bitpit Documentation

Release 1.2.0

Mohammad Alghafli

Oct 13, 2019

Contents

This tutorial gives an introduction to how to use bitpit python library and its features.

This is not a python tutorial. You are expected to have general knowledge in python before you start this tutorial.

If you are looking for quick guide and do not want to spend much time reading, have a look at the quick guide section.

CHAPTER 1

Content

1.1 Installation

1.1.1 Requirements

The major requirement of bitpit is python 3. bitpit is a python 3 library and was never tested in python 2. So first make sure your version of python is 3.

1.1.2 Installation

Make sure you have pip for python 3 installed.

On windows install using pip by running the command:

```
pip install bitpit
```

Or on linux:

```
pip3 install bitpit
```

Of course, pip command should be in your PATH environment variable. If you are using windows there is a good chance pip is not in your PATH. In this case you should specify the full pip path. Search about how to use pip on windows if you are having trouble.

Try to import bitpit to be sure it was installed successfully:

```
>>> import bitpit
>>>
```

If it is imported without errors, you are ready to use it. You may want to have a look at one or more of the following documents:

- *Quick Guide* **For those who want short and quick highlights and usage** example.

- *Download a File* **This is the start of the library tutorial. It shows** different library features.
- *bitpit Reference* **This is the library reference. All classes and functions** documentation is here.

1.2 Quick Guide

This is a quick guide to use the library. Read it if you want to have a quick look in the library and do not want to spend much time here.

bitpit is an event driven http download library with automatic resume and other features. The library is written in an event-driven style similar to GTK+.

1.2.1 Usage example

This is a typical usage example:

```
import bitpit

#will download this
url = 'https://www.python.org/static/img/python-logo.png'
d = bitpit.Downloader(url) #downloader instance

#listen to download events and call a function whenever an event happens
#print state when state changes
d.listen(
        'state-changed',
        lambda var, old_state: print('download state:', var.state)
    )

#print speed in human readable format whenever speed changes
#speed is updated and callback is called every 1 second by default
d.listen(
        'speed-changed',
        lambda var: print('download speed:', *var.human_speed)
    )

#register another callback function to the speed change signal
#print percentage downloaded whenever speed changes
d.listen('speed-changed', lambda var: print(int(var.percentage), '%'))

#print total file size in human readable format when the downloader knows the file␣
↪size
d.listen(
        'size-changed',
        lambda var: print('total file size:', *var.human_size)
    )

#done registering callbacks. lets start our download
#the following call will not block. it will start a new download thread
d.start()

#do some other work while download is taking place...

#wait for download completion or error
d.join()
```

1.2.2 As main script

This module can also be run as a main python script to download a file. You can have a look at the main function for another usage example.

commandline syntax:

```
python -m bitpit.py <url>
```

args:

- url: the url to download.

1.2.3 Other arguments

Most of what you can do is done by passing the desired args to `Downloader.__init__()`. Here are most of the args you can use:

1. url: URL to download
2. **path: The path to download the file at. if not supplied, will guess the file** name from the URL.
3. **restart_wait: Time to wait in case of error before the download is retried.** If not supplied, will never retry in case of error.
4. **update_period: The minimum time to wait before emitting *speed-changed*** signal. Defaults to 1 second.
5. timeout: Connection timeout. Defaults to 10 seconds.
6. **rate_limit: Maximum download bit rate. If not supplied, download without** speed limit.

1.2.4 Tutorial

In *Download a File* you will find a more comprehensive bitpit tutorial.

1.3 Download a File

So we have bitpit installed and ready. Let's start using it. In this tutorial we are going to make a little download program. It is a little bit similar to the downloader function but we will make it a little bit better.

First, we need to import the library:

```
import bitpit
```

Now let's specify the URL we are going to download. We are going to download python logo:

```
url = 'https://www.python.org/static/img/python-logo.png'
```

Next comes bitpit business. We create a `Downloader` instance:

```
dl = bitpit.Downloader(url)
```

Finally we start the download:

```
dl.start()
print('Download has started.')
```

Now the download will start. Notice that `Downloader.start()` call will not block. The message `Download has started.` will be printed immediately before the download finishes. Then our main thread will end but the downloading thread will keep running until the file is fully downloaded or an error occures.

