progressindicator Documentation

Release 0.1.2

Priyam Singh

May 25, 2017

Contents

CHAPTER 1

Introduction

This Package provides ProgressIndicator class for easily providing visual cue about the progress of a task underway. Using the concept of components and providers, One can easily customize the look and feel of the Progress Indicator.

Features include:

- Full customization of the look and feel of the Progress Indicator
- Although by default a large number of components are provided in the package, you can easily create new ones.

CHAPTER 2

Links

- Documentation
- Source
- API Reference

CHAPTER 3

Contents

Installation

```
pip install progressindicator
```

Quickstart

Displaying a ProgressBar

Using a *Progress bar* in your code is very easy. Let us see some ways by which we can show a functioning progress bar during a task.

Usage In a Loop

You can use an `AdvancedProgressBar` instance named *bar* to display a ProgressBar. Here we are publishing the percentage of the task completed to *bar*. Note that we need to call `begin()` and `end()` at appropriate time.

```
from progress_manager import AdvancedProgressBar
bar = AdvancedProgressBar()
bar.begin()
for i in range(1000):
    bar.publish((i+1)/10)
bar.end()
```

Wrapping a Generator

Let us define a generator that just yields numbers from 0 to n.

```
def generator(n):
    for x in range(n):
        yield x
```

You can wrap this generator with a `ProgressManager` instance and you should get a nice progress indicator. Note that you should use a Indeterminate progress bar since there is no way to automatically calculate the progress of a generator since it can even be infinite.

```
for i in bar(generator(n)):
    time.sleep(0.01)
```

Also Note that, there is no need to call `begin()`, `publish()`, or `end()`.

Wrapping a Iterator

Iterators can also be wrapped with a ProgressBar instance. Similar to a generator there is no need to call `begin()`, `publish()`, or `end()`. If the iterator provides a `__len__` implementation, The progress is automatically calculated, like in the following case.

```
for i in bar(range(n)):
    time.sleep(0.01)
```

Using with statement

To avoid calling `begin()` and `end()`, you can wrap your code using a `with` statement that ensures that they are automatically called at appropriate times.

```
with AdvancedProgressBar() as bar:
    for i in range(n):
        time.sleep(0.01)
        bar.publish(100*(i+1)/n)
```

Creating a Custom ProgressBar

The default PrgressBar is enough for most purposes. But if you need to customize the look of the ProgressBar, you can do so via `extensions`.

Building The ProgressBar

`ProgressIndicator` takes a parameter named components which is an iterable of string or extensions. Using this, you can customize the look of your progress bar as per your wish.

```
from progressindicator.core import ProgressIndicator
from progressindicator.extensions import Percentage, Bar
bar = ProgressIndicator(components=[Bar(), "Progress =", Percentage()])
```

Built-in Extensions

A large number of extensions are provided by default. More details on them can be found in the API Reference.

- `Bar`

- BouncingBar
- Ellipses
- Alternator
- Spinner
- Loader
- Timer
- ETA
- Rate
- Percentage

Writing your own Extensions

Although the extensions provided by default should be enough, but you can always create your own extensions by subclassing BaseExtension. More detail on this can be found in the API Reference.

Every extension should call the __init__() method of the *BaseExtension* class by passing a list of tags as requirements. These tags are basically a string. Following is the list of all built-in tags.

- TAG_VALUE
- TAG_MIN_VALUE
- TAG_MAX_VALUE
- TAG_BEGIN_TIME
- TAG_END_TIME
- TAG_ITERATIONS
- TAG_PERCENTAGE
- TAG_TIME_SINCE_BEGIN
- TAG_DELTATIME
- TAG_LAST_UPDATED_AT
- TAG_TIME_SINCE_UPDATE
- TAG_ETA
- TAG_ETA1
- TAG_RATE

You can then override several event methods of BaseExtension, such as on_begin(), on_update(), on_validated(), on_invalidated(), on_end() to suit your needs. In each of these methods you recieve a list of values corresponding to the requirements you passed in the __init__() method. Note that on_validated() and on_invalidated() are called by the default implementation of on_update(). If you override on_update(), those methods will no longer be called unless you call them explicitly. In general, you should use on_validated() and on_invalidated() for most of the purposes. To set the string that is to be displayed by your extension, just call the set_value() method from your extension.

Let's write our own extension which prints nice messages to the screen depending on the percentage of the task completed.

```
class MyExtension(BaseExtension):
    def __init__(self):
        BaseExtension.__init__(self, requirements=[TAG_PERCENTAGE])

    def on_begin(self, params):
        self.set_value("Task has begun")

    def on_validated(self, params):
        if params[0] > 50 and params[0] < 90:
            self.set_value("Task is half completed")
        elif params[0] > 90:
            self.set_value("Task is almost completed")

    def on_end(self, params):
        self.set_value("Task is finished")
```

As you can see how easy it is to create your own extension to customize the look of the Progress indicator according to your needs.

Writing your own Providers

But what if you want need some parameter which is not provided by the built-in tags? You can also create Custom Providers to calculate values and specify a new tag for them that can be used by other extensions.

Creating a provider is similar to creating an extension. But note that, instead of `on_update`, here we can override `on_publish()`. Also the `__init__()` method does not take `update_interval` as a parameter, Instead it takes a parameter `tag` which takes a string. Rest of api is same. The tag should not collide with any built-in tag. Prior to using a provider you need to register it. To register, just call the `register_provider()` method of the *ProgressIndicator* class and pass it an instance of your provider.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search