versioning not used to represent?



Select one or more options from the list

- ☐ Determining if an update is backward-compatible or not.
- ☑ Updating packages periodically.
- ☑ Backup packages periodically.
- ☐ Versioning packages according to the changes introduced.

Continue