If you try the example above, you will see `Download has started` message printed on the screen and nothing else. The program will freeze until the download finishes. Imagine if we have a very big file such as linux mint. It will take a long time without us knowing how much we have downloaded. That is not so convenient isn't it? We will look at that later but for now, let's look at the program we have written so far

```
import bitpit

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

In *Display Download Information*, we will make the program give us information about the download such as whether it has started or faced an error and also the download speed.

1.4 Display Download Information

At the moment we are able to download a file. But we have no information on how fast our download is and if it is completed or there is some error.

Before we start, here is the tiny program we made previously if you need to refresh your mind:

```
import bitpit

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

Now it is time to make it better.

1.4.1 Display the file size

If we are downloading a file, we probably want to know the file size. bitpit is written in an event driven style. It is a little similar to GTK library if you have used it before. We need to do 2 steps to show the file size. First, we need to define a function that will be called when the file size is known:

```
def on_size_changed(downloader):
    print(downloader.size)
```

This function takes 1 argument: `downloader` which is the `Downloader` instance that we just knew its file size. In the function, we print the `Downloader.size` property, which is just the file size in bytes.

Next, we need to tell the downloader to call this function as soon as it knows the file size. You probably want to do this just before you start the download. This is done using `Downloader.listen()` method:

```
dl.listen('size-changed', on_size_changed)
```

The `Downloader.listen()` takes at least 2 arguments. The first is the signal to listen to. Here we listened to the `size-changed` signal which is emitted whenever the downloader gets to know the size of the file being downloaded. The second argument is the function to call when the signal is emitted. Here we put the function we defined above.

After this call to `Downloader.listen()`, our function will be called as soon as the file size is known. Our full program now becomes as follows:

```
import bitpit

def on_size_changed(downloader):
    print('The file size is', downloader.size)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url)

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

If you notice, the size is expressed in bytes. Showing the size in bytes gives us a very big number that is difficult for humans to read. It would be easier for us if we could display the size in Kilobytes or Megabytes. This can be done by modifying the callback function `on_size_changed()` to be as follows:

```
def on_size_changed(downloader):
    print('The file size is', *downloader.human_size)
```

We just replaced `Downloader.size` property with `Downloader.human_size` property. `Downloader.human_size` property gives us a 2-element tuple. The first element is a float representing the size and the second element is a string suffix with the value KB for kilobytes or MB for megabytes and so on. In our call to `print()` function, we unpacked the tuple arguments using python * operator. If you are not familiar with this, check it out in the python here.

When I tried the new callback function, I got the following message printed:

```
The file size is 9.865234375 KB
```

We can use python string formatting to make it look better but we will leave it for later.

1.4.2 Display the download speed

Other than the size, we want to know the download speed. Similar to the size, we define a callback function and listen to a signal. The function we will define will print the speed just like the size. The property we will use is `Downloader.speed`. Also like the size, there is a `Downloader.human_speed`. We will use `Downloader.human_speed`:

```
def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)
```

The signal we want to listen to this time is `speed-changed`:

```
dl.listen('speed-changed', on_speed_changed)
```

The behaviour of `speed-changed` signal is a little bit different than `size-changed`. When the download starts, the signal is emitted every 1 second . It will keep being emitted periodically as long as the download is running. In our program, the signal will not work very well because the file size is very small. Try to download linux mint and you will see the signal working properly.

There are other things we can do to improve our program regarding `speed-changed` signal. For example, we can show how much we have downloaded so far in the callback function because we probably have downloaded something since the last time the signal was emitted. We can check `Downloader.downloaded` and `Downloader.human_downloaded` to know that. Furthermore, our callback will be printing a message every second which makes the terminal full of confusing text. We can make our output better. However, we will leave it to the end of the tutorial. For now we will stick to what we have done so far.

Now our program has become as follows:

```
import bitpit

def on_size_changed(downloader):
    print('The file size is', downloader.size)

def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url)

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)

#print speed periodically
dl.listen('speed-changed', on_speed_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

Just as a final note in this section, you can change the time between `speed-changed` signal emissions in `Downloader.__init__()` when you create the downloader instance by passing the desired number of seconds in the `update_period` argument. Check the class documentation for more details.

1.4.3 Display the download state

Another useful information we need in our download is its state. For example, did it start or not? Is it completed or still in progress? Did it stop normally or because of an error? This is what we are going to do.

Similar to the size and speed, we define a callback function and listen to a signal:

```
def on_state_changed(downloader, old_state):
    print('The state changed to:', downloader.state)


dl.listen('state-changed', on_state_changed)
```

Notice that `state-changed` signal takes at least 2 positional argumetns. The `Downloader` that changed state and the old state the downloader was on. The `state-changed` signal is emitted whenever the download is started, stopped, or completed. To know the new state, check the `Downloader.state` property. It can be one of the following: * `pause`: The download is not started or started then stopped by a calling `Downloader.stop()` method. * `start`: The download just started but is not download anything yet. * `download`: The download is running and in progress. * `error`: The download stopped bacause of an error. * `complete`: The download completed.

Our program now has become like this:

```
import bitpit

def on_size_changed(downloader):
    print('The file size is', downloader.size)

def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)

def on_state_changed(downloader, old_state):
    print('The state changed to:', downloader.state)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url)

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)

#print speed periodically
dl.listen('speed-changed', on_speed_changed)

#print state
dl.listen('state-changed', on_state_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

In *Automatic Restart*, we will make our downloader automatically resume the download when the download is interrupted due to an error.

1.5 Automatic Restart

So far, our program freezes until the download stops. However, when the program ends we are not sure whether the file is stopped because it is completely downloaded or because an error occured. What if an error occured and we want to restart the download again? This is easy. We just pass `restart_wait` argument to `Downloader.__init__()`:

```
dl = bitpit.Downloader(url, restart_wait=30)
```

This argument decides the time to wait before the downloader retries downloading when an error occures. It defaults to -1 if not given which means do not restart even after an error. Because we gave it the value 30 here, anytime an error happens, the downloader will wait for 30 seconds and then retry again. Try to download linux mint and shutdown your internet connection. Here is the output I got:

```
The file size is 1899528192
The speed is 0 B/s
The state changed to: start
The speed is 207.0622560278128 KB/s
The speed is 474.6406851817469 KB/s
The speed is 0 B/s
The state changed to: error
The file size is 1899528192
The speed is 0 B/s
The state changed to: start
The speed is 506.2438224533826 KB/s
The speed is 594.6743846283302 KB/s
```

You can see the state has changed to `error` after I shutdown my internet but the program did not terminate. After 30 seconds, the state changed again to `start` and the download continued. Now our program will only terminate when the download is successfully completed.

One last note, some connection errors are perminant. For instance, if you get a 404 NOT FOUND error, then no matter how many times you try, the error will keep happening. bitpit does not handle that and will keep trying to download regardless of the error. You can check the error that happened by looking at the `Downloader.last_exception` property. You will most probably get an exception from `requests.exceptions` module.

We have only changed 1 line in this lesson. Now our program so far has become:

```
import bitpit

def on_size_changed(downloader):
    print('The file size is', downloader.size)

def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)

def on_state_changed(downloader, old_state):
    print('The state changed to:', downloader.state)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(url, restart_wait=30)

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)
```

(continues on next page)

(continued from previous page)

```
#print speed periodically
dl.listen('speed-changed', on_speed_changed)

#print state
dl.listen('state-changed', on_state_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

We are getting closer to the end.

In *Specify Path and Rate Limit*, we will specify the path and name to save our file instead of saving it in the current directory with the default name. We will also start limiting the download speed instead of eating up all our internet bandwidth before my brother gets angry.

1.6 Specify Path and Rate Limit

So far our program gives us most of the information we need and also restarts when an error occures. There are 2 things we will do in this lesson: First we will specify where we want to save our file and second we want to limit the download speed so that the internet does not become slow for the rest of the family. I grouped the two in 1 lesson because both are straight forward.

1.6.1 Specify the file path

We want to decide where our file will be saved. This is done using the `path` argument to `Downloader.__init__()`:

```
dl = Downloader(url, path='~/Desktop/logo.png', restart_wait=30)
```

The above instruction tells the downloader to save the file in my desktop with the name `logo.png`. In case you do not know what ~ means in a path, it means the user home directory in linux systems. This will probably not work on windows. We can make a portable way that works in both linux and windows by importing and using `pathlib` standard python library:

```
dl = Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30
    )
```

If you are not familiar with `pathlib`, then you should have a look at this awsome library.

You notice that in our first modification above, we supplied a python string in the `path` argument. However, in our second modification, we gave a `pathlib.Path` object. The argument `path` can take both. In fact, you can give anything that `pathlib.Path.__ini__()` supports. If you want, you can also give a binary file-like object and the data will be saved in it.

1.6.2 Download rate limit

To limit the download rate, you simply give `rate_limit` argument to `Downloader.__init__()`:

```
dl = Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048
    )
```

In our example here, we made our maximum download speed 2 KB/s. Let's see the program output now:

```
The file size is 10102
The speed is 0 B/s
The state changed to: start
The speed is 1.9989241550312336 KB/s
The speed is 1.9988802572634587 KB/s
The speed is 1.9987825036005515 KB/s
The speed is 1.9989528185814844 KB/s
The file size is 10102
The speed is 0 B/s
The state changed to: complete
```

You can see that the download speed became very close to 2 KB/s (or a little less). However, note that this may not work as expected for small files.

Our full program so far became:

```
import bitpit
import pathlib

def on_size_changed(downloader):
    print('The file size is', downloader.size)

def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)

def on_state_changed(downloader, old_state):
    print('The state changed to:', downloader.state)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048
    )

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)

#print speed periodically
dl.listen('speed-changed', on_speed_changed)
```

(continues on next page)

(continued from previous page)

```
#print state
dl.listen('state-changed', on_state_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

In *Additional Tuning*, we will do our final tunes to our downloader.

1.7 Additional Tuning

Now we have most of our work done. We are going to look into a few minor additional things we can do to modify our downloader behaviour.

1.7.1 Connection Timeout

We can change the connection timeout settings by giving the `timeout` argument to `Downloader.__init__()`. The default value is 10 seconds. That is relatively small. Let's make it 1 minute:

```
dl = bitpit.Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048,
        timeout=60
    )
```

1.7.2 Chunk Size

We can also supply the download `chunk_size` to `Downloader.__init__()`. The chunk size is the maximum number of bytes to download in a single network read operation. You do not really need to change this at all but just in case you want to change it. Having very low or very high values may slightly affect download speed. There is no hard rule to figure out the best other than trying. In my computer, the default value worked best. The default value is 4 KB. For practice, let's change it to 1 KB:

```
dl = bitpit.Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048,
        timeout=60,
        chunk_size=1024
    )
```

The `chunk_size` cannot be greater than `rate_limit`. If it is greater, *bitpit* will force it to be equal to `rate_limit`.

Here is our program so far:

```
import bitpit
import pathlib

def on_size_changed(downloader):
    print('The file size is', downloader.size)

def on_speed_changed(downloader):
    print('The speed is', *downloader.human_speed)

def on_state_changed(downloader, old_state):
    print('The state changed to:', downloader.state)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048,
        timeout=60,
        chunk_size=1024
    )

#listen to signals
#print size as soon as it is known
dl.listen('size-changed', on_size_changed)

#print speed periodically
dl.listen('speed-changed', on_speed_changed)

#print state
dl.listen('state-changed', on_state_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

Now we have only one thing left to do. If you have noticed, our output is ugly.

In *Elegant Output* we are going to make it pretty. We will also introduce some useful things in *bitpit*.

1.8 Elegant Output

We are finally in the last lesson. Let's make our output beautiful.

Our goal will be to make the output look like this:

```
<state> | <File size> | <Downloaded> | <speed> [<progress bar>] <percentage>% <eta>
```

We have all the information in 1 line seperated by a pipe character "|". The state will show us in real time if there is any error. The file size, downloaded bytes and speed will be in human readable form so that we can easily read it. The progress bar will indicate how much portion we have downloaded so far. The percentage will indicate the same

as the progress bar but in numbers. Finally, `eta` is the estimated time to finish the download. The information will be printed in only 1 line. If it changes, we will make the information be updated in the same line instead of printing so many lines like we did in the `on_speed_changed` callback.

1.8.1 Showing information in one line

First, instead of having a callback function for each signal, let's make 1 callback that will update all the information whenever 1 thing changes. Let's remove `on_size_changed`, `on_speed_changed` and `on_state_changed` callbacks and write 1 callback to print the `state`, `size`, `downloaded`, and `speed` instead:

```
def on_anything_changed(downloader, old_state=None):
    state = downloader.state
    size = '{} {}'.format(*downloader.human_size)
    downloaded = '{} {}'.format(*downloader.human_downloaded)
    speed = '{} {}'.format(*downloader.human_speed)

    text = '{} | {} | {} | {}'.format(state, size, downloaded, speed)
    print(text)
```

We will do the progress bar, the percentage and the estimated download time in a later section.

Next, we modify all `Downloader.listen()` calls to register the new function:

```
#listen to everything
dl.listen('size-changed', on_anything_changed)
dl.listen('speed-changed', on_anything_changed)
dl.listen('state-changed', on_anything_changed)
```

Now our callback will be called when the state changes, when we know the size and periodically when `speed-change` signal is emitted. Notice that we also printed number of bytes `downloaded` which we did not do in previous lessons. Now our output will be something like this:

```
pause | 9.865234375 KB | 0 B | 0 B/s
start | 9.865234375 KB | 0 B | 0 B/s
start | 9.865234375 KB | 0 B | 0 B/s
start | 9.865234375 KB | 2.0 KB | 1.9990254031527928 KB/s
start | 9.865234375 KB | 4.0 KB | 1.9989961600987338 KB/s
start | 9.865234375 KB | 6.0 KB | 1.9988544205541783 KB/s
start | 9.865234375 KB | 8.0 KB | 1.9987502773920875 KB/s
start | 9.865234375 KB | 9.865234375 KB | 1.9987502773920875 KB/s
complete | 9.865234375 KB | 9.865234375 KB | 0 B/s
complete | 9.865234375 KB | 9.865234375 KB | 0 B/s
```

Ok. We still have ugly output. First, let's make all numbers rounded to 2 decimal places. In the callback, we will modify our `format strings`:

```
state = downloader.state
size = '{:0.2f} {}'.format(*downloader.human_size)
downloaded = '{:0.2f} {}'.format(*downloader.human_downloaded)
speed = '{:0.2f} {}'.format(*downloader.human_speed)
```

Second, we do not want to print multiple lines. We want to print only 1 line. Let's use the `print` function arguments to stay on the same line and use the character `\r` to update it:

```
text = '\r{} | {} | {} | {}'.format(state, size, downloaded, speed)
print(text, end='', flush=True)
```

Now our callback will not print many lines. Instead, it will go back to the beginning of the line and print the information on the same line erasing anything previously shown.

Furthermore, let's modify the `print` call to print spaces to fill all the line with 79 characters just to erase the whole line in case we have garbage out of our text width:

```
print(text.ljust(79), end='', flush=True)
```

Our callback now becomes:

```
def on_anything_changed(downloader, old_state=None):
    state = downloader.state
    size = '{:0.2f} {}'.format(*downloader.human_size)
    downloaded = '{:0.2f} {}'.format(*downloader.human_downloaded)
    speed = '{:0.2f} {}'.format(*downloader.human_speed)

    text = '\r{} | {} | {} | {}'.format(state, size, downloaded, speed)
    print(text.ljust(79), end='', flush=True)
```

1.8.2 Showing the progress bar, percentage and ETA

Let's start with the progress bar. We use `Downlaoder.bar()` function to generate a progress bar. The function takes 2 optional arguments. The first is `width` which is the length in characters of the progress bar. It defaults to 30. Let's make it 10. The second is `char` which is the character to use to fill the bar. It defaults to '='. Let's make this a dash instead:

```
bar = downloader.bar(width=10, char='-')
```

Our callback now becomes:

```
def on_anything_changed(downloader, old_state=None):
    state = downloader.state
    size = '{:0.2f} {}'.format(*downloader.human_size)
    downloaded = '{:0.2f} {}'.format(*downloader.human_downloaded)
    speed = '{:0.2f} {}'.format(*downloader.human_speed)
    bar = downloader.bar(width=10, char='-')

    text = '\r{} | {} | {} | {} [{}]'.format(
            state,
            size,
            downloaded,
            speed,
            bar
        )
    print(text.ljust(79), end='', flush=True)
```

Notice how we enclosed the progress bar in brackes within our format string.

Percentage and ETA are straight forward. We use `Downloader.percentage` and `Downloader.eta` properties of the downloader:

```
percentage = int(downloader.percentage)
eta = downloader.eta
```

`Downloader.percentage` property returns the percentage (from 0 to 100) as a `float`. we converted it to `int` to remove any digits after the decimal point to reduce user confusion. eta returns a `datetime.timedelta` instance which tells us the estimated time remaining until the download is completed.

Now our full callback function becomes:

```
def on_anything_changed(downloader, old_state=None):
    state = downloader.state
    size = '{:0.2f} {}'.format(*downloader.human_size)
    downloaded = '{:0.2f} {}'.format(*downloader.human_downloaded)
    speed = '{:0.2f} {}'.format(*downloader.human_speed)
    bar = downloader.bar(width=10, char='-')
    percentage = int(downloader.percentage)
    eta = downloader.eta

    text = '\r{} | {} | {} | {} [{}] {}% {}'.format(
            state,
            size,
            downloaded,
            speed,
            bar,
            percentage,
            eta
        )
    print(text.ljust(79), end='', flush=True)
```

And now, this is our awesome program:

```
import bitpit
import pathlib

def on_anything_changed(downloader, old_state=None):
    state = downloader.state
    size = '{:0.2f} {}'.format(*downloader.human_size)
    downloaded = '{:0.2f} {}'.format(*downloader.human_downloaded)
    speed = '{:0.2f} {}'.format(*downloader.human_speed)
    bar = downloader.bar(width=10, char='-')
    percentage = int(downloader.percentage)
    eta = downloader.eta

    text = '\r{} | {} | {} | {} [{}] {}% {}'.format(
            state,
            size,
            downloaded,
            speed,
            bar,
            percentage,
            eta
        )
    print(text.ljust(79), end='', flush=True)

#will download this
url = 'https://www.python.org/static/img/python-logo.png'

#this is our downloader
dl = bitpit.Downloader(
        url,
        path=pathlib.Path.home() / 'Desktop' / 'logo.png',
        restart_wait=30,
        rate_limit=2048,
        timeout=60,
        chunk_size=1024
```

(continues on next page)

(continued from previous page)

```
    )

#listen to everything
dl.listen('size-changed', on_anything_changed)
dl.listen('speed-changed', on_anything_changed)
dl.listen('state-changed', on_anything_changed)

#start downloading and tell user download has started.
dl.start()
print('Download has started.')

#end of the main thread
```

The output I got from this program is below:

```
start | 9.87 KB | 4.00 KB | 2.00 KB/s [----      ] 40% 0:00:02.934069
```

You can see that fractions of a second are shown in `eta` which is not very nice. However, I will leave this to you to fix.

Finally we have an awesome download program. Of course, there are many things we can improve on it. But I believe this form is enough to explain bitpit features and how to use it.

You may want to have a look at *bitpit Reference* for complete documentation of the library.

THE END...

1.9 bitpit Reference

Date 2019-10-13

Version 1.2.0

Authors

- Mohammad Alghafli <thebsom@gmail.com>

Event driven http download library with automatic resume and other features. The goal of this module is to ease the process of downloading files and resuming interrupted downloads. The library is written in an event-driven style similar to GTK. The module defines the class `Downloader`. Instances of this class download a file from an http server and call callback functions whenever an event happens ralated to this download. Examples of events are download state change (start, pause, complete, error) and download speed change. The following is a typical usage example:

```
import bitpit

#will download this
url = 'https://www.python.org/static/img/python-logo.png'
d = bitpit.Downloader(url) #downloader instance

#listen to download events and call a function whenever an event happens
#print state when state changes
d.listen('state-changed', lambda var: print('download state:', var.state))

#print speed in human readable format whenever speed changes
#speed is updated and callback is called every 1 second by default
d.listen('speed-changed', lambda var: print('download speed:', *var.human_speed))
```

(continues on next page)

(continued from previous page)

```
#register another callback function to the speed change signal
#print percentage downloaded whenever speed changes
d.listen('speed-changed', lambda var: print(int(var.percentage), '%'))

#print total file size in human readable format when the downloader knows the file␣
↪size
d.listen('size-changed', lambda var: print('total file size:', *var.human_size))

#done registering callbacks. lets start our download
#the following call will not block. it will start a new download thread
d.start()

#do some other work while download is taking place...

#wait for download completion or error
d.join()
```

This module can also be run as a main python script to download a file. You can have a look at the main function for another usage example.

commandline syntax:

```
python -m bitpit.py [-r rate_limit] [-m max_running] url [url ...]
```

args:

- url: one or more urls to download.
- -r rate_limit: total rate limit for all running downloads.
- -m max_running: maximum number of running downloads at any single time.

class bitpit.**Downloader**(*url*, *path=None*, *dir_path=False*, *rate_limit=0*, *timeout=10*, *update_period=1*, *restart_wait=-1*, *chunk_size=4096*)

downloader class. instances of this class are able to download files from an http or https server in a dedicated thread, pause download and resume download. it subclasses `Emitter`.

in addition to listen and unlisten, you probably want to use the following methods:

- `self.start()`
- `self.stop()`
- `self.join()`
- `self.bar()`

properties:

name	type	access	description
url	*str*	RW	url to download. cannot be set if *is_alive* is True.
path	*pathlib.Path* or *io.BufferedIOBase*	RW	path to download at. if an instance of *pathlib.Path*, file will be opened and content will be written to it. the file is closed whenever the download stops (completion, pause or error). if it is an instance of *io.BufferedIOBase*, content is written to the object and the object is never closed. cannot be set if *is_alive* property is True.
restart_wait	*int*	RW	number of seconds to wait before restarting the download in case of error. setting it when a restart thread is active will restart the thread again.
restart_time	*datetime.datetime* or *None*	R	the time when the download will be restarted. *None* if there is no scheduled restart.
chunk_size	*int*	RW	number of bytes to write in a single write operation. ok to keep default value. when set, new value takes effect in the next time the download is started.
update_period	*int*	RW	*speed-changed* signal is emitted every this number of seconds.
timeout	*int*	RW	download will interrupt when no bytes are recieved for this number of seconds. when set, new value takes effect in the next time the download is started.
rate_limit	*int*	RW	speed limit of the downloads in bytes per second. may not work well with small files.
human_rate_limit	*tuple*	R	same as *rate_limit* but as human readable tuple. eg. (100.0, 'KB/s').
size	*int*	R	total size of the file being downloaded in bytes. -1 if unknown.
human_size	*tuple*	R	same as *size* but as human readable tuple.
downloaded	*int*	R	bytes downloaded so far.

signals:

- **state-changed: emitted when `state` property changes. its callback** takes 2 positional arguments, the `Downloader` instance which emitted the signal and the old state the `Downloader` was in.
- **size-changed: emitted when `size` property changes. its callback takes** 1 positional argument, the `Downloader` instance which emitted the signal.
- **speed-changed: emitted when `speed` property changes. its callback** takes 1 positional argument, the `Downloader` instance which emitted the signal.
- **url-changed: emitted when `url` property changes. its callback takes 1** positional argument, the `Downloader` instance which emitted the signal.
- **path-changed: emitted when `path` property changes. its callback takes** 1 positional argument, the `Downloader` instance which emitted the signal.
- **restart-time-changed: emitted when `restart_time` property changes.** its callback takes 1 positional argument, the `Downloader` instance which emitted the signal.
- **rate-limit-changed: emitted when `rate_limit` property changes. its** callback takes 1 positional argument, the `Downloader` instance which emitted the signal.

`bar`(*width=30*, *char='='*, *unknown='?'*)

returns a string of width `width` representing a progress bar. the string is filled with `char` and spaces. the number of `char` represents the part of the file downloaded (e.g., if half of the file is downloaded, half of the string will be filled with `char`). the rest of the string will be filled with spaces. if the ratio of downloaded data is not known, returns a string of width `width` filled with the `unknown` argument.

args:

- width (int): number of characters in the bar.
- char (str) character to fill the bar with.
- **unknown (str): character to fill the bar if the ratio downloaded is** unknown.

returns: a string containing `width` characters filled with `char` and spaces to show the ratio of the downloaded bytes to the total file size.

examples: if the width is 8 and 25% of the file is downloaded, the returned string will be '== '

if the width is 8 and the ratio downloaded is not known, the returned string will be '????????'

`join`(*timeout=None*)

waits until the downloading thread terminates for any reason (download completion, error or pause). check `self.state` after join if you want to know the state of the download.

args:

- **timeout (None or int) the timeout for the join operation. defaults** to None meaning no timeout.

`restart`(*wait=None*)

schedules a download restart and returns. it is called when an error occures during download and `self.restart_wait` property >= 0.

args:

- **wait (float or None): seconds to wait before the restart. if None,** uses `self.restart_wait`.

start()
starts a downloading thread. if `self.path` has data, the download will resume and bytes will be appended to the end of the file. does nothing if the downloader is already started. if there is a scheduled restart, it will be cancelled.

stop()
stops downloading thread. does nothing if the downloader is already stopped. if there is a scheduled restart, it will be cancelled.

update_size()
sends a head request to get the size of the file update `self.size`.

class bitpit.**Emitter**
a base class for classes that implement event driven programming. a derived class should define the class attribute `__signals__` which is a sequence of its valid signals.

emit(*signal*, **args*)
calls all callback functions previously registered for the signal by previous calls to `self.listen()`. emitting a signal not present in `__signals__` class property raises `KeyError`. exceptions raised by the callback function are printed to stderr and ignored.

args:

- signal (str): the signal to call its callbacks.
- **args: positional arguments to be passed to the callbacks. args** that were passed to `self.listen()` will be after `args` that are passed to this method.

listen(*signal*, *func*, **args*, ***kwargs*)
registers the callback function `func` for the signal `signal`. whenever the signal is emitted, the callback function will be called with 1 argument which is the object that emitted the signal. listening to a signal not present in class attribute `__signals__` raises `KeyError`. registering a callback function multiple times calls the function that number of times when the signal is emitted.

args:

- signal (str): the signal to listen to.
- func (a callable): the callback function.
- args: positional arguments to be passed to the callback.
- kwargs: keyword arguments to be passed to the callback.

unlisten(*signal*, *func*, **args*, ***kwargs*)
unregisters the callback function `func` for the signal `signal`. unlistening from an unknown signal raises `KeyError`. unlistening a callback which was not passed to `listen` method previously raises a `ValueError`. unlistening a call back will remove it from callback list only once. if the callback was passed to `self.listen()` multiple times, it must be unlistened that number of times to be completely removed from the callback list.

args:

- signal (str): the signal to unlisten from.
- func (a callable): the callback function.
- args: `args` that were passed to `self.listen()`.
- kwargs: `kwargs` that were passed to `self.listen()`.

class bitpit.**Manager**(*max_running=0*, *rate_limit=0*, *restart_wait=30*, ***kwargs*)
download manager class. multiple urls can be added to it. you can specify the maximum number of downloads that run at a single time and the manager will start or stop downloads to reach and not exceed this number.

you can also specify the total download rate limit and the manager class will equally divide the speed over the running downloads.

the `Manager` class subclasses `Emitter` and emits signals when a download is added or removed.

properties:

name	type	access	description
rate_limit	*int*	RW	rate limit for all running downloads. it will be divided equally over the them. a value <= 0 means no rate limit.
max_running	*int*	RW	maximum running downloads at a single time. if the number of started downloads exceed this number, the manager will stop some downloads. if the number is less than this number, the manager will start some downloads. a value <= 0 means no limit.
restart_wait	*float*	RW	minimum time before the manager starts the same download. even if max_running is not reached, if `restart_wait` has not passed since the download last stopped, the download not started immediately. the manager will wait until this number of seconds has passed then start the download. this is to prevent frequent restarts in case of network failure.
kwargs	*dict*	RW	keyword arguments to added downloads when creating an instance of *Downloader* using *self.add()*
downloads	*list*	R	downloads added to this manager. a list containing *Downloader* instances.

signals:

- **add: emitted when a new `Downloader` is added. the signal's callbacks** take 2 positional arguments, the `Manager` instance that emitted the signal and the `Downloader` that was just added. the added `Downloader` can be found in `self.downloads`.
- **remove: emitted when a `Downloader` is removed. the signal's callbacks** take 2 positional arguments, the `Manager` instance that emitted the signal and the `Downloader` that was just removed. the removed `Downloader` can no longer be found in `self.downloads`.
- **property-changed: emitted when *rate_limit*, *max_running*,** *restart_wait* or *kwargs* property is changed.

`add`(*d*)

add a new download to the manager.

args:

- **d (str or `Downloader`): the url or `Downloader` instance to** add. if `d` type is str, a new `Downloader` instance is created with arguments taken from *self.kwargs* property.

returns: the `Downloader` instance added.

`remove`(*d*)

remove a previously added download then emits `remove` signal. if the download is running, it is not stopped.

args:

- d (`Downloader`): the downloader to remove.

`start`()

start download manager thread. after a call to this method, the manager will start checking added downloads to start, stop and change rate limit when necessary.

stop()
stop the manager thread.

stop_all()
pause all currently running downloads. the manager thread is not stopped. if you want to stop the manager and all downloads, call `self.stop()` first.

update()
tell the manager thread to check pending downloads to see if there is need to start, stop or change rate limit to some of them. this is called automatically when the state of any added download changes and when manager properties are changed. you do not need to call it.

bitpit.**human_readable**(*n*, *digits=3*)
return a human readable number of bytes.

args:

- n (float): the number to return as human readable.
- digits (int): the number of digits before the decimal point.

returns:

tuple:

0. (float) human readable number or None if n is None.
1. (str) suffix or None if n is None.

bitpit.**main**(*urls*, *rate_limit='0'*, *max_running=5*)
downloads the given urls until done downloading them all. displays statistics about downloads in the following format: s | speed | downloaded | percent | eta | name

in the above format, the first item *s* is the first letter of the state of the download. for example, for complete downloads, that would be the letter *c*. Similarly, *e* would be for error and *f* for fatal error. *speed* is the download speed in human readable format. *downloaded* is the number of downloaded bytes in human readable format. *percent* is percentage downloaded. *eta* is estimated time to complete the download. *name* is the name of the file being downloaded or part of the name if the name is very long.

args:

- urls: the urls to download.
- rate_limit: total rate limit for all downloads
- max_running: maximum running downloads at any given time

1.10 Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index